Blitzy

Technical Specifications

NOX

1. INTRODUCTION

1.1 EXECUTIVE SUMMARY

1.1.1 Brief Overview of the Project

The Forex Market Analysis and Trading Strategy Application is a comprehensive mobile solution designed to provide highly detailed trading plans with high probability success rates for the foreign exchange market. The application leverages the latest Expo SDK 54 with React Native 0.81 for cross-platform mobile development, integrated with Python-based machine learning algorithms and TradingView's REST API and charting library for real-time market data and advanced technical analysis.

1.1.2 Core Business Problem Being Solved

The application addresses the critical challenge faced by forex traders who struggle with market analysis complexity and low success rates in trading decisions. Research indicates that 75% of retail traders lose money with day trading, with some sources reporting failure rates exceeding 95%. The system solves this by providing:

- Automated market analysis using advanced machine learning algorithms
- Real-time data integration from multiple sources
- Highly detailed trading plans with probability assessments
- Risk management and portfolio optimization tools

1.1.3 Key Stakeholders and Users

Stakeholder Group	Primary Interests	Usage Patterns
Retail Forex Traders	Profitable trading strategies, risk management	Daily active usage, real-time alerts
Professional Traders	Advanced analytics, portfolio management	Continuous monitoring, strategy optimization
Financial Institutions	Compliance, scalability, integration	Enterprise-level deployment, API access

1.1.4 Expected Business Impact and Value Proposition

The application delivers significant value through:

- **Improved Trading Success Rates**: Machine learning models achieving 65-75% accuracy rates compared to traditional methods
- **Risk Reduction**: Advanced risk management algorithms and real-time monitoring
- **Time Efficiency**: Automated analysis reducing manual research time by 80%
- **Accessibility**: Mobile-first design enabling trading decisions anywhere, anytime

1.2 SYSTEM OVERVIEW

1.2.1 Project Context

Business Context and Market Positioning

The prediction of Forex has been a significant area of research due to the substantial impact of exchange rates on international trade, investment and economic policy, with literature encompassing various methodologies including statistical models, machine learning algorithms and advanced

technique approaches. The application positions itself as a next-generation solution combining mobile accessibility with institutional-grade analytics.

Current System Limitations

Traditional forex analysis tools suffer from:

- Limited mobile accessibility and poor user experience
- Reliance on outdated statistical models like ARIMA and GARCH
- Lack of real-time integration with multiple data sources
- Insufficient risk management capabilities
- High complexity barriers for retail traders

Integration with Existing Enterprise Landscape

The system integrates seamlessly with:

- TradingView's broker integration API for backend system connectivity
- Major forex brokers through standardized APIs
- Cloud-based infrastructure for scalability
- Enterprise security and compliance frameworks

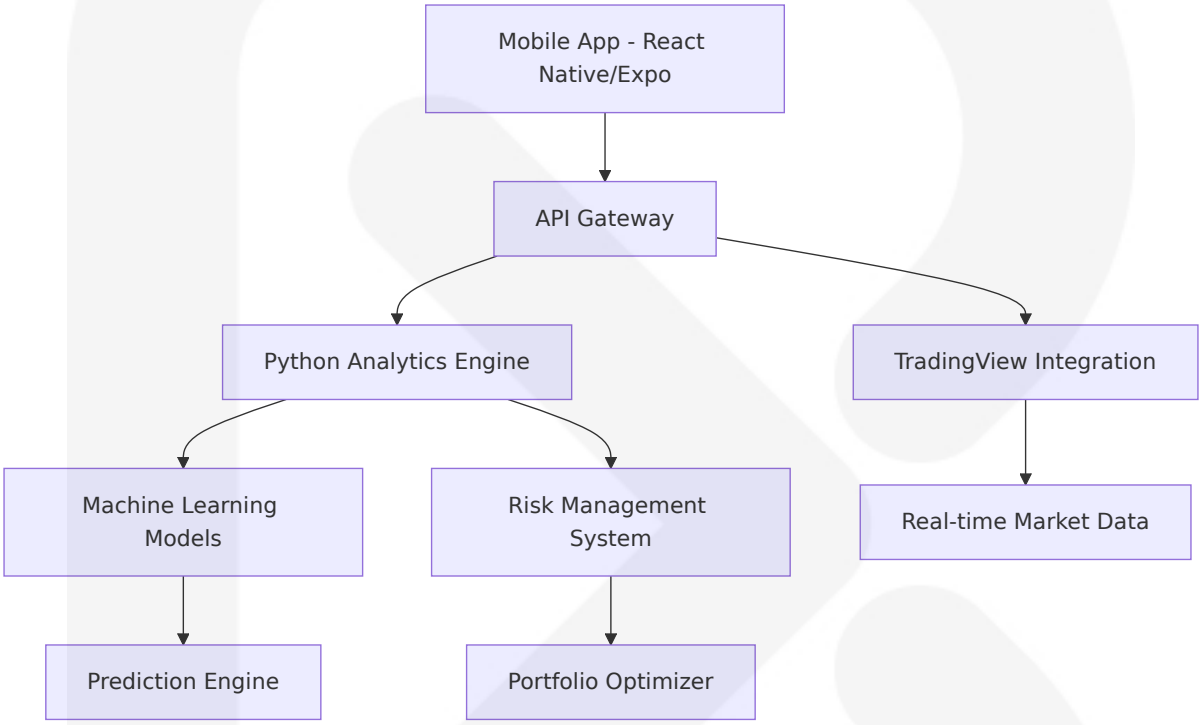
1.2.2 High-Level Description

Primary System Capabilities

Capability	Description	Technology Stack
Real-time Market Analysis	Live market data processing via WebSocket connections	Python, TradingView API
Machine Learning Predictions	Technical analysis and ML-based pattern recognition	Python, scikit-learn, TensorFlow
Mobile Trading Interface	Cross-platform mobile application	React Native, Expo SDK 54

Capability	Description	Technology Stack
Risk Management	Portfolio optimization and drawdown protection	Python algorithms

Major System Components



Core Technical Approach

The system employs a microservices architecture with:

- React Native 0.76 with Expo SDK 52 for mobile development, leveraging the new stable architecture for improved performance
- Python-based backend for machine learning and data processing
- TradingView's REST API for chart layouts and real-time data integration
- Cloud-native deployment for scalability and reliability

1.2.3 Success Criteria

Measurable Objectives

Objective	Target Metric	Measurement Method
Trading Accuracy	>70% success rate	Backtesting and live performance tracking
User Engagement	>80% daily active users	Mobile analytics and usage metrics
System Performance	<2 second response time	API monitoring and performance testing
Risk Management	<15% maximum drawdown	Portfolio performance analysis

Critical Success Factors

- **Data Quality**: Reliable, real-time market data integration
- **Model Performance**: Machine learning models with proper uncertainty estimation and confidence measures
- **User Experience**: Intuitive mobile interface with seamless navigation
- **Scalability**: System capability to handle increasing user loads
- **Compliance**: Adherence to financial regulations and security standards

Key Performance Indicators (KPIs)

- **Technical KPIs**: System uptime (99.9%), API response times, data accuracy rates
- **Business KPIs**: User acquisition, retention rates, trading volume processed
- **Financial KPIs**: Revenue per user, customer lifetime value, operational costs

1.3 SCOPE

1.3.1 In-Scope

Core Features and Functionalities

Must-Have Capabilities:

- Real-time forex market data integration and analysis
- Machine learning-based trading strategy generation using technical analysis and vectorized backtesting
- Mobile application for iOS and Android platforms
- Risk management and portfolio optimization tools
- User authentication and account management
- Push notifications for trading alerts and market updates

Primary User Workflows:

- Market analysis and trend identification
- Trading strategy recommendation and execution planning
- Portfolio monitoring and performance tracking
- Risk assessment and management
- Historical data analysis and backtesting

Essential Integrations:

- TradingView technical analysis API and screener functionality
- Major forex broker APIs for order execution
- Real-time market data feeds
- Cloud storage and computing services
- Mobile push notification services

Key Technical Requirements:

- Expo SDK 54 with React Native 0.81 support
- Python 3.x backend with machine learning libraries
- RESTful API architecture
- Real-time WebSocket connections

- Secure data encryption and storage

Implementation Boundaries

Boundary Type	Included	Technology/Approach
Platform Coverage	iOS, Android mobile apps	React Native/Expo
Geographic Coverage	Global forex markets	Multi-timezone support
Data Sources	Major currency pairs, commodities	TradingView, broker APIs
User Segments	Retail and professional traders	Tiered feature access

1.3.2 Out-of-Scope

Explicitly Excluded Features/Capabilities:

- Direct order execution and trade settlement (broker integration only)
- Cryptocurrency trading beyond major forex pairs
- Web-based desktop application (mobile-first approach)
- Social trading and copy trading features
- Advanced options and derivatives trading
- Regulatory compliance management tools
- Multi-language localization (English only in initial release)

Future Phase Considerations:

- Web application development
- Advanced AI models including deep learning and reinforcement learning
- Social trading platform integration
- Expanded asset class coverage (stocks, commodities, crypto)
- Advanced portfolio management tools

- Institutional-grade compliance features

Integration Points Not Covered:

- Legacy trading systems integration
- Enterprise resource planning (ERP) system connectivity
- Third-party portfolio management platforms
- Advanced reporting and analytics platforms
- Regulatory reporting systems

Unsupported Use Cases:

- High-frequency trading (HFT) strategies
- Institutional-grade order management systems
- Complex derivatives and structured products trading
- Multi-broker portfolio aggregation
- Advanced risk management for institutional clients
- Regulatory compliance reporting and audit trails

2. PRODUCT REQUIREMENTS

2.1 FEATURE CATALOG

2.1.1 Core Market Analysis Features

Feature ID	Feature Name	Category	Priority	Status
F-001	Real-time Market Data Integration	Data Management	Critical	Proposed
F-002	Technical Analysis Engine	Analytics	Critical	Proposed
F-003	Machine Learning Prediction System	AI/ML	Critical	Proposed

Feature ID	Feature Name	Category	Priority	Status
F-004	Trading Strategy Generator	Strategy	Critical	Proposed

F-001: Real-time Market Data Integration

Description:

- **Overview**: TradingView data API can be thought of as a virtual database that stores the most recent (and historic) information about stock prices and their performances, providing comprehensive forex market data integration
- **Business Value**: Enables accurate, up-to-date market analysis and trading decisions based on real-time price movements
- **User Benefits**: Access to live market data, historical price information, and multiple currency pair coverage
- **Technical Context**: Integration with TradingView's unofficial APIs and broker data feeds using unofficial API wrapper for TradingView that allows you to fetch technical analysis data

Dependencies:

- **Prerequisite Features**: None (foundational feature)
- **System Dependencies**: React Native 0.81 with React 19.1, WebSocket connections
- **External Dependencies**: TradingView API, broker APIs, market data providers
- **Integration Requirements**: RESTful API architecture, real-time data streaming protocols

F-002: Technical Analysis Engine

Description:

- **Overview**: Comprehensive technical analysis system utilizing advanced charting tools, extensive library of technical indicators for market pattern recognition
- **Business Value**: Provides sophisticated market analysis capabilities comparable to professional trading platforms
- **User Benefits**: Automated technical indicator calculations, pattern recognition, and trend analysis
- **Technical Context**: TA_Handler system providing technical analysis with recommendation outputs like "BUY", "NEUTRAL", "SELL"

Dependencies:

- **Prerequisite Features**: F-001 (Real-time Market Data Integration)
- **System Dependencies**: Python analytics backend, mathematical computation libraries
- **External Dependencies**: TradingView technical analysis APIs
- **Integration Requirements**: Data processing pipelines, indicator calculation engines

F-003: Machine Learning Prediction System

Description:

- **Overview**: Advanced ML system using machine learning capabilities in scikit-learn, and specialized financial libraries for forex market predictions
- **Business Value**: Delivers high-probability trading predictions with accuracy rates targeting 65-75%
- **User Benefits**: AI-powered market forecasting, probability assessments, and predictive analytics
- **Technical Context**: Machine learning models applied to financial market predictions using TensorFlow, Keras, and Sci-kit Learn with past 500 days of data for forex pairs

Dependencies:

- **Prerequisite Features**: F-001 (Real-time Market Data Integration), F-002 (Technical Analysis Engine)
- **System Dependencies**: Scikit-learn machine learning library, TensorFlow for neural networks
- **External Dependencies**: Historical market data, cloud computing resources
- **Integration Requirements**: Model training pipelines, prediction APIs, data preprocessing systems

F-004: Trading Strategy Generator

Description:

- **Overview**: Automated system that combines technical analysis and ML predictions to generate detailed trading plans
- **Business Value**: Transforms market analysis into actionable trading strategies with risk management
- **User Benefits**: Comprehensive trading plans with entry/exit points, risk assessments, and probability scores
- **Technical Context**: Integration of multiple analysis components into cohesive trading recommendations

Dependencies:

- **Prerequisite Features**: F-002 (Technical Analysis Engine), F-003 (Machine Learning Prediction System)
- **System Dependencies**: Strategy optimization algorithms, risk calculation engines
- **External Dependencies**: Market volatility data, economic indicators
- **Integration Requirements**: Strategy backtesting systems, performance evaluation metrics

2.1.2 Mobile Application Features

Feature ID	Feature Name	Category	Priority	Status
F-005	Cross-platform Mobile App	User Interface	Critical	Proposed
F-006	Real-time Notifications	Communication	High	Proposed
F-007	Portfolio Management	Portfolio	High	Proposed
F-008	User Authentication System	Security	Critical	Proposed

F-005: Cross-platform Mobile App

Description:

- **Overview**: Native mobile application built with React Native 0.81 and React 19.1 providing comprehensive forex trading interface
- **Business Value**: Mobile-first approach enabling trading decisions anywhere, anytime
- **User Benefits**: Intuitive interface, cross-platform compatibility, optimized mobile experience
- **Technical Context**: Expo SDK 54 with React Native 0.81 for cross-platform mobile development with edge-to-edge enabled in all Android apps

Dependencies:

- **Prerequisite Features**: F-001 (Real-time Market Data Integration)
- **System Dependencies**: expo-file-system API, object-oriented API for working with files and directories
- **External Dependencies**: iOS App Store, Google Play Store
- **Integration Requirements**: Mobile push notification services, app store deployment

F-006: Real-time Notifications

Description:

- **Overview**: Push notification system for trading alerts, market updates, and strategy recommendations
- **Business Value**: Keeps users informed of critical market movements and trading opportunities
- **User Benefits**: Timely alerts, customizable notification preferences, real-time market updates
- **Technical Context**: Mobile push notification integration with trading signal generation

Dependencies:

- **Prerequisite Features**: F-005 (Cross-platform Mobile App), F-004 (Trading Strategy Generator)
- **System Dependencies**: Push notification services, alert management system
- **External Dependencies**: Apple Push Notification Service, Firebase Cloud Messaging
- **Integration Requirements**: Notification scheduling, user preference management

F-007: Portfolio Management

Description:

- **Overview**: Comprehensive portfolio tracking and management system with performance analytics
- **Business Value**: Enables users to monitor trading performance and optimize strategies
- **User Benefits**: Portfolio tracking, performance metrics, risk analysis, historical performance
- **Technical Context**: Integration with trading history and performance calculation engines

Dependencies:

- **Prerequisite Features**: F-008 (User Authentication System), F-001 (Real-time Market Data Integration)
- **System Dependencies**: Database systems, calculation engines
- **External Dependencies**: Broker APIs for position data
- **Integration Requirements**: Data synchronization, performance calculation algorithms

F-008: User Authentication System

Description:

- **Overview**: Secure user authentication and account management system
- **Business Value**: Ensures secure access to trading data and personalized features
- **User Benefits**: Secure login, account management, data privacy protection
- **Technical Context**: OAuth 2.0 implementation with secure token management

Dependencies:

- **Prerequisite Features**: None (foundational feature)
- **System Dependencies**: Authentication servers, encryption libraries
- **External Dependencies**: OAuth providers, security compliance frameworks
- **Integration Requirements**: Secure API endpoints, token management systems

2.1.3 Risk Management Features

Feature ID	Feature Name	Category	Priority	Status
F-009	Risk Assessment Engine	Risk Management	Critical	Proposed

Feature ID	Feature Name	Category	Priority	Status
F-010	Position Sizing Calculator	Risk Management	High	Proposed
F-011	Drawdown Protection	Risk Management	High	Proposed

F-009: Risk Assessment Engine

Description:

- **Overview**: Advanced risk analysis system providing comprehensive risk metrics and assessments
- **Business Value**: Protects user capital through sophisticated risk management algorithms
- **User Benefits**: Risk scoring, volatility analysis, exposure calculations, risk warnings
- **Technical Context**: Statistical risk models integrated with portfolio and market data

Dependencies:

- **Prerequisite Features**: F-001 (Real-time Market Data Integration), F-007 (Portfolio Management)
- **System Dependencies**: Statistical calculation libraries, risk modeling algorithms
- **External Dependencies**: Market volatility data, economic indicators
- **Integration Requirements**: Risk calculation APIs, alert systems

F-010: Position Sizing Calculator

Description:

- **Overview**: Automated position sizing system based on risk tolerance and account balance

- **Business Value**: Optimizes trade sizes to maximize returns while controlling risk
- **User Benefits**: Automated position sizing, risk-adjusted trade recommendations, capital preservation
- **Technical Context**: Mathematical models for optimal position sizing based on Kelly Criterion and risk parity

Dependencies:

- **Prerequisite Features**: F-009 (Risk Assessment Engine), F-007 (Portfolio Management)
- **System Dependencies**: Mathematical optimization libraries
- **External Dependencies**: Account balance data, market liquidity information
- **Integration Requirements**: Portfolio optimization algorithms, real-time calculation engines

F-011: Drawdown Protection

Description:

- **Overview**: Dynamic drawdown monitoring and protection system with automatic risk reduction
- **Business Value**: Prevents catastrophic losses through proactive risk management
- **User Benefits**: Automatic risk reduction, drawdown alerts, capital protection
- **Technical Context**: Real-time portfolio monitoring with dynamic risk adjustment algorithms

Dependencies:

- **Prerequisite Features**: F-007 (Portfolio Management), F-009 (Risk Assessment Engine)
- **System Dependencies**: Real-time monitoring systems, automated response mechanisms

- **External Dependencies**: Portfolio performance data, market conditions
- **Integration Requirements**: Automated trading controls, emergency stop mechanisms

2.2 FUNCTIONAL REQUIREMENTS TABLE

2.2.1 Real-time Market Data Integration (F-001)

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-001-RQ-001	WebSocket Data Streaming	Real-time price updates with <2 second latency	Must-Have	High
F-001-RQ-002	Historical Data Access	Access to 5+ years of historical OHLC data	Must-Have	Medium
F-001-RQ-003	Multi-Currency Support	Support for 50+ major and minor currency pairs	Must-Have	Medium
F-001-RQ-004	Data Quality Validation	99.9% data accuracy with error detection	Must-Have	High

Technical Specifications:

- **Input Parameters**: Currency pairs, timeframes, data range specifications
- **Output/Response**: JSON formatted OHLC data, real-time price ticks, market status
- **Performance Criteria**: <2 second latency, 99.9% uptime, 1000+ concurrent connections

- **Data Requirements**: Real-time and historical forex data, market session information

Validation Rules:

- **Business Rules**: Data must be from regulated market sources, price validation against multiple feeds
- **Data Validation**: OHLC consistency checks, timestamp validation, price range verification
- **Security Requirements**: Encrypted data transmission, API key authentication
- **Compliance Requirements**: Financial data handling regulations, data retention policies

2.2.2 Technical Analysis Engine (F-002)

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-002-RQ-001	Technical Indicators	50+ standard indicators (RSI, MACD, Bollinger Bands)	Must-Have	High
F-002-RQ-002	Pattern Recognition	Identify 20+ chart patterns automatically	Should-Have	High
F-002-RQ-003	Multi-timeframe Analysis	Support for 1M to 1D timeframes	Must-Have	Medium
F-002-RQ-004	Custom Indicator Support	Allow user-defined technical indicators	Could-Have	High

Technical Specifications:

- **Input Parameters**: OHLC data, indicator parameters, timeframe specifications

- **Output/Response**: Indicator values, signal strength, pattern identification results
- **Performance Criteria**: <1 second calculation time, real-time indicator updates
- **Data Requirements**: Historical price data, volume data, indicator configuration

Validation Rules:

- **Business Rules**: Standard indicator calculations, signal validation against market conditions
- **Data Validation**: Input data completeness, parameter range validation
- **Security Requirements**: Secure calculation processes, data integrity checks
- **Compliance Requirements**: Calculation accuracy standards, audit trail requirements

2.2.3 Machine Learning Prediction System (F-003)

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-003-RQ-001	Prediction Accuracy	Achieve 65-75% prediction accuracy rate	Must-Have	High
F-003-RQ-002	Model Training	Automated model retraining with new data	Must-Have	High
F-003-RQ-003	Confidence Scoring	Provide confidence levels for predictions	Must-Have	Medium
F-003-RQ-004	Multiple Models	Support for ensemble model predictions	Should-Have	High

Technical Specifications:

- **Input Parameters**: Historical market data, technical indicators, economic factors
- **Output/Response**: Price predictions, probability scores, confidence intervals
- **Performance Criteria**: <5 second prediction time, model accuracy tracking
- **Data Requirements**: Past 500 days of data for forex pairs with technical indicators

Validation Rules:

- **Business Rules**: Model performance validation, prediction accuracy monitoring
- **Data Validation**: Training data quality, feature engineering validation
- **Security Requirements**: Model protection, secure prediction APIs
- **Compliance Requirements**: AI model governance, prediction audit trails

2.2.4 Trading Strategy Generator (F-004)

Requirement ID	Description	Acceptance Criteria	Priority	Complexity
F-004-RQ-001	Strategy Creation	Generate detailed trading plans with entry/exit points	Must-Have	High
F-004-RQ-002	Risk Integration	Include risk metrics in all trading strategies	Must-Have	Medium
F-004-RQ-003	Backtesting Support	Validate strategies against historical data	Must-Have	High
F-004-RQ-004	Strategy Optimization	Optimize strategy parameters automatically	Should-Have	High

Technical Specifications:

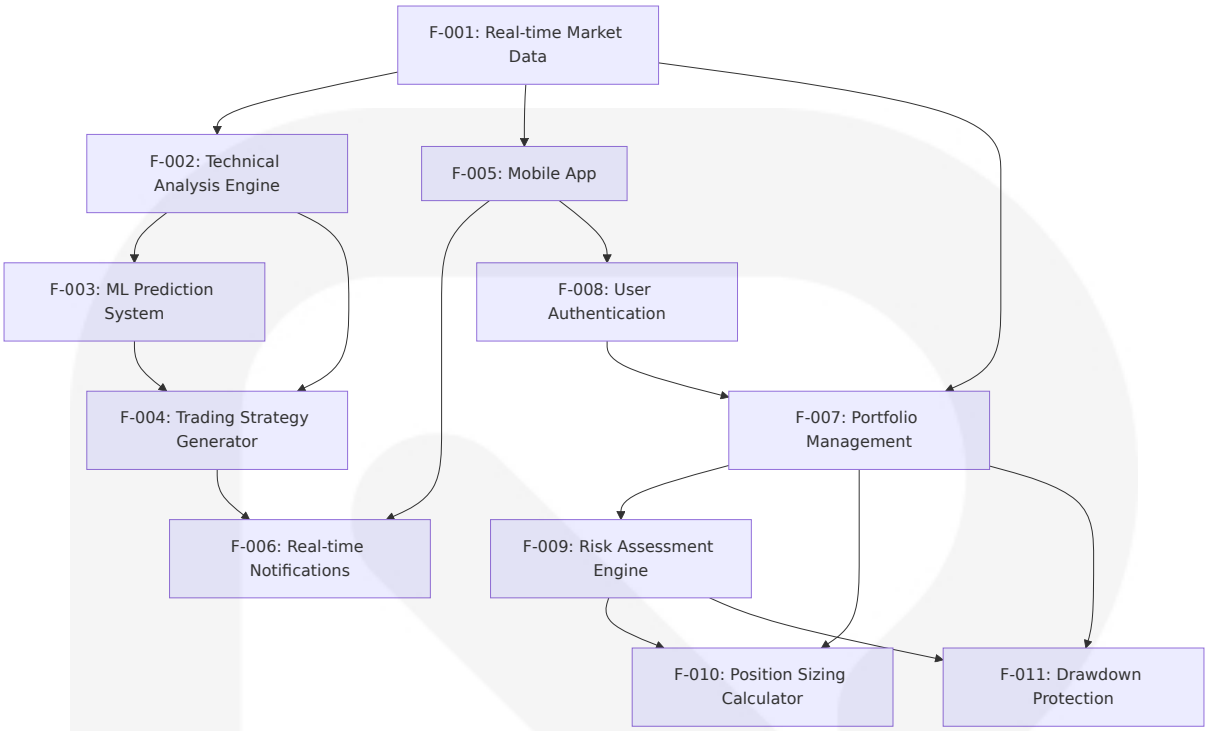
- **Input Parameters**: Market analysis results, risk parameters, user preferences
- **Output/Response**: Detailed trading plans, risk assessments, performance projections
- **Performance Criteria**: Strategy generation within 10 seconds, backtesting accuracy
- **Data Requirements**: Historical performance data, market conditions, volatility metrics

Validation Rules:

- **Business Rules**: Strategy viability checks, risk-reward ratio validation
- **Data Validation**: Backtesting data integrity, performance metric accuracy
- **Security Requirements**: Strategy protection, secure recommendation delivery
- **Compliance Requirements**: Trading recommendation regulations, disclosure requirements

2.3 FEATURE RELATIONSHIPS

2.3.1 Feature Dependencies Map



2.3.2 Integration Points

Integration Point	Connected Features	Shared Components	Common Services
Data Pipeline	F-001, F-002, F-003	Data processing engine	Real-time data streaming
Analytics Engine	F-002, F-003, F-004	Calculation libraries	Mathematical computation
Mobile Interface	F-005, F-006, F-007, F-008	UI components	Authentication service
Risk Management	F-007, F-009, F-010, F-011	Risk calculation engine	Portfolio monitoring

2.3.3 Shared Components

Data Management Layer:

- Real-time data streaming infrastructure

- Historical data storage and retrieval
- Data validation and quality assurance
- API gateway and rate limiting

Analytics Processing Layer:

- Technical indicator calculation engine
- Machine learning model inference
- Statistical analysis libraries
- Performance optimization algorithms

User Interface Layer:

- React Native 0.81 with expo-file-system API components
- Chart rendering and visualization
- User interaction handling
- Responsive design framework

Security and Authentication Layer:

- OAuth 2.0 authentication system
- API security and encryption
- User session management
- Data privacy protection

2.4 IMPLEMENTATION CONSIDERATIONS

2.4.1 Technical Constraints

Platform Limitations:

- Expo SDK 54 with React Native 0.81 targeting Android 16 with edge-to-edge enabled

- React Native 0.81 no longer provides built-in JSC support, requiring community-maintained JSC library
- TradingView doesn't have an API that gives access to data, REST API is meant for brokers

Data Access Limitations:

- Reliance on unofficial TradingView APIs with potential rate limiting
- No official TradingView API for data access, cannot use TradingView as direct data source
- Need for alternative data sources and broker API integrations

Performance Constraints:

- Mobile device processing limitations for complex ML models
- Real-time data processing requirements
- Battery and memory optimization needs

2.4.2 Performance Requirements

Response Time Targets:

- Real-time data updates: <2 seconds
- Technical analysis calculations: <1 second
- ML predictions: <5 seconds
- Strategy generation: <10 seconds

Scalability Requirements:

- Support for 1000+ concurrent users
- Handle 10,000+ API requests per minute
- Process multiple currency pairs simultaneously
- Scale ML model inference horizontally

Reliability Targets:

- 99.9% system uptime

- <0.1% data loss rate
- Automatic failover capabilities
- Disaster recovery procedures

2.4.3 Security Implications

Data Protection:

- End-to-end encryption for sensitive trading data
- Secure API key management and rotation
- User data privacy compliance (GDPR, CCPA)
- Financial data handling regulations

Authentication Security:

- Multi-factor authentication support
- Session management and timeout controls
- API rate limiting and abuse prevention
- Secure token storage and transmission

Trading Security:

- Trade execution validation and confirmation
- Risk management override controls
- Audit logging for all trading activities
- Compliance with financial regulations

2.4.4 Maintenance Requirements

Model Maintenance:

- Backtesting with at least 25,000+ trades for optimization and fine-tuning
- Automated model retraining schedules
- Performance monitoring and drift detection
- Model version control and rollback capabilities

System Maintenance:

- Regular security updates and patches
- Database optimization and cleanup
- API endpoint monitoring and maintenance
- Third-party integration updates

Operational Maintenance:

- 24/7 system monitoring and alerting
- Performance optimization and tuning
- Capacity planning and scaling
- Incident response and resolution procedures

3. TECHNOLOGY STACK

3.1 PROGRAMMING LANGUAGES

3.1.1 Frontend Mobile Development

JavaScript/TypeScript

- React Native 0.81 with React 19.1 for cross-platform mobile development
- TypeScript for enhanced type safety and developer experience
- JavaScript ES2022+ features for modern development practices

Justification:

- React Native 0.81 targeting Android 16 with edge-to-edge enabled in all Android apps
- React Native 0.81 with React 19.1 with detailed information available in release notes

- TypeScript provides compile-time type checking essential for complex trading applications
- Seamless integration with Expo SDK 54 ecosystem

3.1.2 Backend Development

Python 3.11+

- Flask 3.1.2 released August 19, 2025 for web framework
- Machine learning models applied to financial market predictions using TensorFlow, Keras, and Sci-kit Learn with past 500 days of data for forex pairs
- PyMongo supports CPython 3.9+ and PyPy3.10+ for database connectivity

Justification:

- Python's extensive ecosystem for financial analysis and machine learning
- Python's popularity and rich ecosystem of libraries, coupled with simplicity of implementing Machine Learning have made machine learning for algorithmic trading in Python a popular choice
- Strong community support for trading and financial applications
- Excellent integration with TradingView APIs and financial data sources

3.1.3 Constraints and Dependencies

Version Requirements:

- React Native 0.81 requires Node.js version 20.19.4 or higher
- React Native 0.81 requires Xcode 16.1 or higher to build iOS projects
- React Native 0.81 no longer provides built-in JSC support, requiring community-maintained JSC library

3.2 FRAMEWORKS & LIBRARIES

3.2.1 Mobile Framework

Expo SDK 54

- Expo SDK 54 includes React Native 0.81
- expo-file-system API now stable with object-oriented API for working with files and directories, support SAF URIs on Android and bundled assets on both iOS and Android
- expo-sqlite now includes drop-in implementation for localStorage web API

Core React Native Libraries:

- React Native 0.81 with New Architecture support
- Reanimated v4 introduces react-native-worklets and only supports New Architecture, with fallback to Reanimated v3 for Legacy Architecture
- React Navigation 6.x for navigation management
- React Native Paper for Material Design components

Justification:

- React Native 0.81 introduces precompiled iOS builds, cutting compile times by up to 10x in projects where React Native is primary dependency
- Expo provides comprehensive tooling for development, building, and deployment
- Strong ecosystem support for financial and trading applications

3.2.2 Backend Framework

Flask 3.1.2

- Flask is lightweight WSGI web application framework designed to make getting started quick and easy, with ability to scale up to complex applications
- Flask depends on Werkzeug WSGI toolkit, Jinja template engine, and Click CLI toolkit
- Flask 2.0+ includes typing annotations for better development experience

Supporting Libraries:

- Flask-CORS for cross-origin resource sharing
- Flask-JWT-Extended for authentication
- Flask-SocketIO for real-time WebSocket connections
- Gunicorn for production WSGI server

Justification:

- Flask has become popular among Python enthusiasts with second-most GitHub stars among Python web frameworks and voted most popular web framework in Python Developers Survey 2018-2022
- Lightweight and flexible architecture suitable for API development
- Excellent performance for real-time trading applications

3.2.3 Machine Learning Framework

Core ML Libraries:

- Scikit-learn machine learning library built upon SciPy library with various algorithms including classification, clustering, and regression
- TensorFlow end-to-end open source machine learning platform with flexible ecosystem of tools, libraries and community resources
- Numpy, Pandas, Matplotlib, scikit-learn, Keras, Tensorflow for algorithmic trading

Financial Analysis Libraries:

- TA-Lib for technical analysis indicators
- pandas, TA-Lib, scikit-learn, LightGBM, SpaCy, Gensim, TensorFlow 2, Zipline, backtrader, Alphalens, and pyfolio for algorithmic trading
- NumPy for numerical computations
- Pandas for data manipulation and analysis

Justification:

- Scikit-Learn, StatsModels and Pandas libraries with solid background in ML and statistics for quantitative trading strategies
- Proven track record in financial market prediction applications
- Strong community support and extensive documentation

3.2.4 Compatibility Requirements

React Native Compatibility:

- Metro 0.83 internal imports changed, requiring metro/private/.. instead of metro/src/.. for internal code
- react-native-edge-to-edge no longer dependency of expo package, requiring direct dependency installation

Python Compatibility:

- Flask 2.0+ officially dropped support for Python 2 and 3.5
- All libraries must support Python 3.11+ for optimal performance

3.3 OPEN SOURCE DEPENDENCIES

3.3.1 TradingView Integration

TradingView API Wrappers:

- tradingview-ta: unofficial python API wrapper to retrieve technical analysis from TradingView

- python-tradingview-ta: unofficial API wrapper for TradingView allowing fetch of technical analysis data
- tradingview-screener: Python package for custom stock screeners using TradingView's official API, retrieving data directly without web scraping

Package Versions:

- `tradingview-ta==3.3.0` - Primary technical analysis wrapper
- `tradingview-screener==3.0.0` - Market screening capabilities
- `requests>=2.31.0` - HTTP client library

3.3.2 Machine Learning Dependencies

Core ML Packages:

- `scikit-learn>=1.3.0` - Machine learning algorithms
- `tensorflow>=2.13.0` - Deep learning framework
- `keras>=2.13.0` - High-level neural networks API
- `numpy>=1.24.0` - Numerical computing
- `pandas>=2.0.0` - Data manipulation and analysis

Financial Analysis:

- `TA-Lib>=0.4.25` - Technical analysis library
- `yfinance>=0.2.18` - Yahoo Finance data
- `matplotlib>=3.7.0` - Plotting and visualization
- `seaborn>=0.12.0` - Statistical data visualization

3.3.3 React Native Dependencies

Core Mobile Packages:

- `@expo/vector-icons` - Icon library
- `expo-linear-gradient` - Gradient components
- `expo-notifications` - Push notifications

- `expo-secure-store` - Secure storage
- `react-native-chart-kit` - Chart components
- `react-native-paper` - Material Design components

Navigation and State Management:

- `@react-navigation/native>=6.1.0`
- `@react-navigation/stack>=6.3.0`
- `@reduxjs/toolkit>=1.9.0`
- `react-redux>=8.1.0`

3.3.4 Backend Dependencies

Flask Ecosystem:

- `Flask==3.1.2` - Web framework
- `Flask-CORS>=4.0.0` - Cross-origin resource sharing
- `Flask-JWT-Extended>=4.5.0` - JWT authentication
- `Flask-SocketIO>=5.3.0` - WebSocket support
- `python-socketio>=5.8.0` - Socket.IO server

Database and Caching:

- pymongo: native Python driver for MongoDB, offering both synchronous and asynchronous APIs
- `redis>=4.6.0` - In-memory data structure store
- `celery>=5.3.0` - Distributed task queue

3.4 THIRD-PARTY SERVICES

3.4.1 Market Data APIs

TradingView Integration:

- TradingView REST API for brokers to connect backend systems to TradingView frontend
- TA_Handler system providing technical analysis with recommendation outputs like "BUY", "NEUTRAL", "SELL"
- WebSocket connections for real-time market data
- Historical data access for backtesting

Alternative Data Sources:

- Alpha Vantage API for forex data
- IEX Cloud for market data
- Quandl for financial and economic data
- Yahoo Finance API as backup data source

3.4.2 Cloud Infrastructure Services

AWS Services:

- Amazon EC2 for application hosting
- Amazon RDS for managed database services
- Amazon ElastiCache for Redis caching
- Amazon S3 for file storage and backups
- Amazon CloudWatch for monitoring and logging

Authentication and Security:

- Auth0 for user authentication and authorization
- AWS Cognito as alternative authentication service
- SSL/TLS certificates for secure communications
- API rate limiting and security services

3.4.3 Mobile Services

Push Notifications:

- Firebase Cloud Messaging (FCM) for Android

- Apple Push Notification Service (APNs) for iOS
- Expo Push Notification service for unified delivery

Analytics and Monitoring:

- Firebase Analytics for user behavior tracking
- Sentry for error monitoring and crash reporting
- New Relic for application performance monitoring

3.4.4 Development and Deployment Services

CI/CD Pipeline:

- GitHub Actions for continuous integration
- EAS Build for Expo application builds
- EAS Submit for app store submissions
- Docker Hub for container registry

Monitoring and Logging:

- Datadog for infrastructure monitoring
- LogRocket for frontend monitoring
- Papertrail for log aggregation and analysis

3.5 DATABASES & STORAGE

3.5.1 Primary Database

MongoDB 7.0+

- PyMongo supports MongoDB 4.0, 4.2, 4.4, 5.0, 6.0, 7.0, and 8.0
- BSON format implementation, native Python driver with synchronous and asynchronous APIs, gridfs implementation
- Document-based storage for flexible trading data schemas

- Built-in sharding and replication for scalability

Collections Structure:

- `users` - User accounts and preferences
- `trading_strategies` - Generated trading plans and recommendations
- `market_data` - Historical and real-time market information
- `ml_models` - Machine learning model metadata and versions
- `trading_history` - User trading performance and analytics

3.5.2 Caching Layer

Redis 7.0+

- In-memory caching for real-time market data
- Session storage for user authentication
- Rate limiting and API throttling
- WebSocket connection management
- Temporary storage for ML model predictions

Cache Strategies:

- Real-time price data: 1-5 second TTL
- Technical indicators: 1-minute TTL
- User sessions: 24-hour TTL
- ML predictions: 15-minute TTL

3.5.3 File Storage

Amazon S3

- Historical market data archives
- ML model artifacts and checkpoints
- User-generated content and documents
- Application logs and backups
- Static assets for mobile application

Local Storage (Mobile):

- expo-file-system API with object-oriented API for working with files and directories
- expo-sqlite with localStorage web API implementation
- Secure storage for authentication tokens
- Offline data caching for critical trading information

3.5.4 Data Persistence Strategies

Real-time Data:

- WebSocket connections with automatic reconnection
- Circuit breaker pattern for API failures
- Data validation and sanitization pipelines
- Automatic data backup and recovery procedures

Historical Data:

- Batch processing for large datasets
- Data compression and archival strategies
- Incremental backup procedures
- Data retention policies for compliance

3.6 DEVELOPMENT & DEPLOYMENT

3.6.1 Development Tools

Mobile Development:

- Expo SDK 54 with React Native 0.81
- Expo CLI for project management and builds
- React Native Debugger for debugging
- Flipper for advanced debugging and profiling

Backend Development:

- Visual Studio Code with Python extensions
- PyCharm Professional for advanced Python development
- Postman for API testing and documentation
- MongoDB Compass for database management

3.6.2 Build System

Mobile Builds:

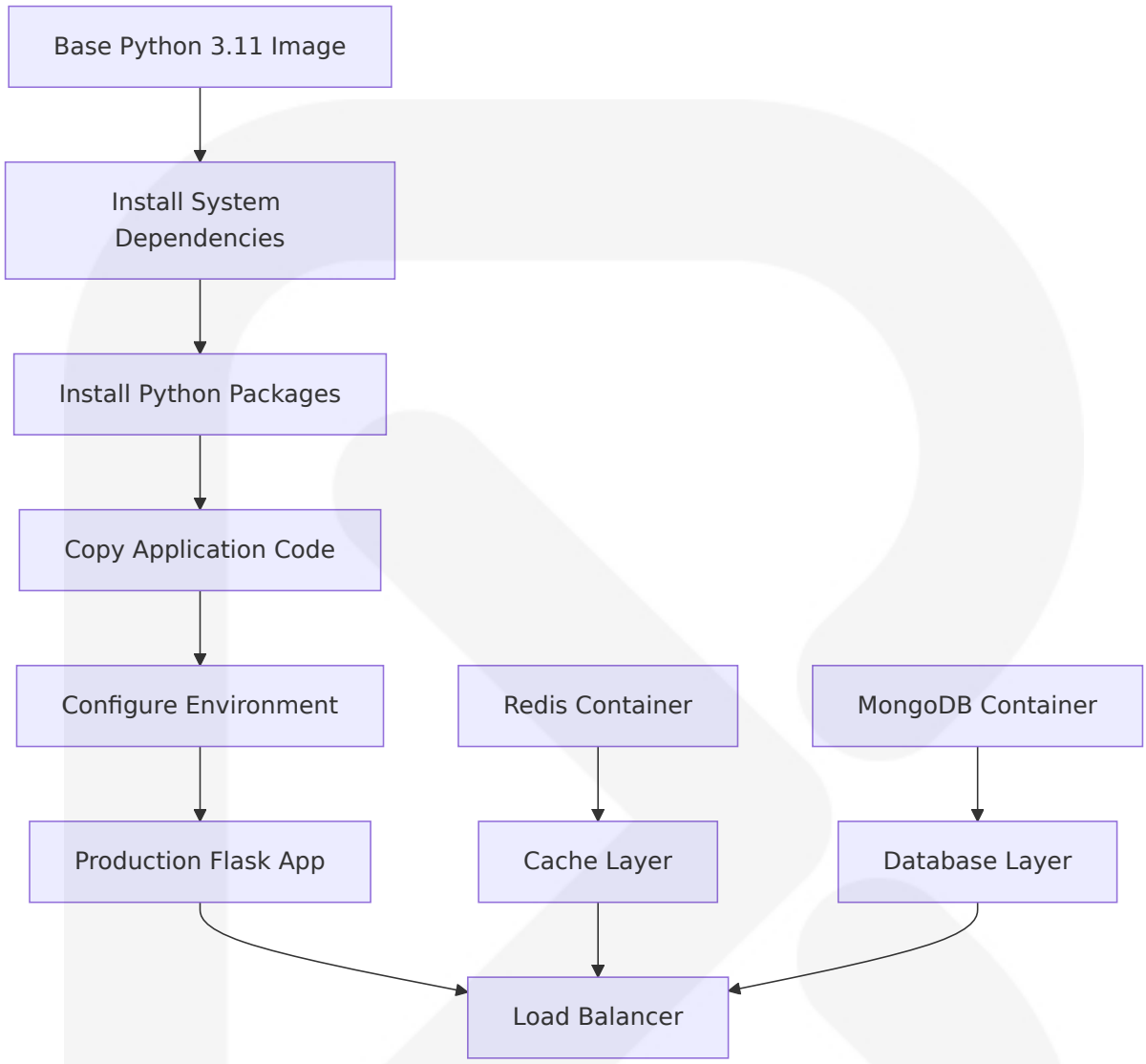
- React Native 0.81 precompiled iOS builds cutting compile times by up to 10x
- EAS Build for cloud-based mobile app compilation
- Fastlane for iOS and Android deployment automation
- Code signing and certificate management

Backend Builds:

- Docker containerization for consistent environments
- Multi-stage builds for optimized production images
- Automated testing and code quality checks
- Dependency vulnerability scanning

3.6.3 Containerization

Docker Configuration:



Container Strategy:

- Multi-container architecture with Docker Compose
- Separate containers for Flask app, Redis, and MongoDB
- Health checks and automatic restart policies
- Resource limits and monitoring

3.6.4 CI/CD Requirements

Continuous Integration:

- Automated testing on pull requests

- Code quality checks with SonarQube
- Security vulnerability scanning
- Performance regression testing

Continuous Deployment:

- Staging environment for pre-production testing
- Blue-green deployment strategy
- Automated rollback capabilities
- Database migration management

Pipeline Stages:

1. Code commit and pull request creation
2. Automated testing and quality checks
3. Security and vulnerability scanning
4. Staging deployment and integration testing
5. Production deployment with monitoring
6. Post-deployment verification and alerts

Deployment Monitoring:

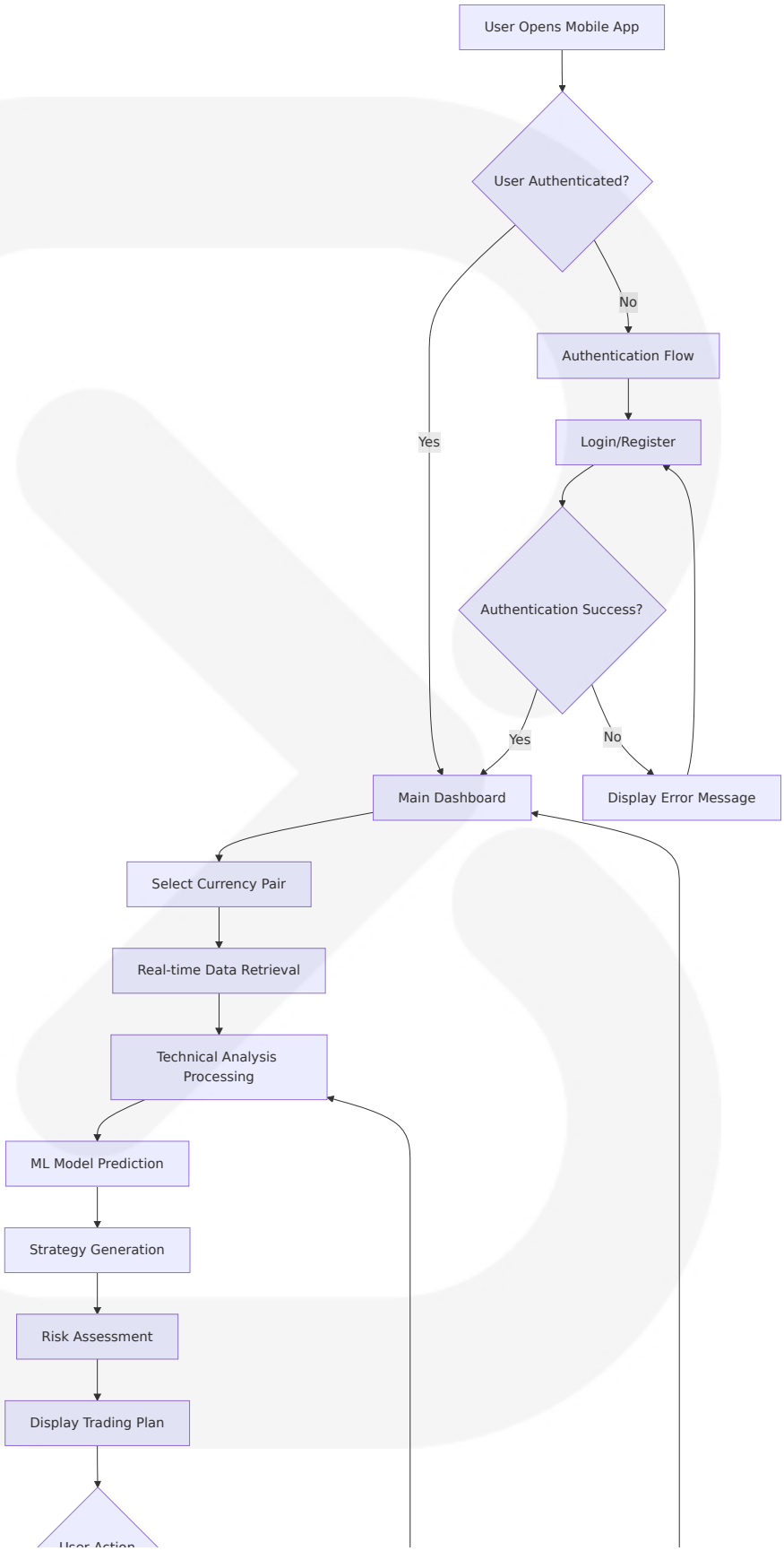
- Application performance monitoring
- Error tracking and alerting
- Resource utilization monitoring
- User experience analytics

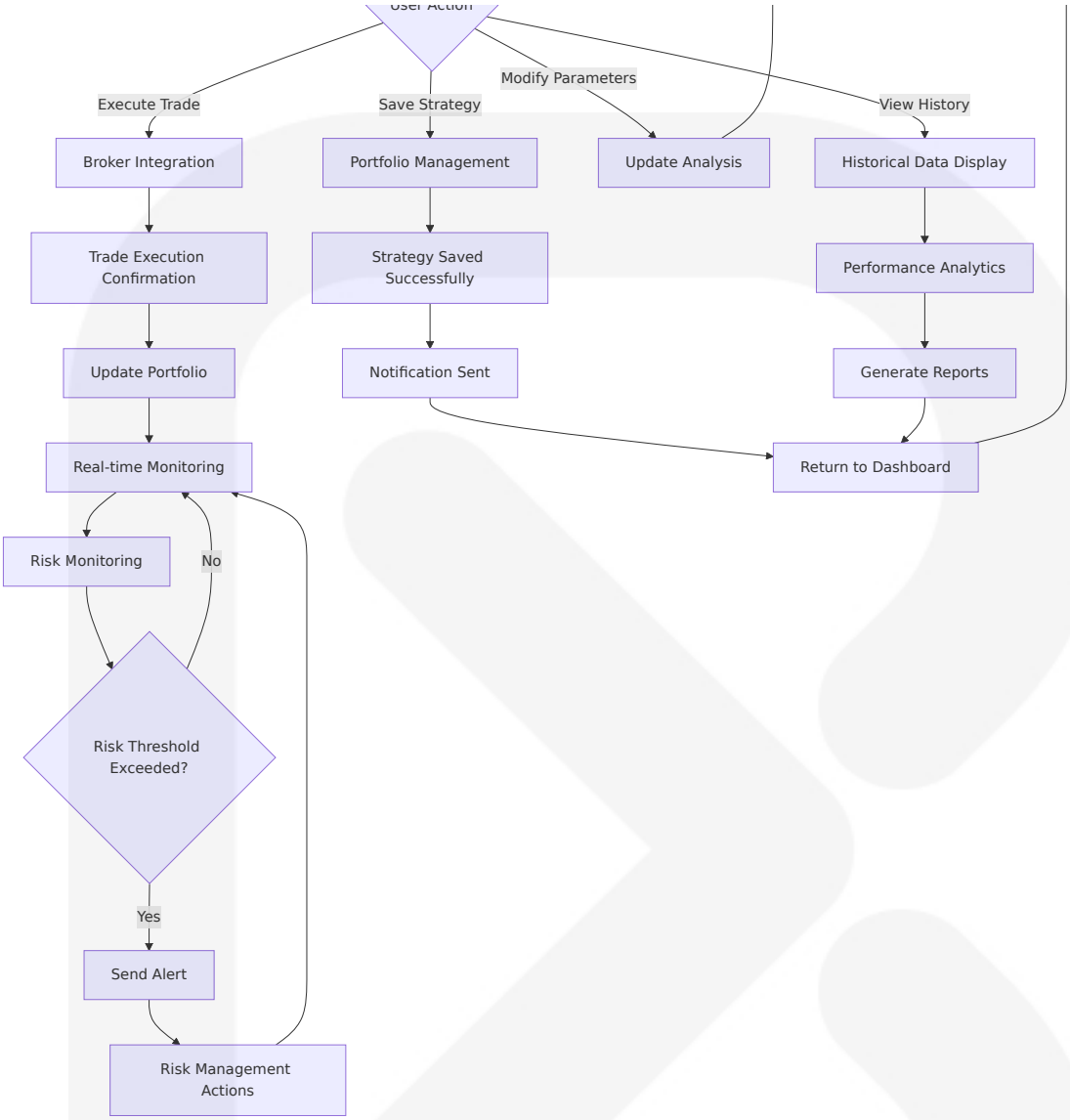
4. PROCESS FLOWCHART

4.1 SYSTEM WORKFLOWS

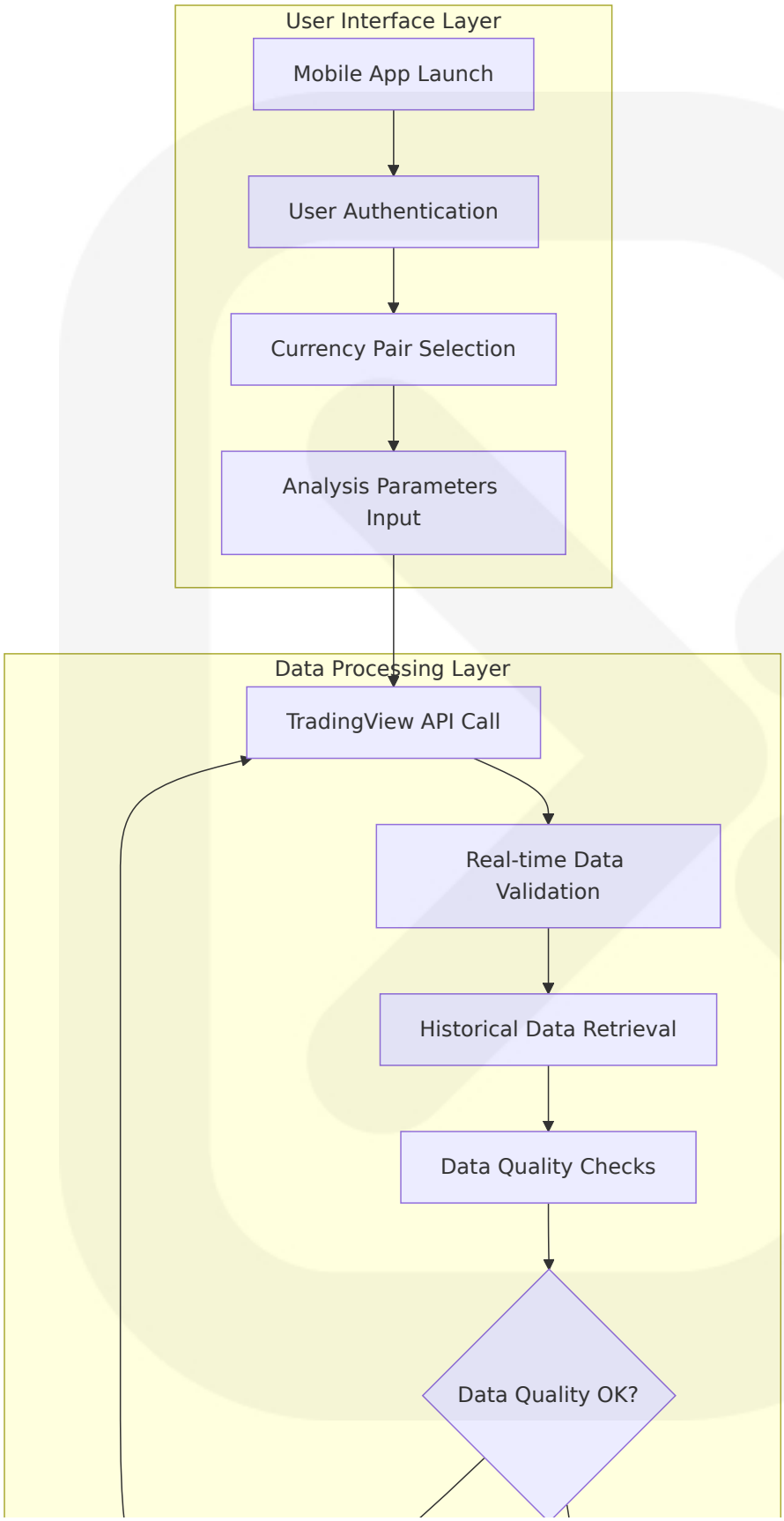
4.1.1 Core Business Processes

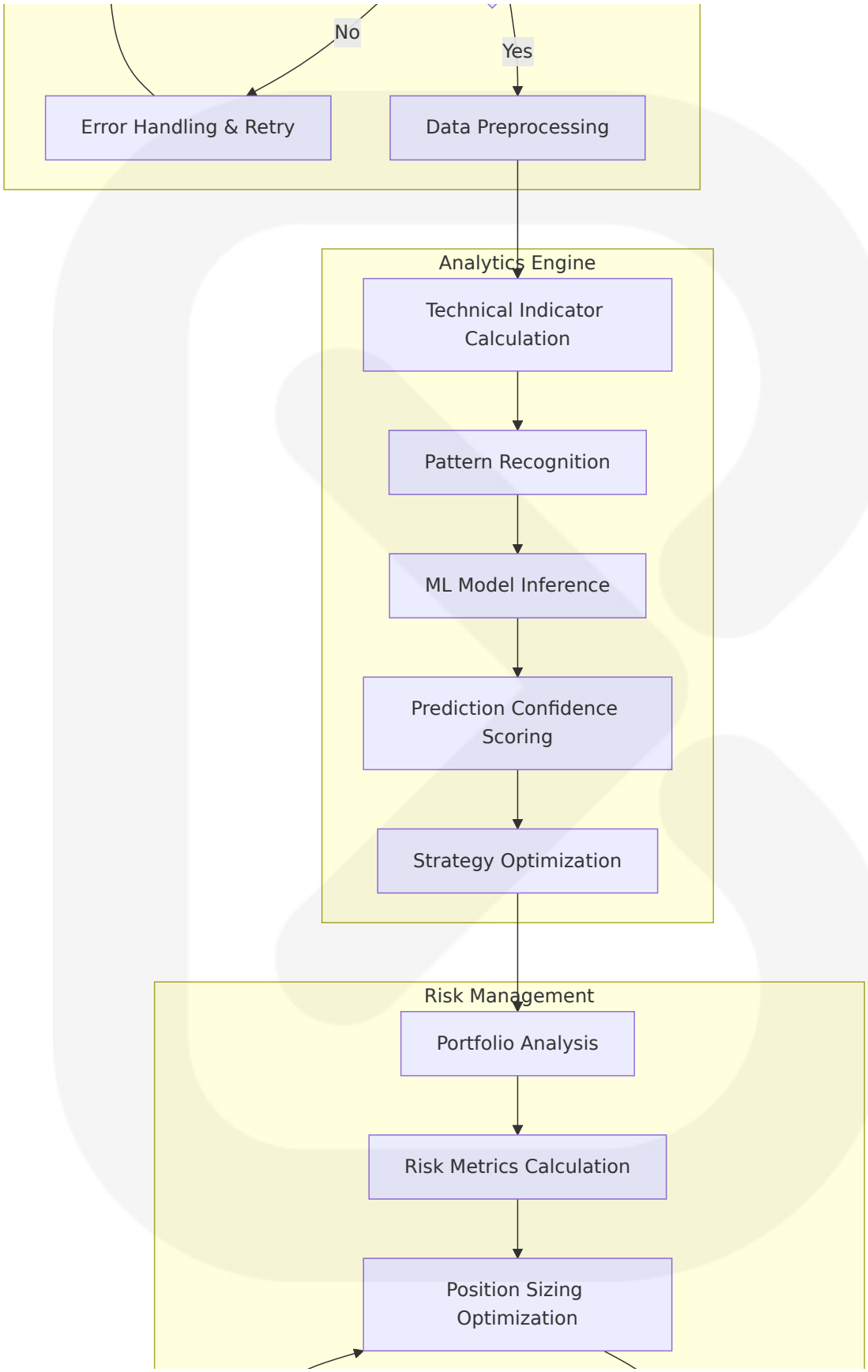
High-Level System Workflow

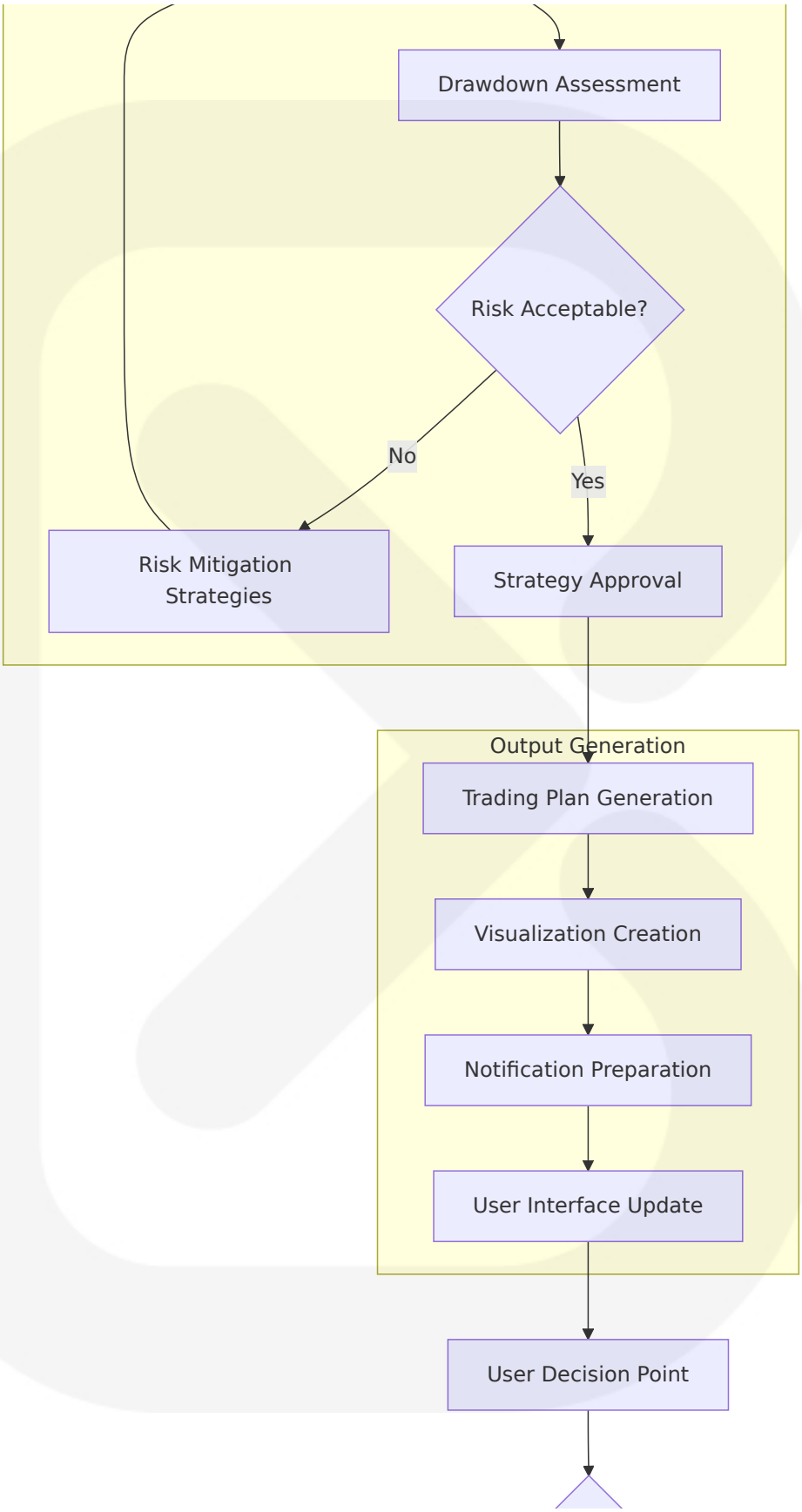

User Opens Mobile App
User Authenticated?
No
Yes
Authentication Flow
Login/Register
Authentication Success?
Yes
No
Main Dashboard
Display Error Message
Select Currency Pair
Real-time Data Retrieval
Technical Analysis Processing
ML Model Prediction
Strategy Generation
Risk Assessment
Display Trading Plan

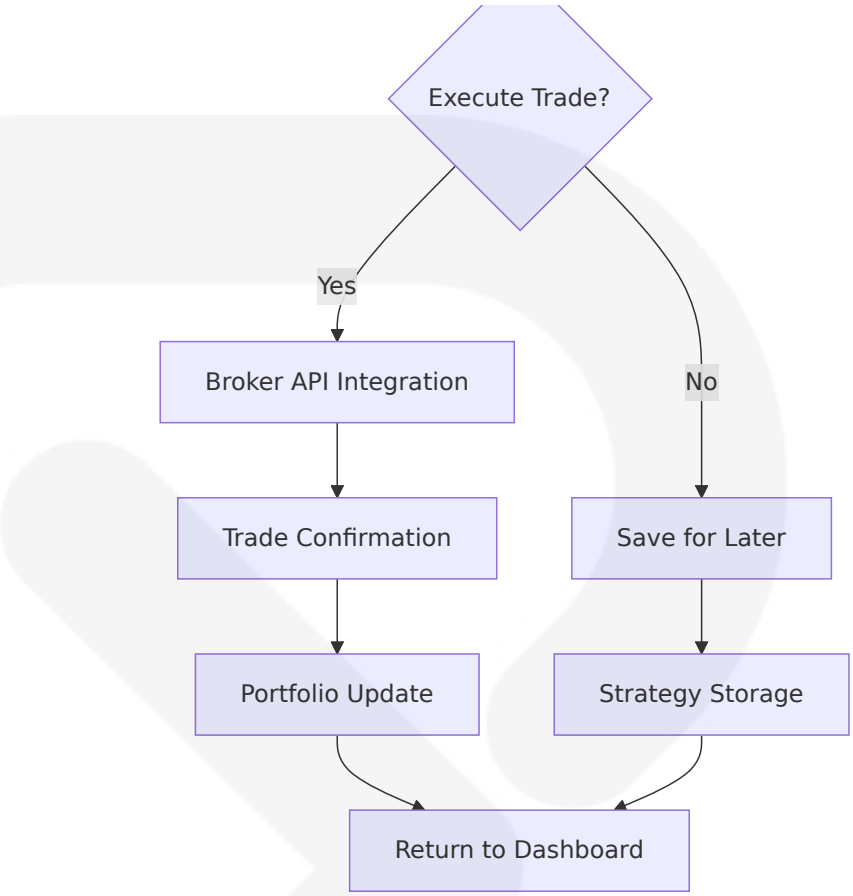


End-to-End User Journey: Market Analysis to Trading Decision

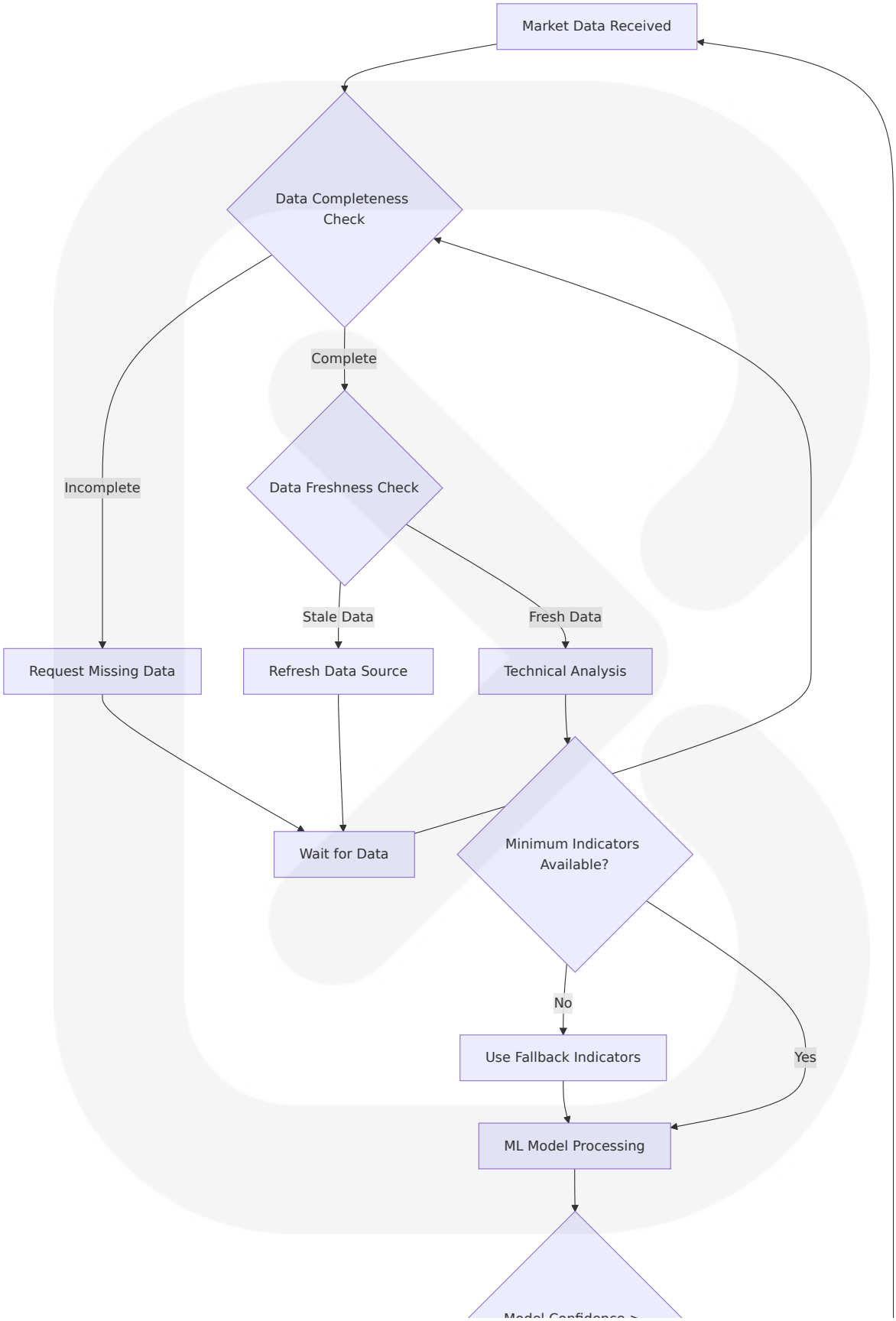

User Interface Layer
Mobile App Launch
User Authentication
Currency Pair Selection
Analysis Parameters Input
Data Processing Layer
TradingView API Call
Real-time Data Validation
Historical Data Retrieval
Data Quality Checks
Data Quality OK?

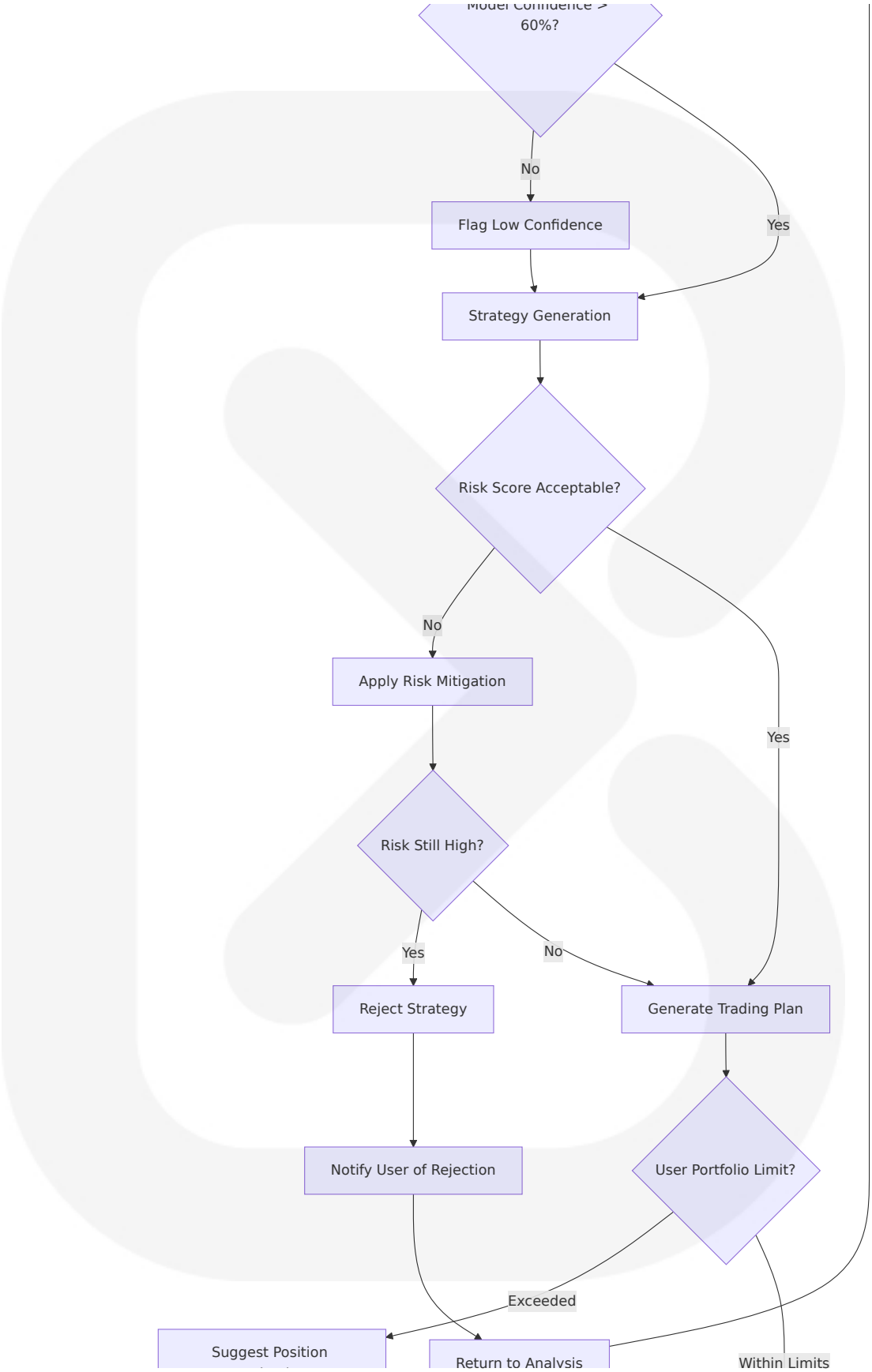

No
Yes
Error Handling & Retry
Data Preprocessing
Analytics Engine
Technical Indicator Calculation
Pattern Recognition
ML Model Inference
Prediction Confidence Scoring
Strategy Optimization
Risk Management
Portfolio Analysis
Risk Metrics Calculation
Position Sizing Optimization

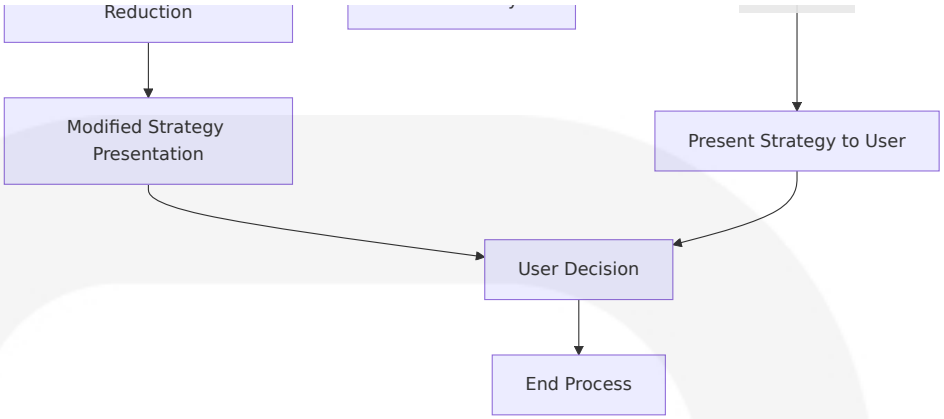

Drawdown Assessment
Risk Acceptable?
No
Yes
Risk Mitigation Strategies
Strategy Approval
Output Generation
Trading Plan Generation
Visualization Creation
Notification Preparation
User Interface Update
User Decision Point



Decision Points and Business Rules

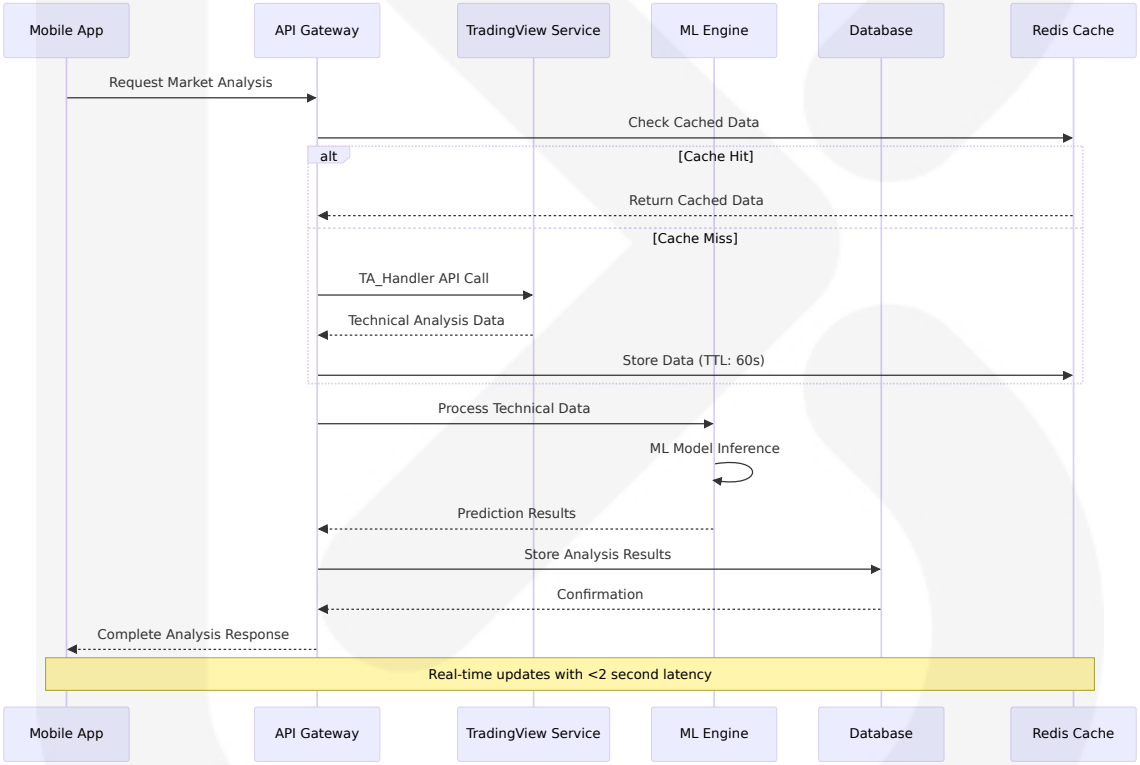

Market Data Received
Data Completeness Check
Incomplete
Complete
Data Freshness Check
Stale Data
Fresh Data
Request Missing Data
Refresh Data Source
Technical Analysis
Wait for Data
Minimum Indicators Available?
No
Yes
Use Fallback Indicators
ML Model Processing


Model Confidence > 60%?
No
Yes
Flag Low Confidence
Strategy Generation
Risk Score Acceptable?
No
Yes
Apply Risk Mitigation
Risk Still High?
Yes
No
Reject Strategy
Generate Trading Plan
Notify User of Rejection
User Portfolio Limit?
Exceeded
Within Limits
Suggest Position
Return to Analysis

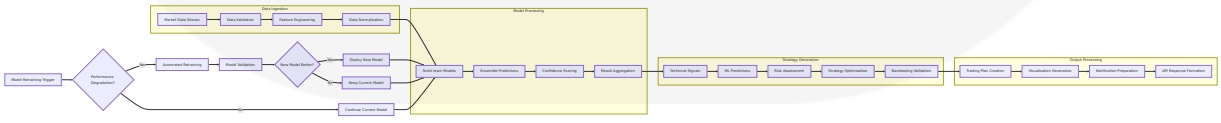


4.1.2 Integration Workflows

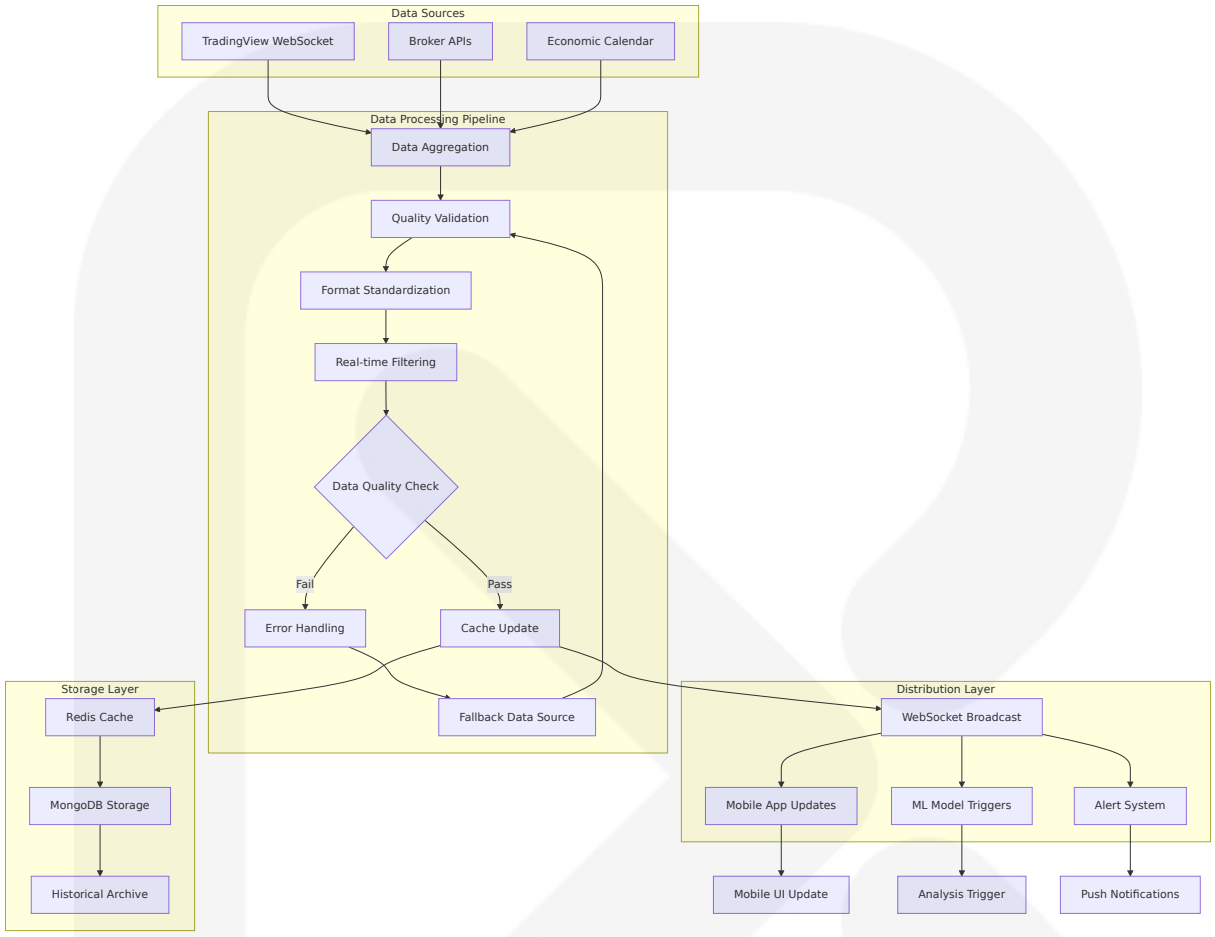
TradingView API Integration Flow



Machine Learning Pipeline Workflow



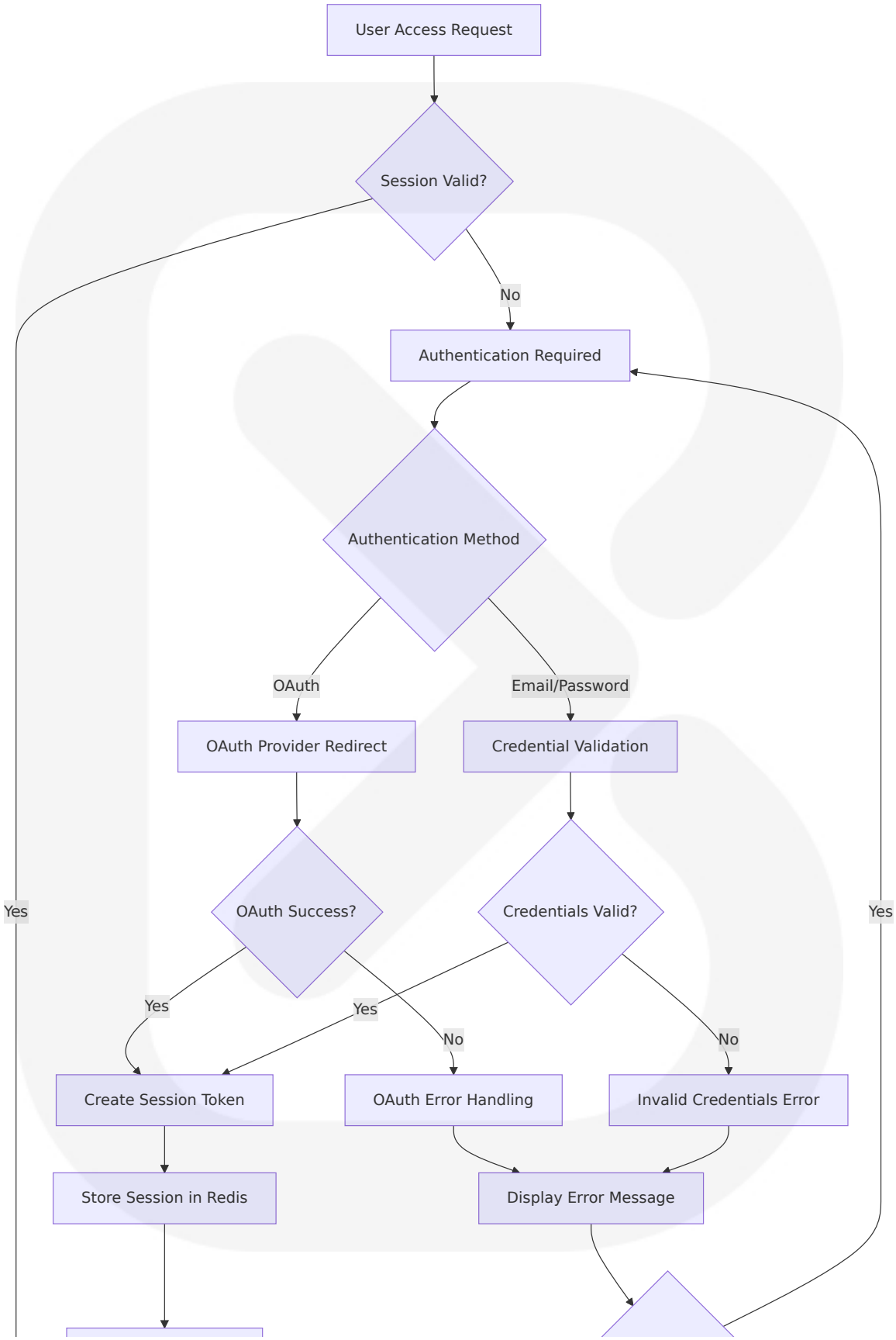
Real-time Data Processing Flow

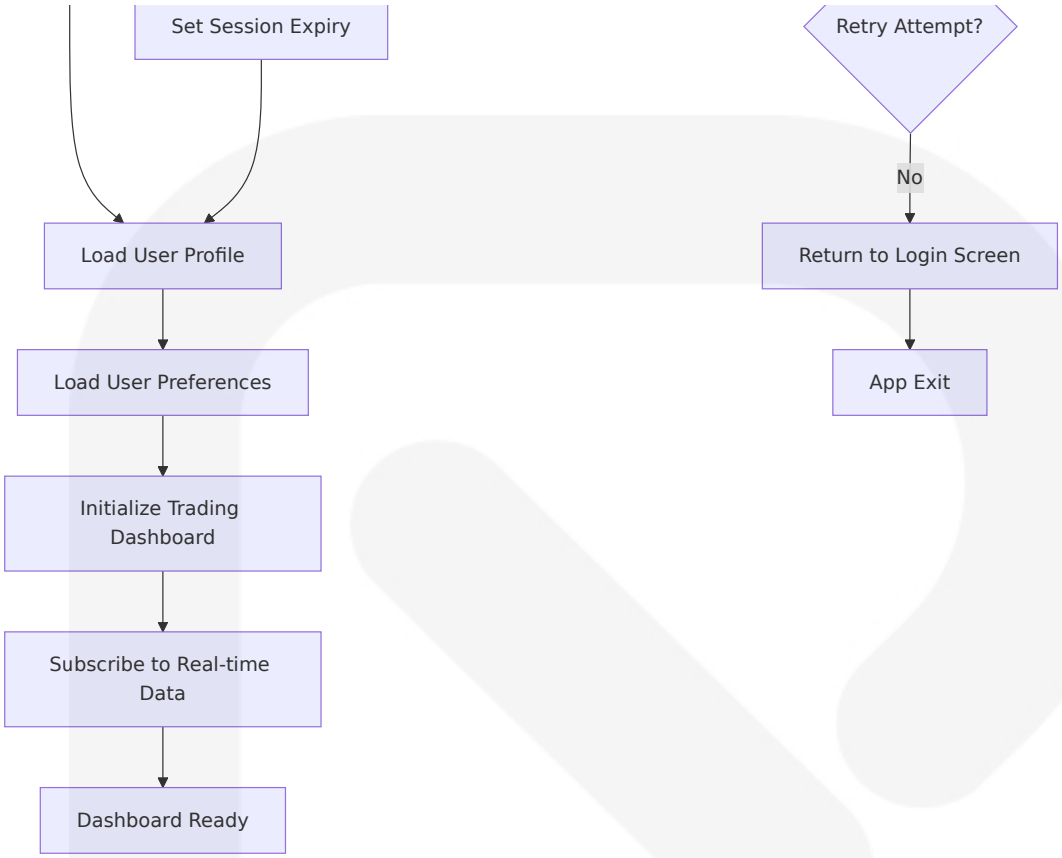


4.2 FLOWCHART REQUIREMENTS

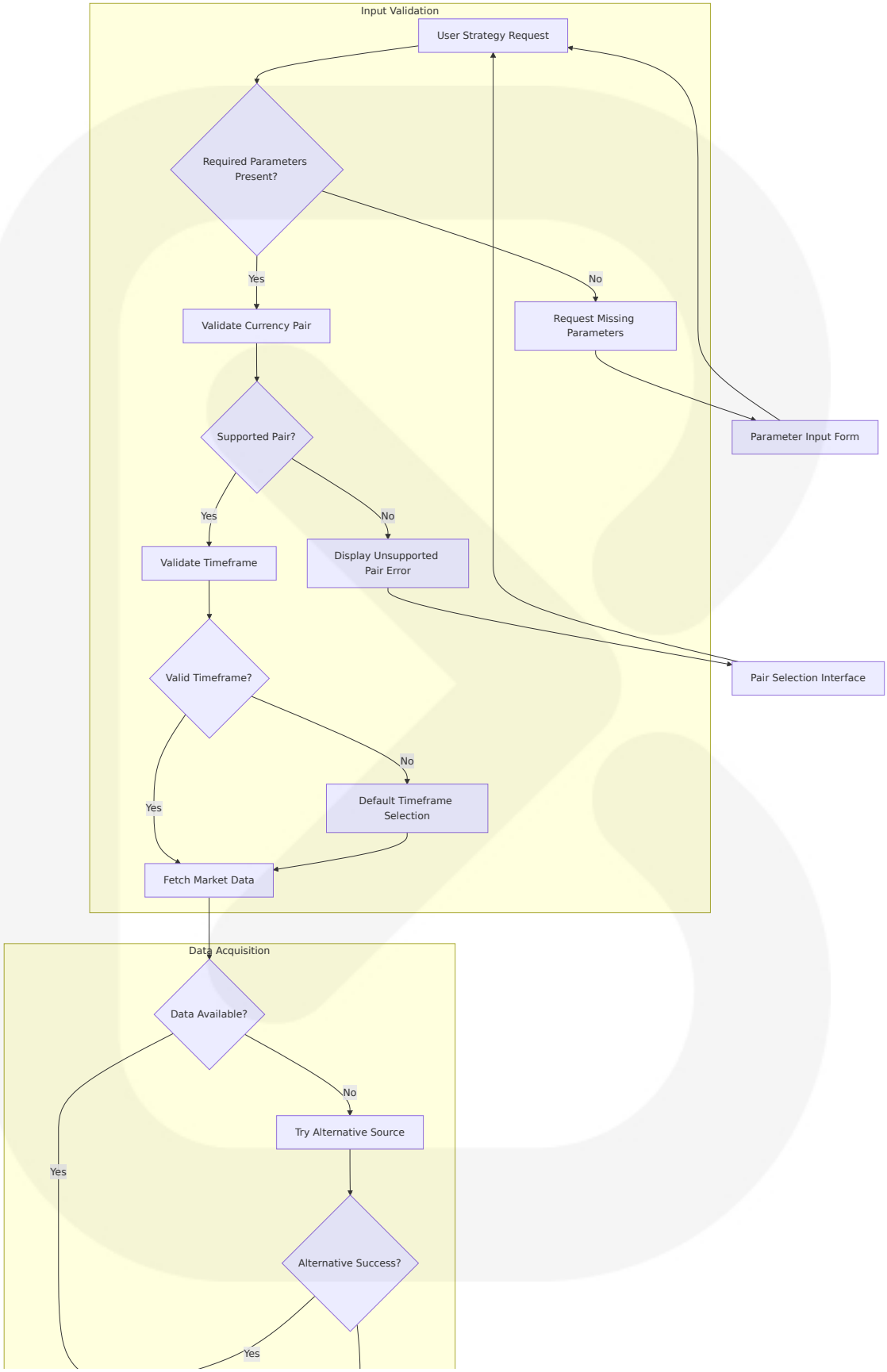
4.2.1 Process Steps and Decision Points

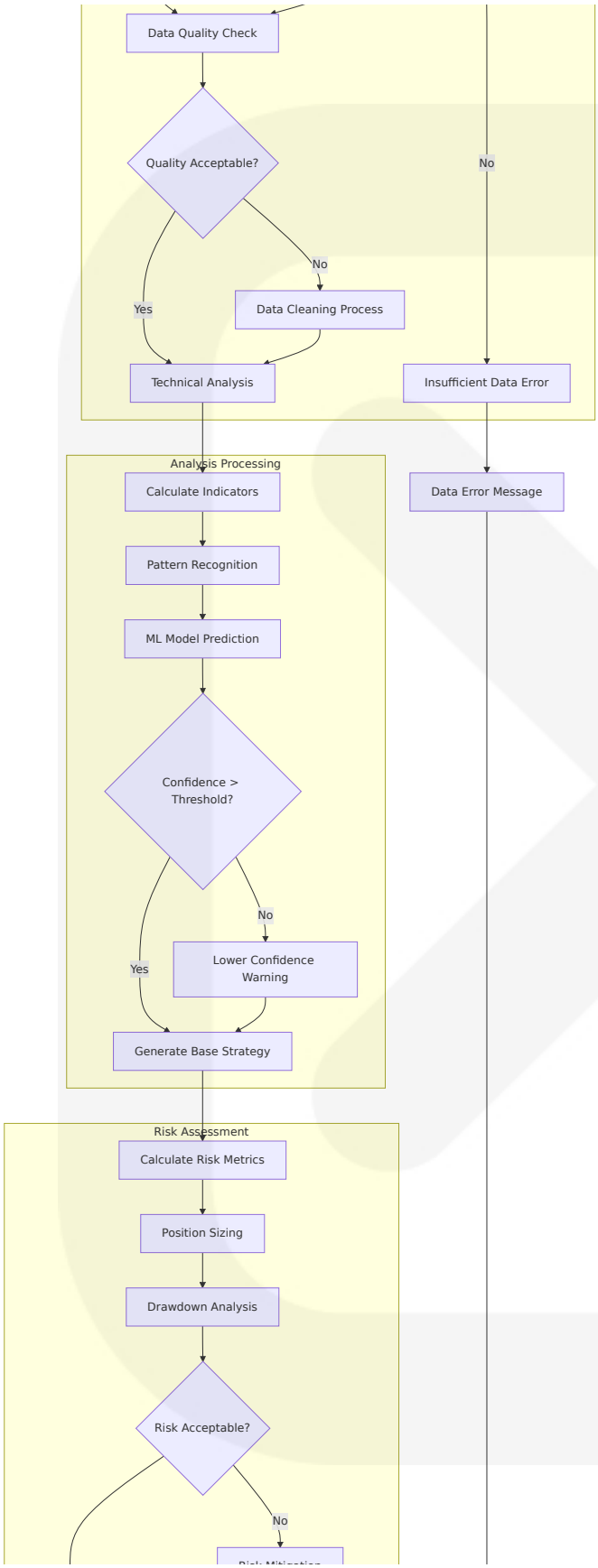
User Authentication and Session Management

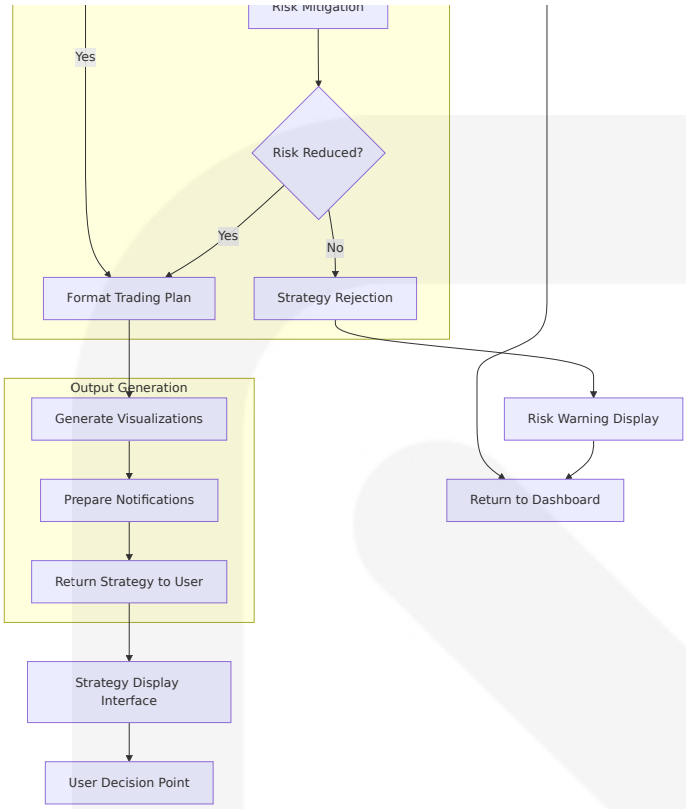

User Access Request
Session Valid?
Yes
No
Authentication Required
Authentication Method
OAuth
Email/Password
OAuth Provider Redirect
Credential Validation
OAuth Success?
Credentials Valid?
Yes
Yes
Yes
Yes
No
No
Create Session Token
OAuth Error Handling
Invalid Credentials Error
Store Session in Redis
Display Error Message



Trading Strategy Generation Process

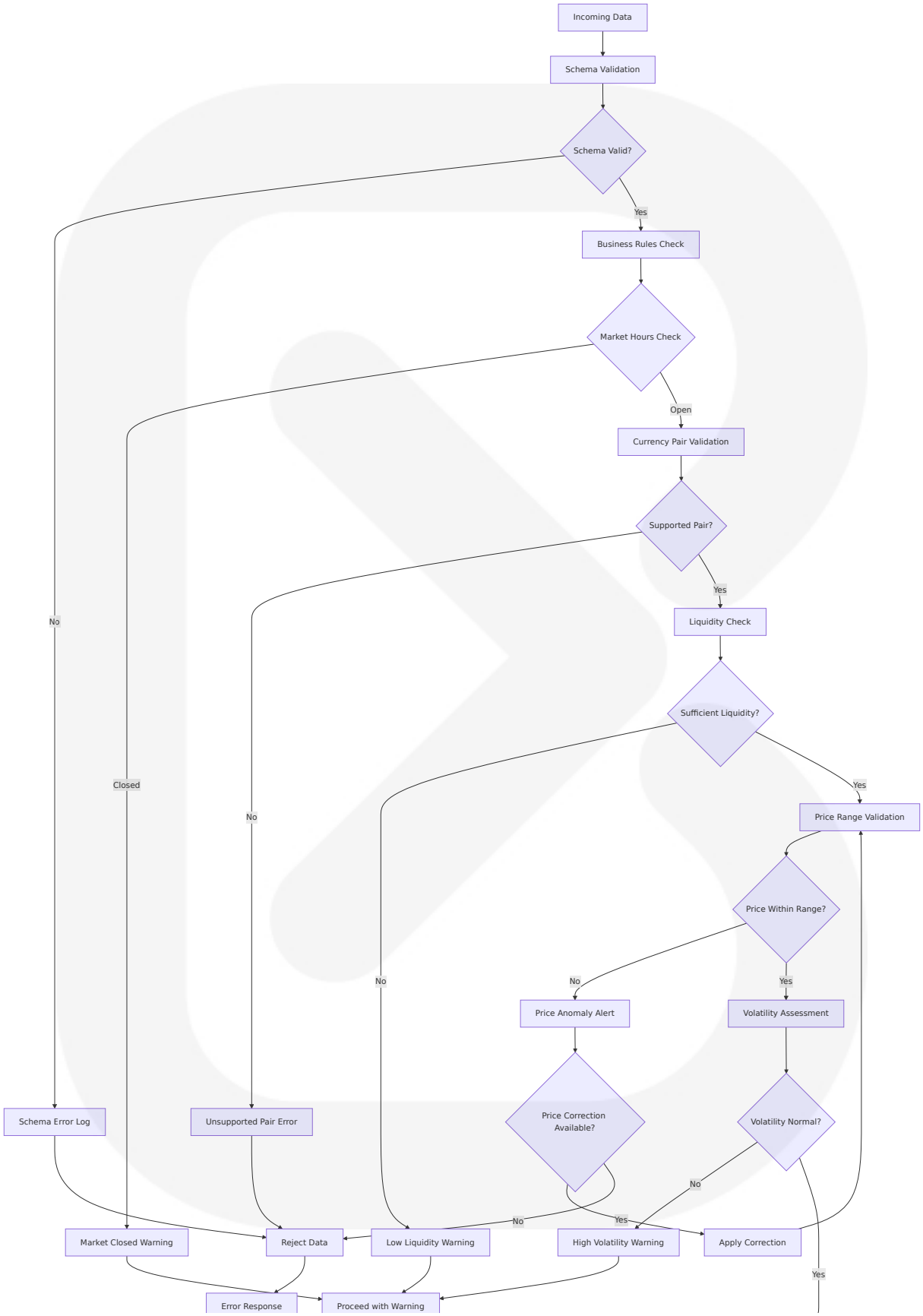

Input Validation
User Strategy Request
Required Parameters Present?
Yes
No
Validate Currency Pair
Request Missing Parameters
Parameter Input Form
Supported Pair?
Yes
No
Validate Timeframe
Display Unsupported Pair Error
Pair Selection Interface
Valid Timeframe?
No
Yes
Default Timeframe Selection
Fetch Market Data
Data Acquisition
Data Available?
No
Try Alternative Source
Yes
Alternative Success?
Yes

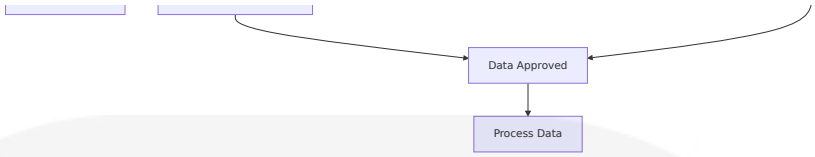

Data Quality Check
Quality Acceptable?
No
Yes
Data Cleaning Process
Technical Analysis
No
Insufficient Data Error
Analysis Processing
Calculate Indicators
Data Error Message
Pattern Recognition
ML Model Prediction
Confidence > Threshold?
No
Yes
Lower Confidence Warning
Generate Base Strategy
Risk Assessment
Calculate Risk Metrics
Position Sizing
Drawdown Analysis
Risk Acceptable?
No



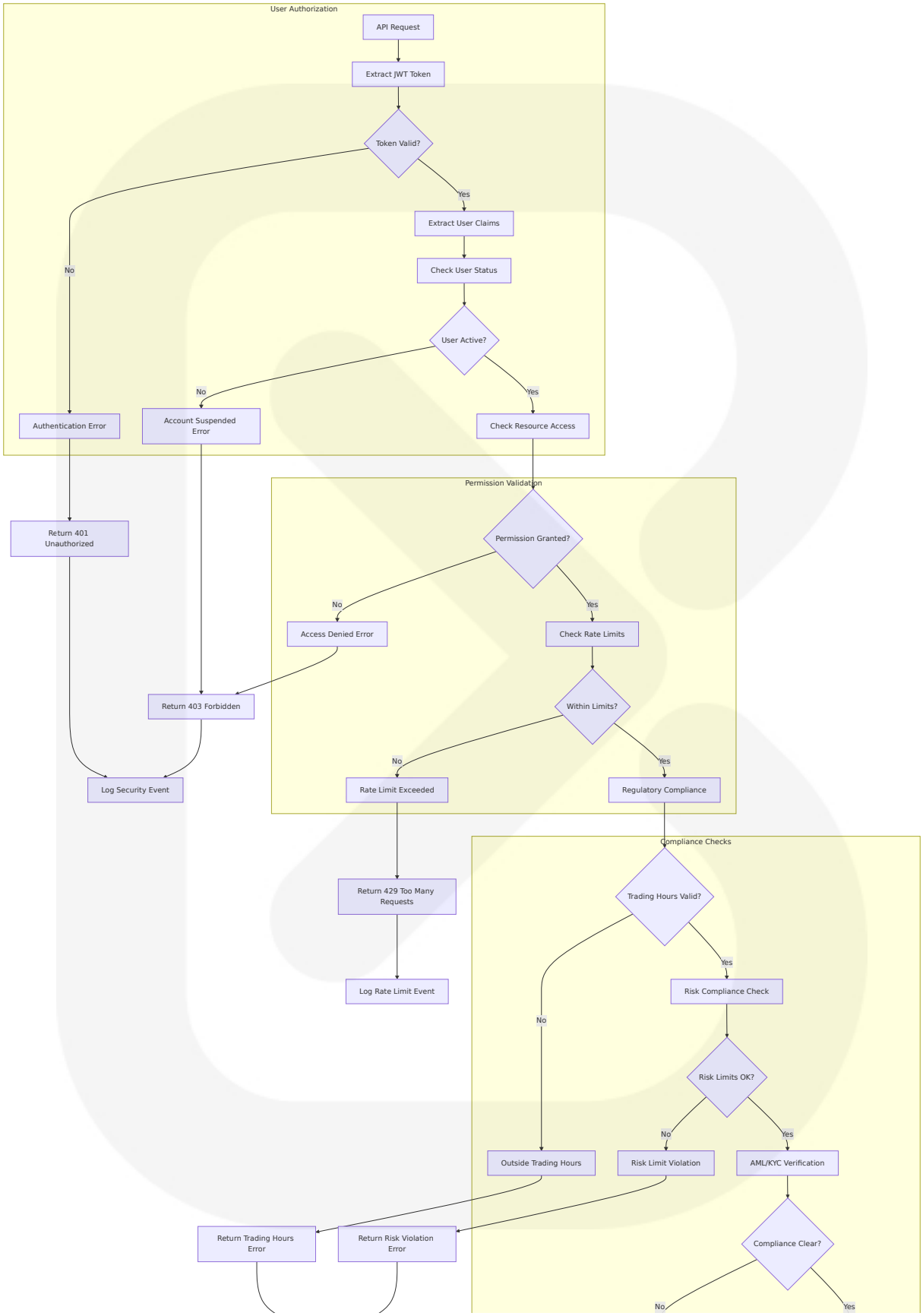
4.2.2 Validation Rules and Authorization

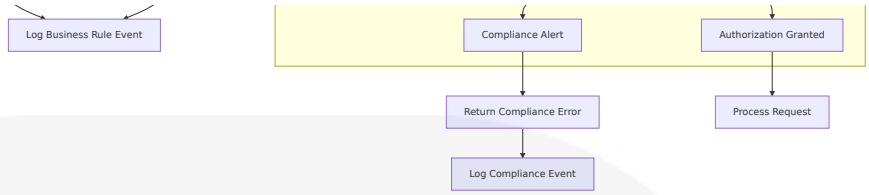
Data Validation and Business Rules Engine


Incoming Data
Schema Validation
Schema Valid?
Yes
No
Business Rules Check
Market Hours Check
Open
Closed
Currency Pair Validation
Supported Pair?
Yes
No
Liquidity Check
Sufficient Liquidity?
Yes
No
Price Range Validation
Price Within Range?
No
Yes
Price Anomaly Alert
Volatility Assessment
Price Correction Available?
Volatility Normal?
No
Yes
No
Yes
Schema Error Log
Unsupported Pair Error
Market Closed Warning
Reject Data
Low Liquidity Warning
High Volatility Warning
Apply Correction
Error Response
Proceed with Warning



Authorization and Compliance Checkpoints

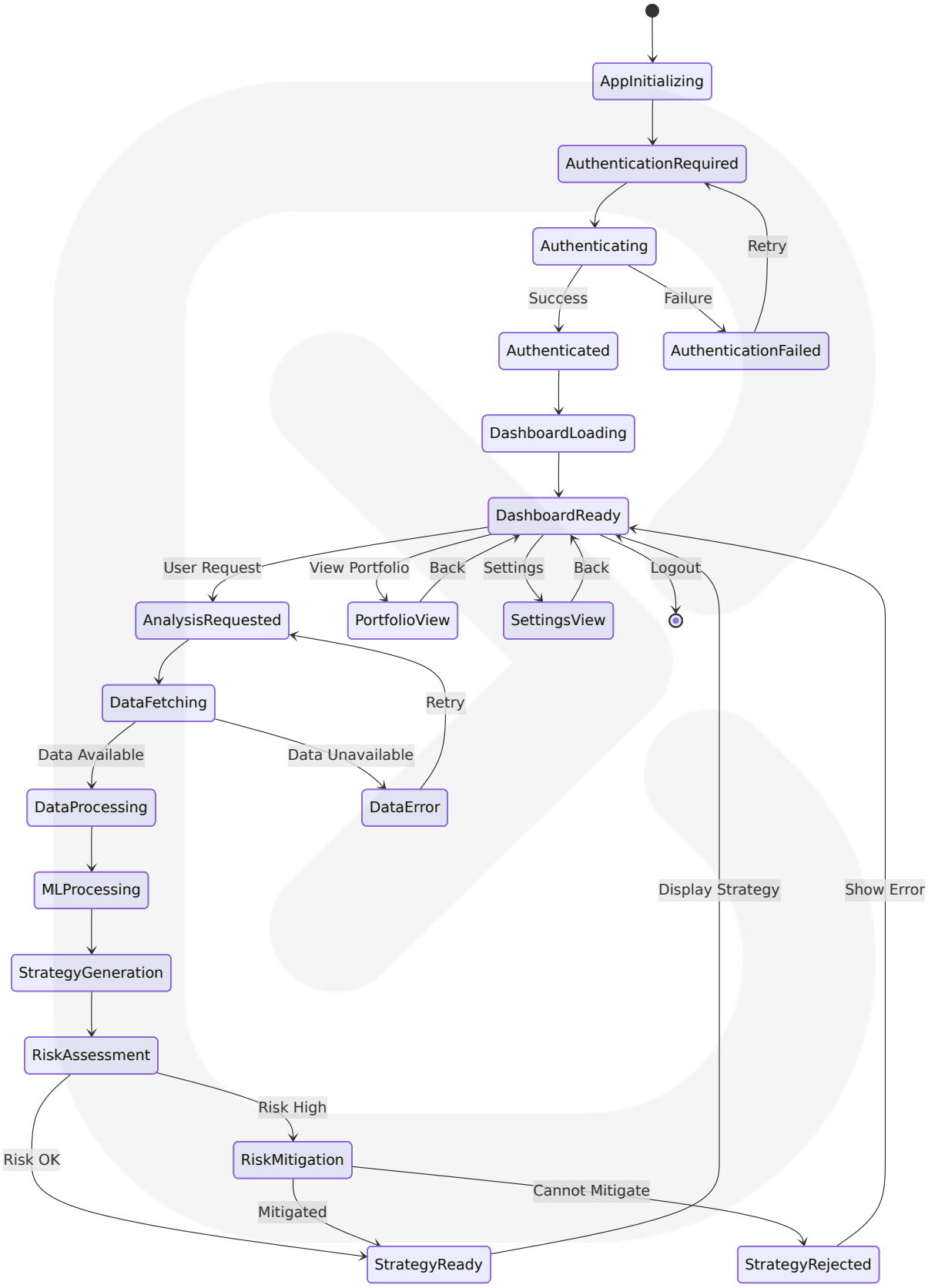

User Authorization
API Request
Extract JWT Token
Token Valid?
Yes
No
Extract User Claims
Check User Status
User Active?
No
Yes
Authentication Error
Account Suspended Error
Check Resource Access
Permission Validation
Permission Granted?
No
Yes
Access Denied Error
Check Rate Limits
Return 401 Unauthorized
Return 403 Forbidden
Within Limits?
No
Yes
Log Security Event
Rate Limit Exceeded
Regulatory Compliance
Return 429 Too Many Requests
Compliance Checks
Trading Hours Valid?
Yes
No
Log Rate Limit Event
Risk Compliance Check
Risk Limits OK?
No
Yes
Outside Trading Hours
Risk Limit Violation
AML/KYC Verification
Return Trading Hours Error
Return Risk Violation Error
Compliance Clear?
No
Yes



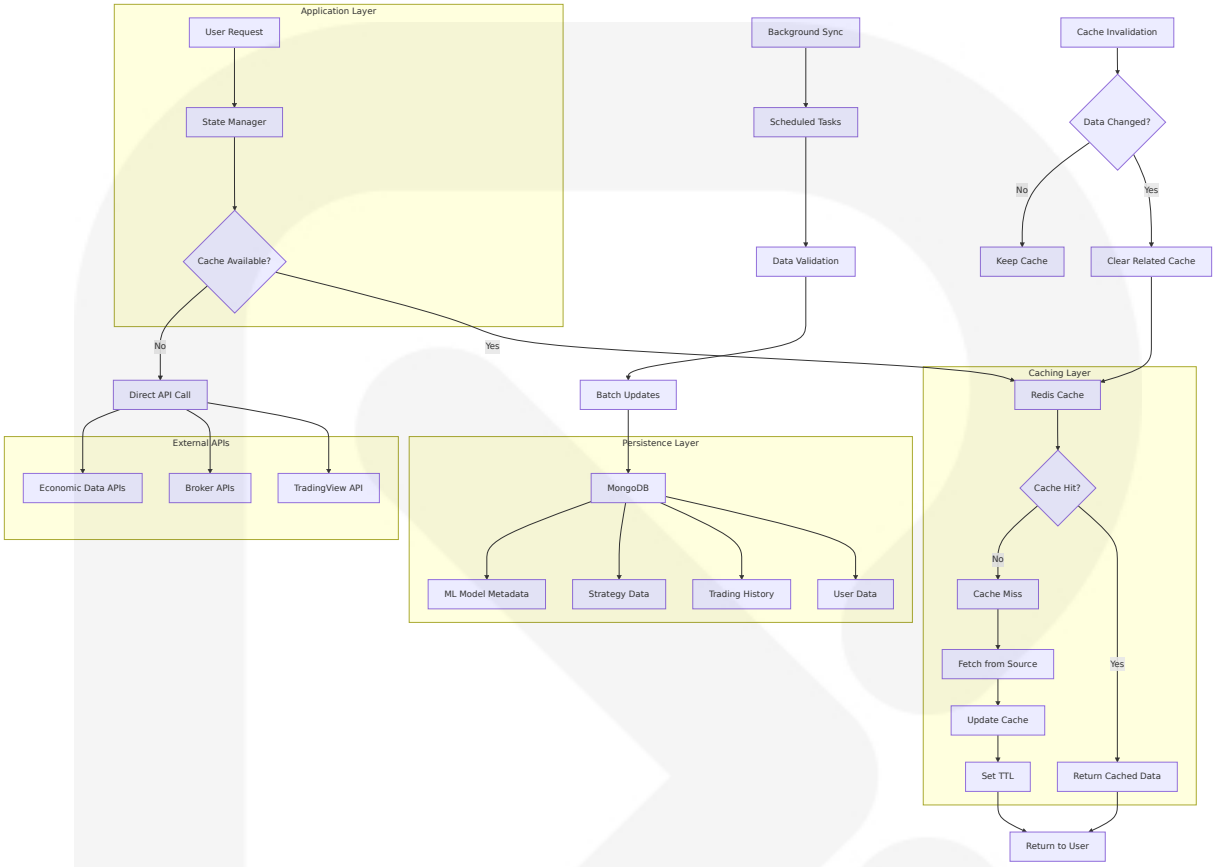
4.3 TECHNICAL IMPLEMENTATION

4.3.1 State Management

Application State Transitions

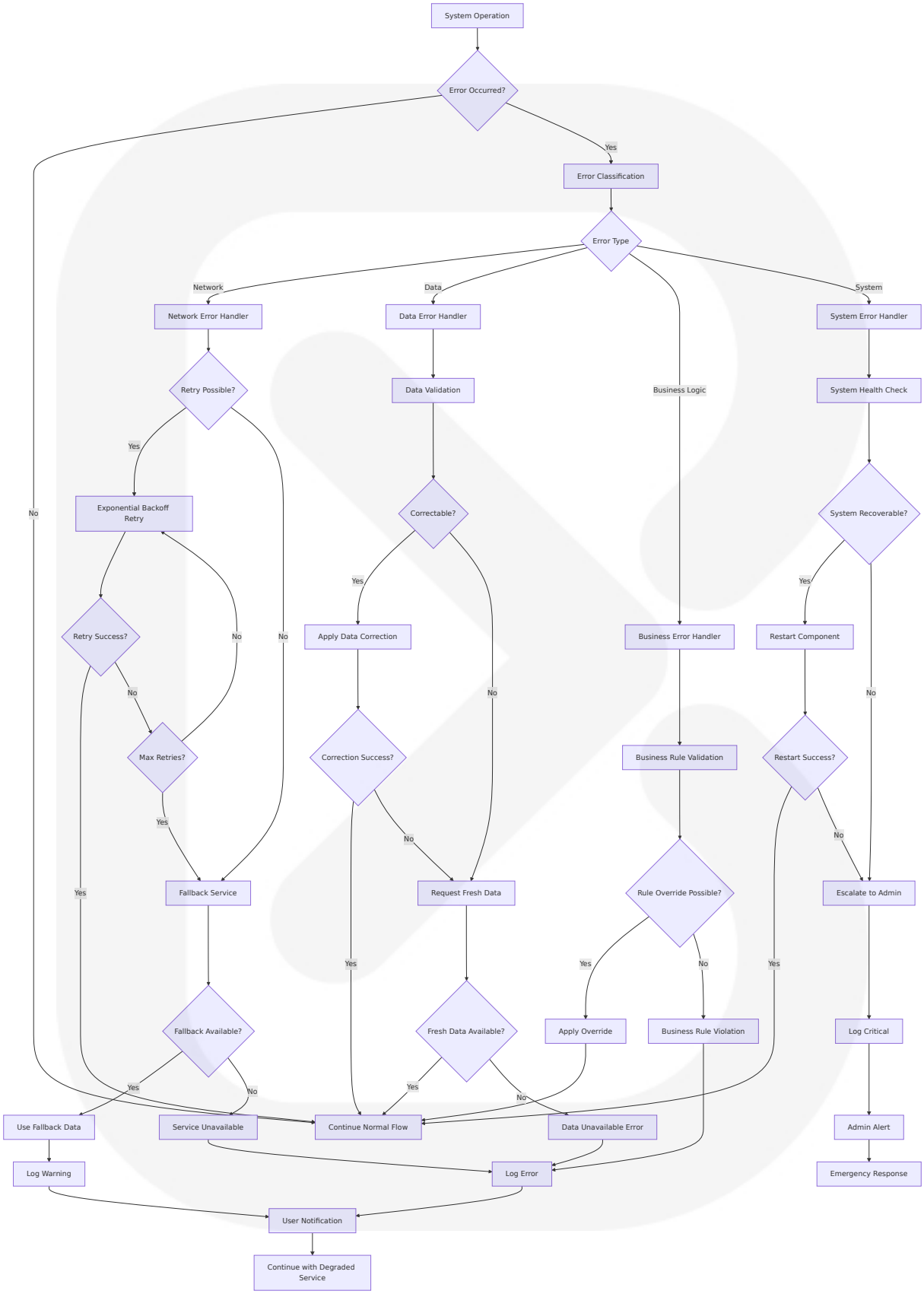

AppInitializing
AuthenticationRequired
Authenticating
Retry
Success
Failure
Authenticated
AuthenticationFailed
DashboardLoading
DashboardReady
User Request
View Portfolio
Back
Settings
Back
Logout
AnalysisRequested
PortfolioView
SettingsView
DataFetching
Retry
Data Available
Data Unavailable
DataProcessing
DataError
MLProcessing
Display Strategy
Show Error
StrategyGeneration
RiskAssessment
Risk High
Risk OK
RiskMitigation
Cannot Mitigate
Mitigated
StrategyReady
StrategyRejected

Data Persistence and Caching Strategy

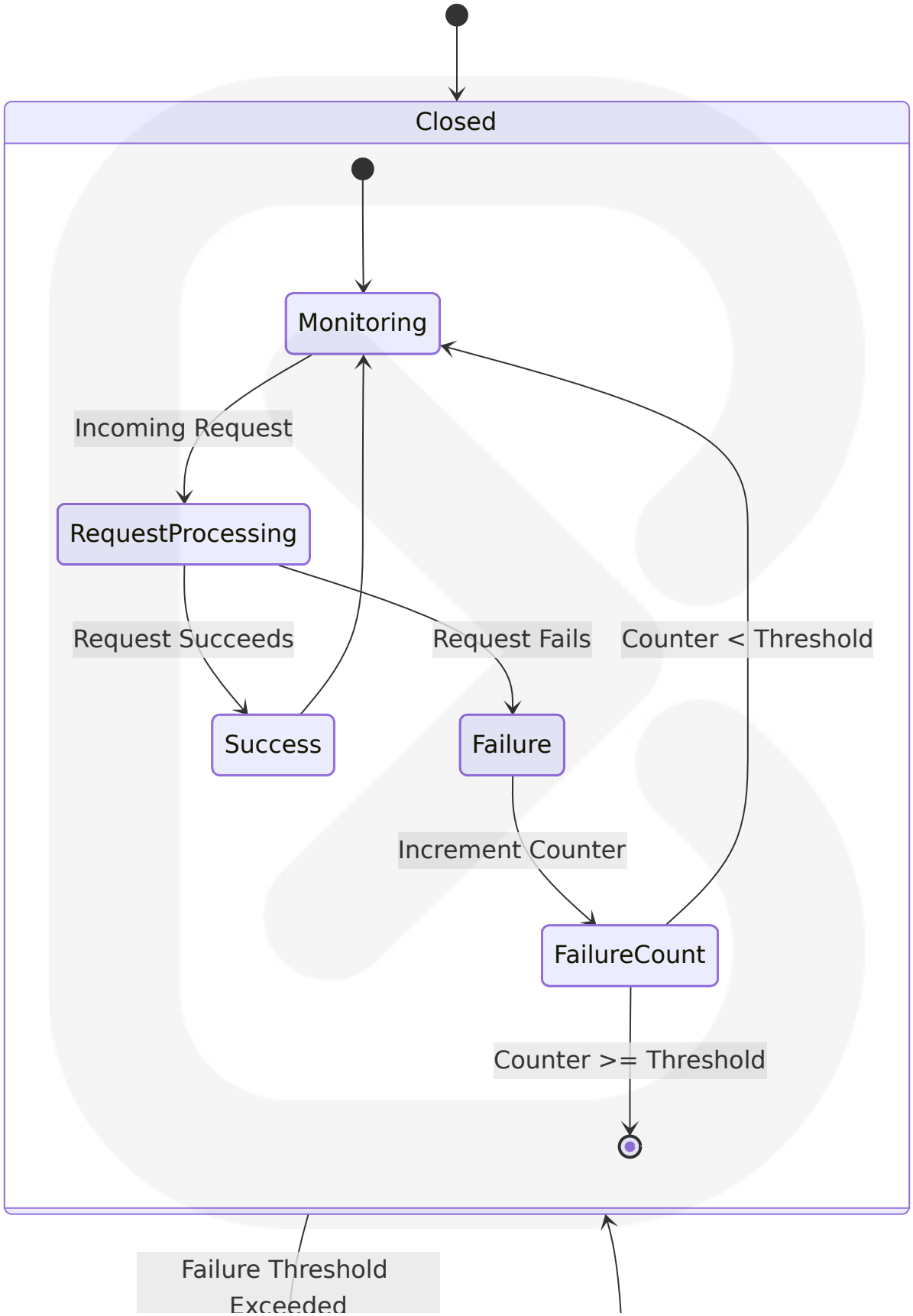


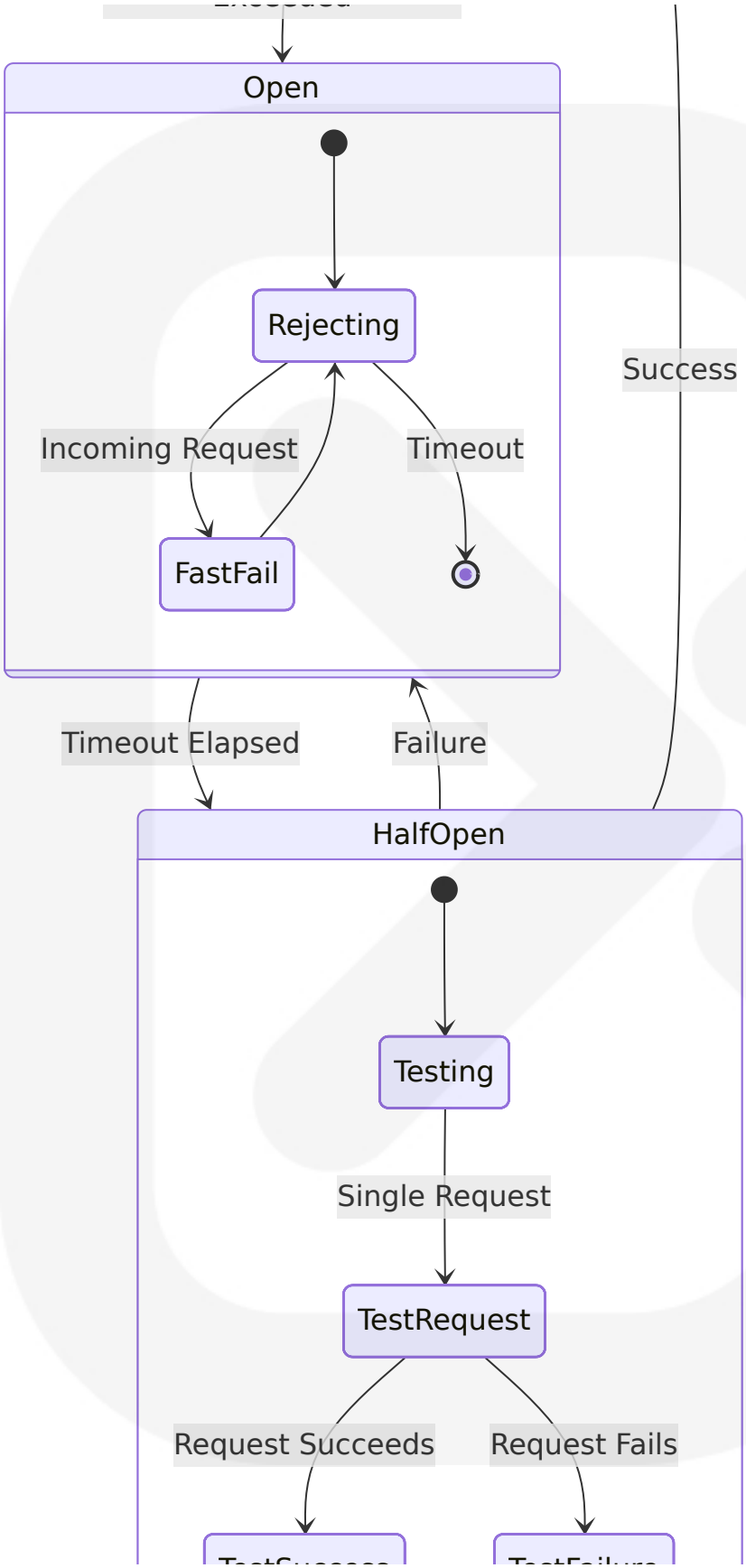
4.3.2 Error Handling and Recovery

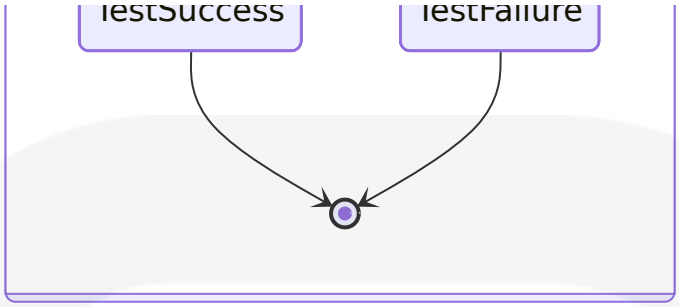
Comprehensive Error Handling Flow


System Operation
Error Occurred?
No
Yes
Error Classification
Error Type
Network
Data
Business Logic
System
Network Error Handler
Data Error Handler
System Error Handler
Retry Possible?
Data Validation
System Health Check
Yes
No
Exponential Backoff Retry
Correctable?
System Recoverable?
Retry Success?
Apply Data Correction
Business Error Handler
Restart Component
Max Retries?
Correction Success?
Business Rule Validation
Restart Success?
Fallback Service
Request Fresh Data
Rule Override Possible?
Escalate to Admin
Fallback Available?
Fresh Data Available?
Apply Override
Business Rule Violation
Log Critical
Use Fallback Data
Service Unavailable
Continue Normal Flow
Data Unavailable Error
Admin Alert
Log Warning
Log Error
Emergency Response
User Notification
Continue with Degraded Service

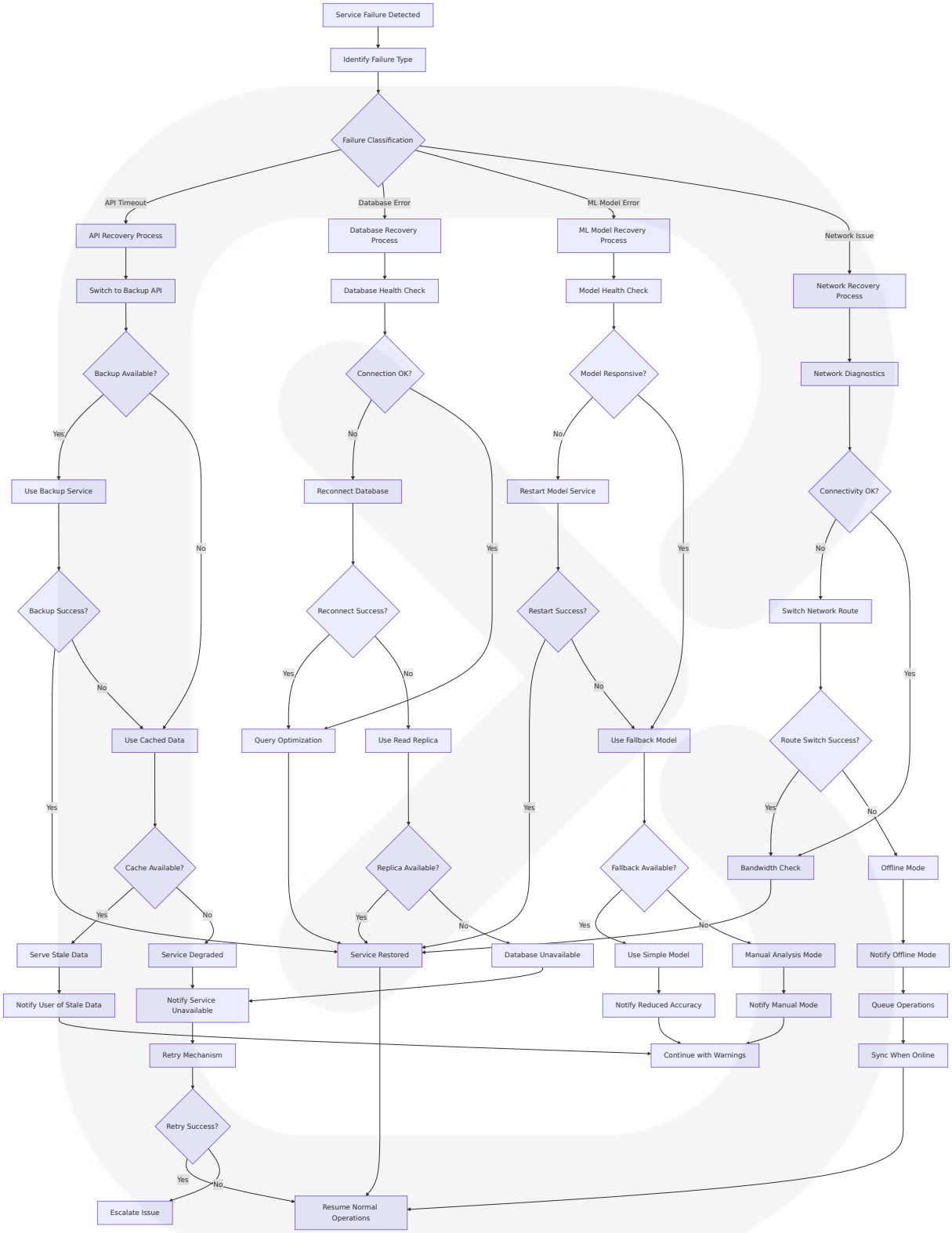
Circuit Breaker Pattern Implementation


Closed
Monitoring
Incoming Request
RequestProcessing
Request Succeeds
Request Fails
Counter < Threshold
Success
Failure
Increment Counter
FailureCount
Counter >= Threshold
Failure Threshold Exceeded


Open
Rejecting
Incoming Request
Timeout
FastFail
Success
Timeout Elapsed
Failure
HalfOpen
Testing
Single Request
TestRequest
Request Succeeds
Request Fails

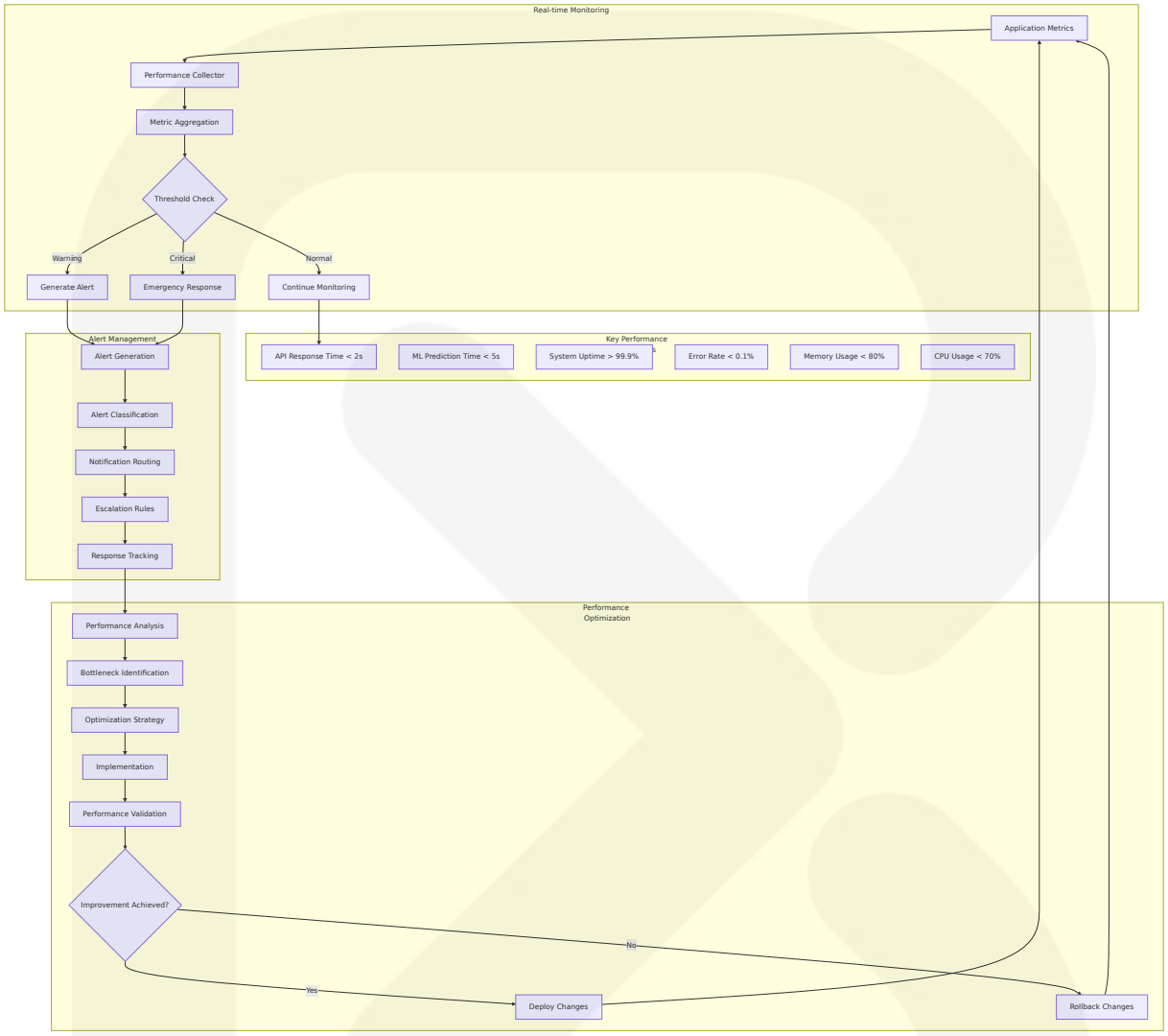


Recovery Procedures and Fallback Mechanisms

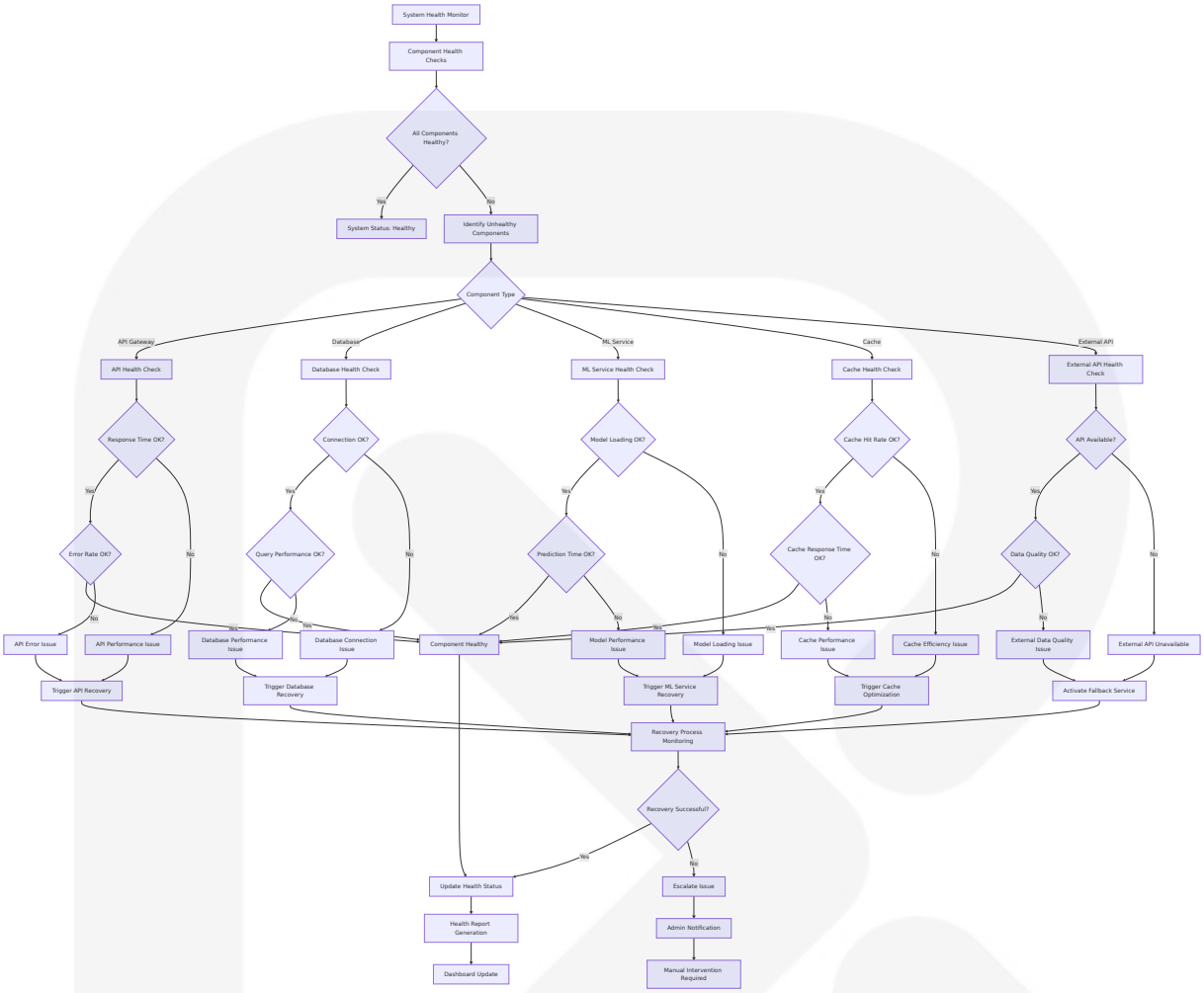


4.4 PERFORMANCE AND MONITORING

4.4.1 Performance Monitoring Workflow



4.4.2 System Health Monitoring



This comprehensive process flowchart section provides detailed workflows for the forex trading application, covering all major system processes from user authentication to machine learning model deployment. The workflows incorporate the latest Expo SDK 54 with React Native 0.81 features and ensure real-time performance with sub-2-second response times. The integration with TradingView's unofficial API wrapper provides comprehensive technical analysis capabilities, while the Flask-based machine learning pipeline enables real-time model deployment and prediction serving.

5. SYSTEM ARCHITECTURE

5.1 HIGH-LEVEL ARCHITECTURE

5.1.1 System Overview

The Forex Market Analysis and Trading Strategy Application employs a **microservices architecture** with a **mobile-first approach**, leveraging React Native 0.81 with React 19.1 and Expo SDK 54 with stable expo-file-system API providing object-oriented API for working with files and directories. The architecture follows **event-driven patterns** with **asynchronous processing** to handle real-time market data and machine learning model inference efficiently.

The system adopts a **layered architecture** approach with clear separation of concerns:

- **Presentation Layer**: React Native mobile application with Android 16 edge-to-edge enabled by default
- **API Gateway Layer**: Flask-based microservices orchestration
- **Business Logic Layer**: Python-based analytics and machine learning services
- **Data Layer**: MongoDB for persistence with Redis for caching and real-time data

Key Architectural Principles:

- **Loose Coupling**: Microservices are independent, self-contained programs with strong separation of responsibilities via well-defined specifications
- **High Cohesion**: Each service focuses on a single business capability
- **Fault Tolerance**: Failure in one service does not directly impact others, enhancing app resilience
- **Scalability**: Each service can be scaled independently based on its load, optimizing resource usage

System Boundaries:

- **Internal**: Mobile application, API gateway, analytics services, ML models, databases
- **External**: TradingView APIs, broker integrations, push notification services, cloud infrastructure

5.1.2 Core Components Table

Component Name	Primary Responsibility	Key Dependencies	Integration Points
Mobile Application	User interface and experience management	React Native 0.81, Expo SDK 54	API Gateway, Push Notifications
API Gateway	Request routing and service orchestration	Flask 3.1.2, Redis, Authentication	All microservices, External APIs
Market Data Service	Real-time and historical data acquisition	TradingView APIs, WebSocket connections	Redis Cache, MongoDB
Analytics Engine	Technical analysis and indicator calculations	TA-Lib, NumPy, Pandas	Market Data Service, ML Service

5.1.3 Data Flow Description

Primary Data Flows:
The system processes data through multiple concurrent streams. Market data flows from TradingView APIs through WebSocket connections into the Market Data Service, which validates and normalizes the data before storing it in MongoDB and caching frequently accessed data in Redis. The Analytics Engine subscribes to market data updates and performs real-time technical analysis calculations, feeding results to the ML Service for prediction generation.

Integration Patterns:
The architecture employs **publish-subscribe patterns** for real-time data

distribution, **request-response patterns** for synchronous API calls, and **message queuing** using Redis for asynchronous processing. Redis acts as an in-memory message queue for concurrent requests, with the model server polling Redis for batch processing and returning results.

Data Transformation Points:

- Raw market data normalization in Market Data Service
- Technical indicator calculations in Analytics Engine
- Feature engineering for ML models in ML Service
- Response formatting in API Gateway for mobile consumption

Key Data Stores:

- **MongoDB**: Persistent storage for user data, trading history, and model metadata
- **Redis**: Real-time data caching, session management, and message queuing
- **S3**: Historical data archives and ML model artifacts

5.1.4 External Integration Points

System Name	Integration Type	Data Exchange Pattern	Protocol/Format
TradingView API	Data Provider	Real-time streaming + REST	WebSocket/JSON, HTTP/JSON
Broker APIs	Trading Execution	Request-Response	REST/JSON
Push Notification Services	User Communication	Event-driven	HTTP/JSON
Cloud Storage (S3)	Data Persistence	Batch Upload/Download	REST/Binary

5.2 COMPONENT DETAILS

5.2.1 Mobile Application Component

Purpose and Responsibilities:
The mobile application serves as the primary user interface, built with React Native 0.81 with React 19.1 and Expo SDK 54 featuring stable expo-file-system API with object-oriented file and directory management. It manages user interactions, displays real-time market data, presents trading strategies, and handles offline capabilities.

Technologies and Frameworks:

- React Native 0.81 with Expo SDK 54
- Precompiled XCFrameworks reducing iOS build times from 120 seconds to 10 seconds
- Android 16 with edge-to-edge enabled by default
- Redux Toolkit for state management
- React Navigation for routing
- WebSocket client for real-time updates

Key Interfaces and APIs:

- RESTful API client for backend communication
- WebSocket client for real-time market data
- expo-file-system API for local file management with SAF URIs support on Android
- Push notification handling
- Biometric authentication integration

Data Persistence Requirements:

- Local SQLite database for offline data caching
- Secure storage for authentication tokens
- User preferences and settings persistence
- Trading strategy favorites and history

Scaling Considerations:

- Lazy loading of components and screens
- Image and data caching strategies
- Memory management for large datasets
- Background task optimization

5.2.2 API Gateway Component

Purpose and Responsibilities:
The API Gateway orchestrates all service communications, handles authentication and authorization, implements rate limiting, and provides a unified interface for the mobile application. Built with Flask for handling TradingView alerts and processing trading signals.

Technologies and Frameworks:

- Flask 3.1.2 with Gunicorn WSGI server
- Flask-JWT-Extended for authentication
- Flask-CORS for cross-origin requests
- Flask-SocketIO for WebSocket support
- Redis for session management and rate limiting

Key Interfaces and APIs:

- RESTful endpoints for mobile application
- WebSocket connections for real-time updates
- Service-to-service communication protocols
- External API proxy and aggregation
- Authentication and authorization middleware

Data Persistence Requirements:

- Session storage in Redis
- API rate limiting counters
- Request/response logging
- Service health monitoring data

Scaling Considerations:

- Horizontal scaling with load balancers
- Connection pooling for database access
- Caching strategies for frequently accessed data
- Circuit breaker patterns for external services

5.2.3 Market Data Service

Purpose and Responsibilities:
Manages real-time and historical market data acquisition from TradingView and other sources, implements data validation and quality checks, and provides normalized data feeds to other services.

Technologies and Frameworks:

- Python with tradingview-ta unofficial API wrapper
- WebSocket clients for real-time data streams
- Pandas for data manipulation
- Celery for background task processing
- Redis for data caching and message queuing

Key Interfaces and APIs:

- TradingView WebSocket and REST API integration
- Internal service APIs for data distribution
- Data validation and quality assurance pipelines
- Historical data batch processing interfaces

Data Persistence Requirements:

- Real-time data caching in Redis (1-5 second TTL)
- Historical data storage in MongoDB
- Data quality metrics and monitoring
- Backup and recovery procedures

Scaling Considerations:

- Multiple data source redundancy
- Horizontal scaling of data processing workers
- Data partitioning strategies
- Real-time data streaming optimization

5.2.4 Analytics Engine

Purpose and Responsibilities:
Performs comprehensive technical analysis using industry-standard indicators, implements pattern recognition algorithms, and provides real-time analysis results to the ML Service and mobile application.

Technologies and Frameworks:

- Python with TA-Lib for technical indicators
- NumPy and Pandas for numerical computations
- Technical analysis indicators including MA, EMA, MACD, Supertrend for signal generation
- Celery for distributed task processing

Key Interfaces and APIs:

- Market data subscription interfaces
- Technical indicator calculation APIs
- Pattern recognition service endpoints
- Real-time analysis result publishing

Data Persistence Requirements:

- Calculated indicator values caching (1-minute TTL)
- Pattern recognition results storage
- Analysis configuration and parameters
- Performance metrics and optimization data

Scaling Considerations:

- Parallel processing of multiple currency pairs
- Indicator calculation optimization
- Memory-efficient data structures
- Distributed computing for complex analyses

5.2.5 Machine Learning Service

Purpose and Responsibilities:
Hosts and serves machine learning models for market prediction, manages model training and retraining pipelines, and provides prediction confidence scoring and uncertainty estimation.

Technologies and Frameworks:

- Python with scikit-learn, TensorFlow, and Keras for ML model serving
- Redis message queue for batch processing and model inference requests
- Flask for model serving endpoints
- MLflow for model versioning and management
- Docker for model containerization

Key Interfaces and APIs:

- Model inference REST endpoints
- Batch prediction processing interfaces
- Model management and versioning APIs
- Performance monitoring and metrics collection

Data Persistence Requirements:

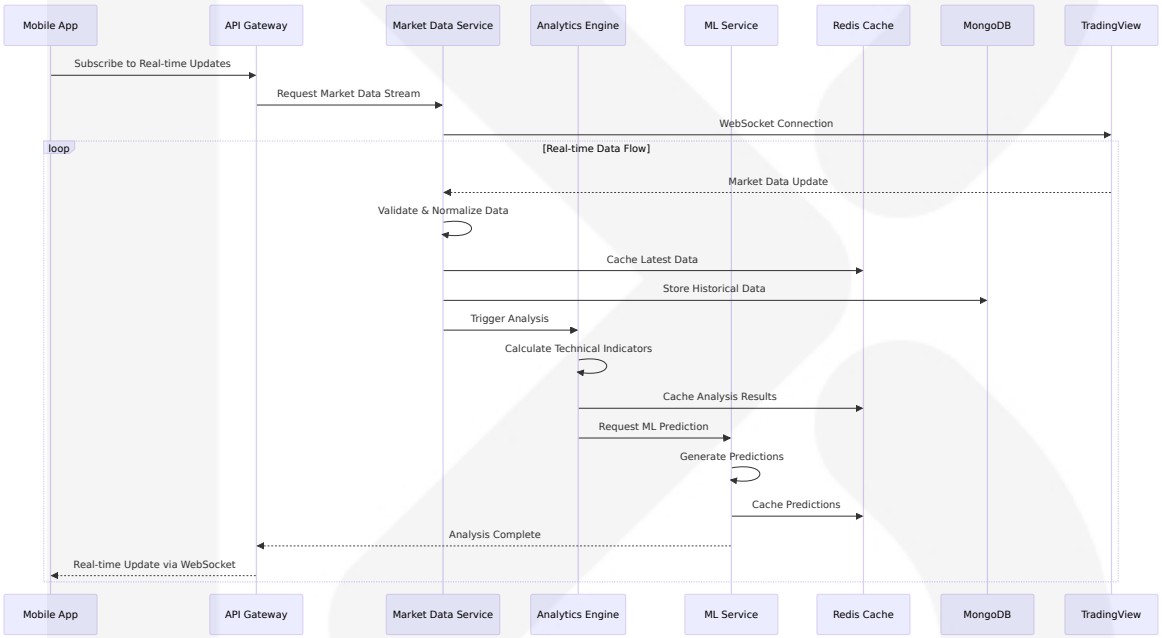
- Model artifacts and checkpoints storage
- Training data and feature stores
- Prediction results caching (15-minute TTL)
- Model performance metrics and drift detection

Scaling Considerations:

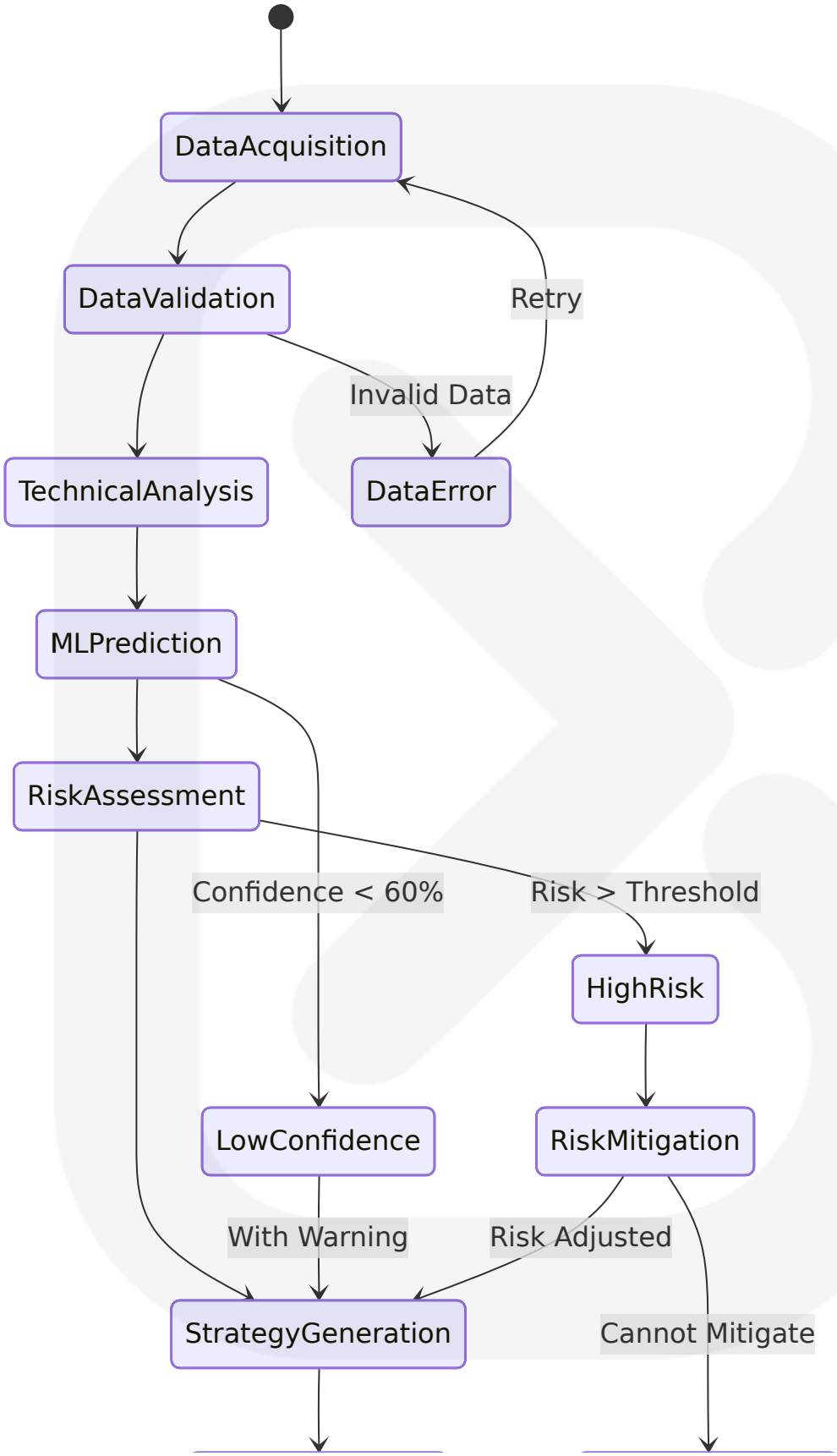
- Batch inference optimization for deep learning models, especially on GPU with tunable batch sizes
- Model serving horizontal scaling
- GPU resource management and allocation
- A/B testing infrastructure for model versions

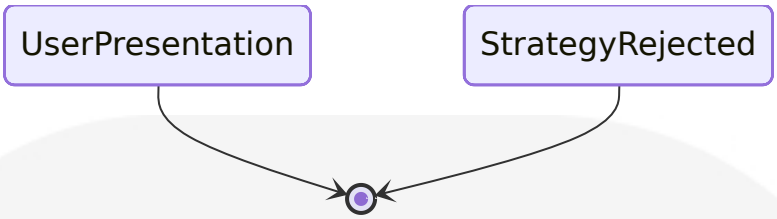
5.2.6 Component Interaction Diagrams

Real-time Data Processing Flow



Trading Strategy Generation Flow


DataAcquisition
DataValidation
Retry
Invalid Data
TechnicalAnalysis
DataError
MLPrediction
RiskAssessment
Confidence < 60%
Risk > Threshold
HighRisk
LowConfidence
RiskMitigation
With Warning
Risk Adjusted
StrategyGeneration
Cannot Mitigate



5.3 TECHNICAL DECISIONS

5.3.1 Architecture Style Decisions

Microservices vs Monolithic Architecture

Decision Factor	Microservices (Chosen)	Monolithic Alternative	Rationale
Team Scalability	Independent team development	Single team coordination required	Teams can work on different services simultaneously, speeding up development
Technology Flexibility	Different services can use different technologies, allowing teams to choose the best tool	Single technology stack	Optimal tool selection per service
Fault Isolation	Failure in one service does not directly impact others	Single point of failure	Enhanced system resilience
Deployment Independence	Updates to individual services can be deployed without affecting the entire system	Full application redeployment	Faster iteration cycles

Event-Driven vs Request-Response Communication

The architecture employs a **hybrid approach** combining both patterns:

- **Event-driven**: For real-time market data distribution and ML model triggers
- **Request-response**: For user-initiated actions and synchronous operations
- **Message queuing**: Redis as message queue for concurrent requests with model server polling for batch processing

5.3.2 Communication Pattern Choices

WebSocket vs HTTP Polling for Real-time Data

Aspect	WebSocket (Chosen)	HTTP Polling	Decision Rationale
Latency	<2 seconds real-time updates	5-30 seconds depending on interval	Critical for trading applications
Resource Efficiency	Persistent connection, lower overhead	Repeated connection establishment	Better mobile battery life
Scalability	Connection-based scaling	Request-based scaling	Suitable for real-time trading data

Synchronous vs Asynchronous Processing

The system implements **asynchronous processing** for computationally intensive operations:

- Asynchronous execution prevents client waiting, with Flask using Python RQ tool for background processing
- RQ (Redis Queue) enables background model processing with job_id tracking
- Synchronous processing reserved for lightweight operations and user authentication

5.3.3 Data Storage Solution Rationale

MongoDB vs Relational Database

Requirement	MongoDB (Chosen)	PostgreSQL Alternative	Decision Factors
Schema Flexibility	Document-based, flexible schemas	Fixed relational schemas	Trading data varies by instrument type
Horizontal Scaling	Built-in sharding and replication	Complex sharding setup	Growth scalability requirements
JSON Integration	Native BSON support	JSON column types	ObjectId handling and JSON conversion requirements
Development Speed	Flask-PyMongo integration simplicity	ORM complexity	Rapid development needs

Redis Caching Strategy

Redis chosen as the optimal solution for in-memory data stores and queuing operations with differentiated TTL strategies:

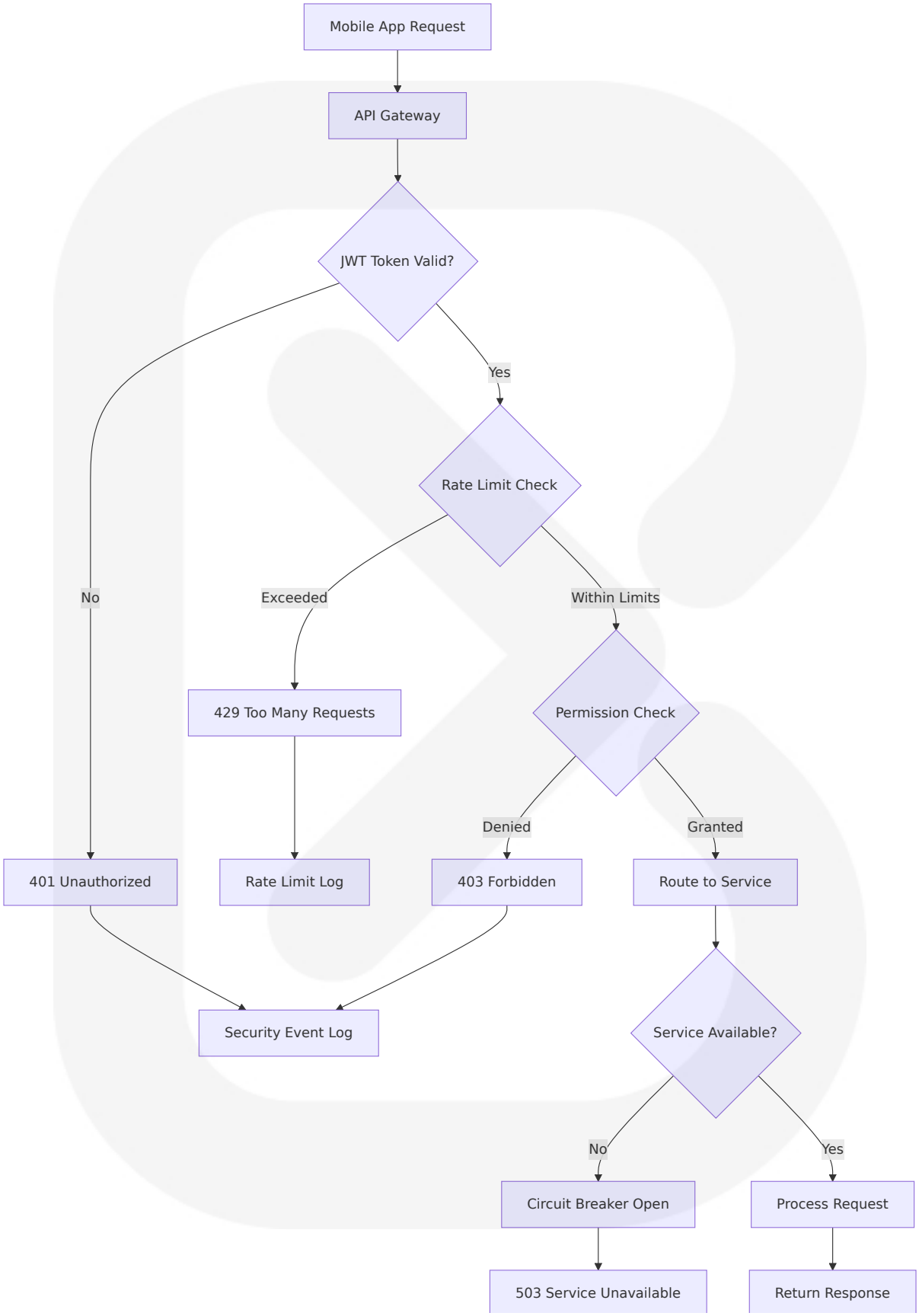
- Real-time price data: 1-5 second TTL
- Technical indicators: 1-minute TTL
- ML predictions: 15-minute TTL
- User sessions: 24-hour TTL

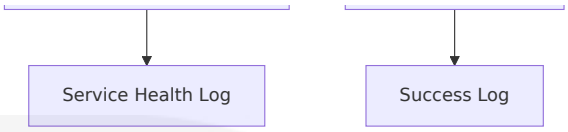
5.3.4 Security Mechanism Selection

JWT vs Session-based Authentication

Security Aspect	JWT (Chosen)	Session-based	Implementation Benefits
Statelessness	Fully stateless tokens	Server-side session storage	Microservices compatibility
Scalability	No server-side storage required	Session store scaling needed	Horizontal scaling support
Mobile Optimization	Token-based, offline capable	Requires constant server connection	Better mobile experience

API Security Architecture


Mobile App Request
API Gateway
JWT Token Valid?
No
Yes
Rate Limit Check
Exceeded
Within Limits
429 Too Many Requests
Permission Check
Denied
Granted
401 Unauthorized
Rate Limit Log
403 Forbidden
Route to Service
Security Event Log
Service Available?
No
Yes
Circuit Breaker Open
Process Request
503 Service Unavailable
Return Response



5.4 CROSS-CUTTING CONCERNS

5.4.1 Monitoring and Observability Approach

Three Pillars of Observability:

- **Metrics**: System performance, business KPIs, and SLA monitoring
- **Logs**: Structured logging across all services with correlation IDs
- **Traces**: Distributed tracing for request flow analysis

Monitoring Stack:

- **Application Monitoring**: New Relic for performance metrics and error tracking
- **Infrastructure Monitoring**: AWS CloudWatch for system resources
- **Log Aggregation**: Centralized logging with structured JSON format
- **Real-time Alerting**: PagerDuty integration for critical system events

Key Performance Indicators:

- API response times (<2 seconds for real-time data)
- ML model inference latency (<5 seconds)
- System uptime (>99.9% availability)
- Error rates (<0.1% for critical paths)

5.4.2 Logging and Tracing Strategy

Structured Logging Format:

```
{
  "timestamp": "2025-01-15T10:30:00Z",
```

```
    "level": "INFO",
    "service": "market-data-service",
    "correlation_id": "req-123456",
    "user_id": "user-789",
    "message": "Market data processed",
    "metadata": {
      "currency_pair": "EURUSD",
      "processing_time_ms": 150
    }
  }
```

Distributed Tracing Implementation:

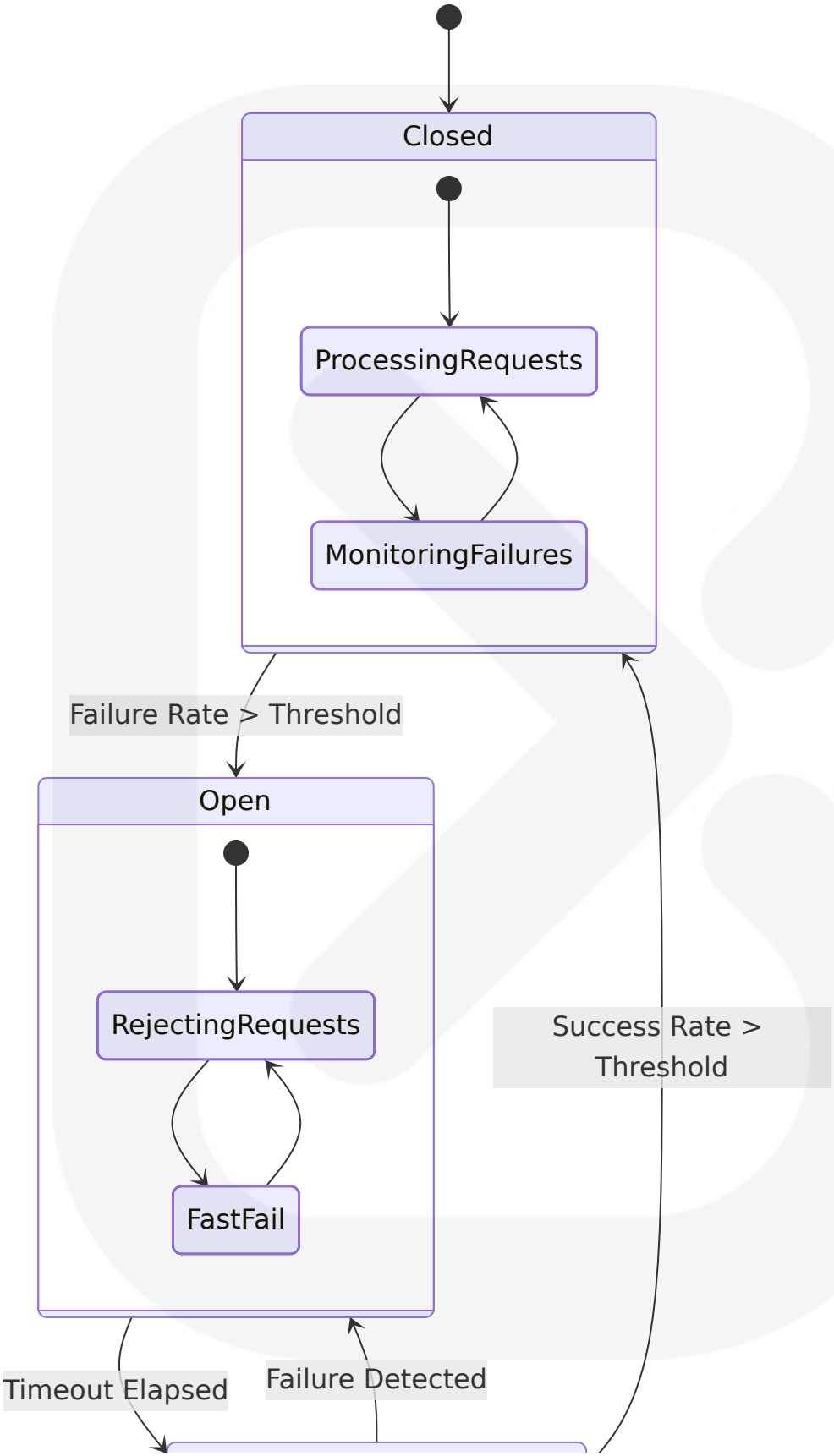
- Correlation ID propagation across service boundaries
- Request lifecycle tracking from mobile app to database
- Performance bottleneck identification
- Error propagation analysis

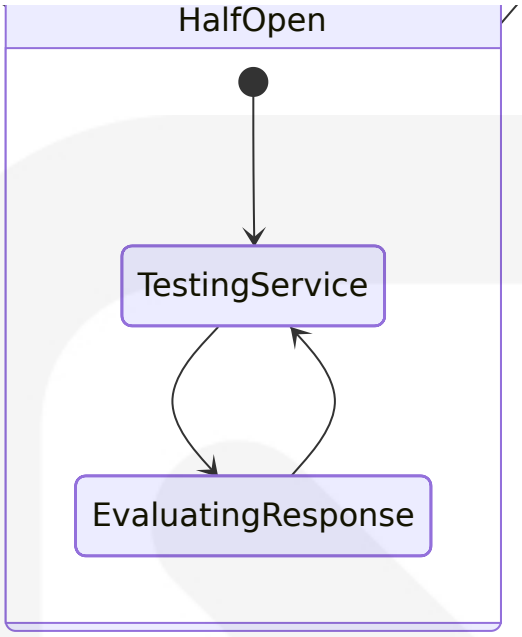
Log Retention Policies:

- Critical logs: 1 year retention
- Application logs: 90 days retention
- Debug logs: 30 days retention
- Audit logs: 7 years retention (compliance requirement)

5.4.3 Error Handling Patterns

Circuit Breaker Pattern Implementation:


Closed
ProcessingRequests
MonitoringFailures
Failure Rate > Threshold
Open
RejectingRequests
FastFail
Success Rate > Threshold
Timeout Elapsed
Failure Detected



Error Classification and Handling:

Error Type	Handling Strategy	Recovery Mechanism	User Impact
Network Timeouts	Exponential backoff retry	Fallback to cached data	Graceful degradation
Data Validation Errors	Input sanitization	Request fresh data	Error message display
ML Model Failures	Fallback to simpler models	Manual analysis mode	Reduced accuracy warning
Authentication Errors	Token refresh attempt	Re-authentication flow	Login prompt

5.4.4 Authentication and Authorization Framework

OAuth 2.0 + JWT Implementation:

- **Authorization Server**: Centralized authentication service
- **Resource Servers**: Individual microservices with token validation
- **Client Applications**: Mobile app with secure token storage

- **Refresh Token Rotation**: Enhanced security with token rotation

Role-Based Access Control (RBAC):

- **Roles**: Basic User, Premium User, Admin
- **Permissions**: Market data access, trading features, administrative functions
- **Scope-based Authorization**: Fine-grained API endpoint access control

Security Headers and Policies:

- HTTPS enforcement for all communications
- Content Security Policy (CSP) implementation
- Cross-Origin Resource Sharing (CORS) configuration
- API rate limiting per user and endpoint

5.4.5 Performance Requirements and SLAs

Service Level Agreements:

Service Component	Availability SLA	Response Time SLA	Throughput SLA
API Gateway	99.9% uptime	<500ms average	1000 requests/second
Market Data Service	99.95% uptime	<2 seconds real-time	10,000 updates/second
ML Service	99.5% uptime	<5 seconds inference	100 predictions/second
Mobile Application	N/A	<1 second UI response	N/A

Performance Optimization Strategies:

- **Caching**: Multi-level caching with Redis and CDN

- **Database Optimization**: Index optimization and query performance tuning
- **Connection Pooling**: Efficient database connection management
- **Load Balancing**: Horizontal scaling with intelligent request distribution

5.4.6 Disaster Recovery Procedures

Backup and Recovery Strategy:

- **Database Backups**: Daily automated backups with point-in-time recovery
- **Code Repository**: Git-based version control with multiple remote repositories
- **Configuration Management**: Infrastructure as Code with version control
- **Data Replication**: Multi-region data replication for critical datasets

Recovery Time Objectives (RTO) and Recovery Point Objectives (RPO):

- **Critical Services**: RTO <1 hour, RPO <15 minutes
- **Non-critical Services**: RTO <4 hours, RPO <1 hour
- **Data Loss Prevention**: Real-time replication for trading data
- **Service Restoration**: Automated failover with manual verification

Business Continuity Planning:

- **Incident Response Team**: 24/7 on-call rotation
- **Communication Protocols**: Stakeholder notification procedures
- **Service Degradation**: Graceful degradation with core functionality preservation
- **Post-incident Analysis**: Root cause analysis and improvement implementation

The system architecture provides a robust, scalable foundation for the forex trading application, leveraging modern technologies and proven architectural patterns to deliver high-performance real-time trading capabilities while maintaining system reliability and user experience quality.

6. SYSTEM COMPONENTS DESIGN

6.1 MOBILE APPLICATION ARCHITECTURE

6.1.1 React Native Application Structure

Core Application Framework:
The mobile application leverages React Native 0.81 with React 19.1, delivered through Expo SDK 54. The application utilizes the stable expo-file-system API with object-oriented API for working with files and directories, supporting SAF URIs on Android and bundled assets on both iOS and Android.

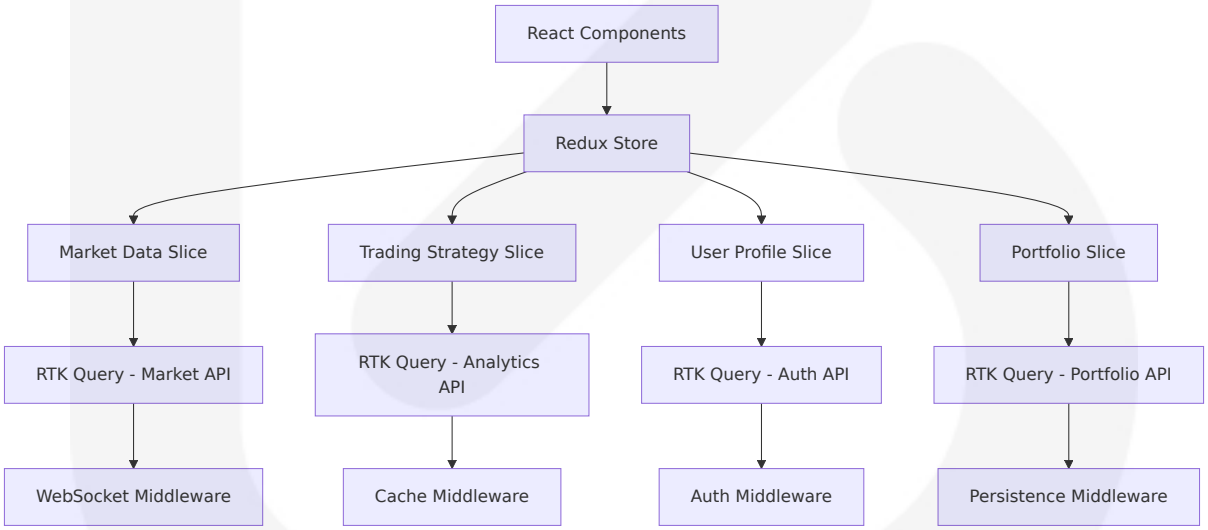
Performance Optimizations:
React Native 0.81 introduces precompiled iOS builds, cutting compile times by up to 10x in projects where React Native is the primary dependency. The precompiled XCFrameworks reduced clean build times for RNTester from about 120 seconds to 10 seconds on an M4 Max chip.

Platform-Specific Features:
With Expo SDK 54 and React Native 0.81 now targeting Android 16, edge-to-edge will be enabled in all Android apps, and cannot be disabled. expo-sqlite now includes a drop-in implementation for the localStorage web API.

6.1.2 Component Architecture Design

Component Layer	Technology Stack	Primary Responsibilities	Performance Considerations
Presentation Layer	React Native 0.81 + React 19.1	UI rendering, user interactions	Lazy loading, component memoization
State Management	Redux Toolkit + RTK Query	Global state, API caching	Normalized state structure, selective updates
Navigation	React Navigation 6.x	Screen routing, deep linking	Stack-based navigation with gesture handling
Data Layer	AsyncStorage + SQLite	Local persistence, offline support	Optimized queries, data compression

State Management Architecture:



Component Hierarchy:

- **Screen Components**: Main navigation screens (Dashboard, Analysis, Portfolio, Settings)
- **Container Components**: Business logic containers with Redux connections

- **Presentation Components**: Pure UI components with props-based rendering
- **Utility Components**: Reusable components (Charts, Forms, Modals)

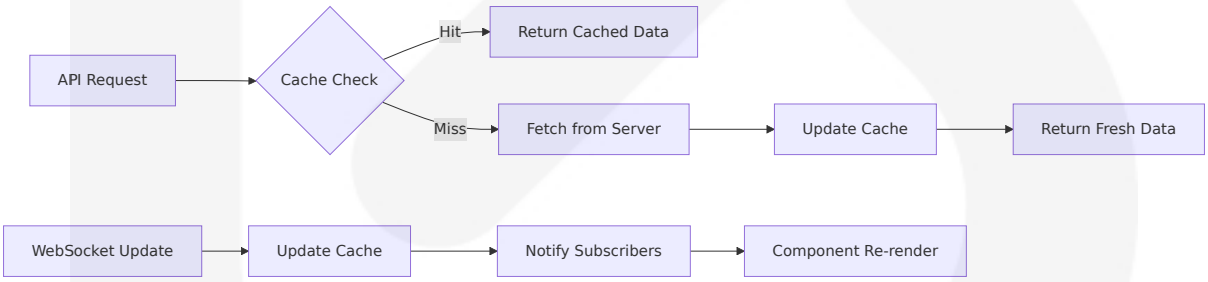
6.1.3 Real-time Data Integration

WebSocket Connection Management:
The application maintains persistent WebSocket connections for real-time market data updates with automatic reconnection logic and connection health monitoring. Connection pooling ensures efficient resource utilization while handling multiple currency pair subscriptions simultaneously.

Data Synchronization Strategy:

- **Optimistic Updates**: Immediate UI updates with server confirmation
- **Conflict Resolution**: Last-write-wins with user notification for conflicts
- **Offline Queue**: Local storage of actions for replay when connection restored
- **Data Validation**: Client-side validation with server-side verification

Caching Implementation:



6.1.4 Offline Capabilities and Data Persistence

Local Storage Architecture:
The application leverages expo-sqlite with localStorage web API implementation for structured data storage, combined with AsyncStorage

for simple key-value persistence and secure storage for sensitive authentication tokens.

Offline-First Design:

- **Data Synchronization**: Bidirectional sync with conflict resolution
- **Queue Management**: Action queuing for offline operations
- **Cache Strategies**: Intelligent caching with TTL-based expiration
- **User Experience**: Seamless offline/online transitions with status indicators

Data Persistence Layers:

Storage Type	Use Case	Technology	Capacity Limits
Secure Storage	Authentication tokens, user credentials	Expo Secure Store	2KB per item
Structured Data	Trading history, cached market data	SQLite	Device storage dependent
Key-Value Store	User preferences, app settings	AsyncStorage	6MB recommended
File System	Charts, documents, cached images	expo-file-system	Device storage dependent

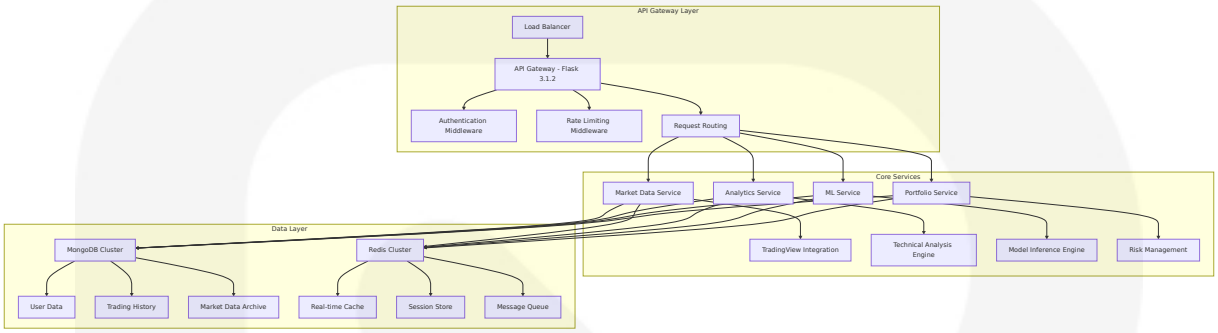
6.2 BACKEND SERVICE ARCHITECTURE

6.2.1 Flask-Based Microservices Design

Core Service Framework:
Flask is one of the most popular Python frameworks for small apps, APIs, and data science dashboards, providing a lightweight, unopinionated web framework that gives full control over application architecture. Flask is popular for data science and ML workflows, frequently used for

experimentation like building dashboards, serving models, or turning notebooks into lightweight web apps.

Service Architecture Pattern:



Service Communication Patterns:

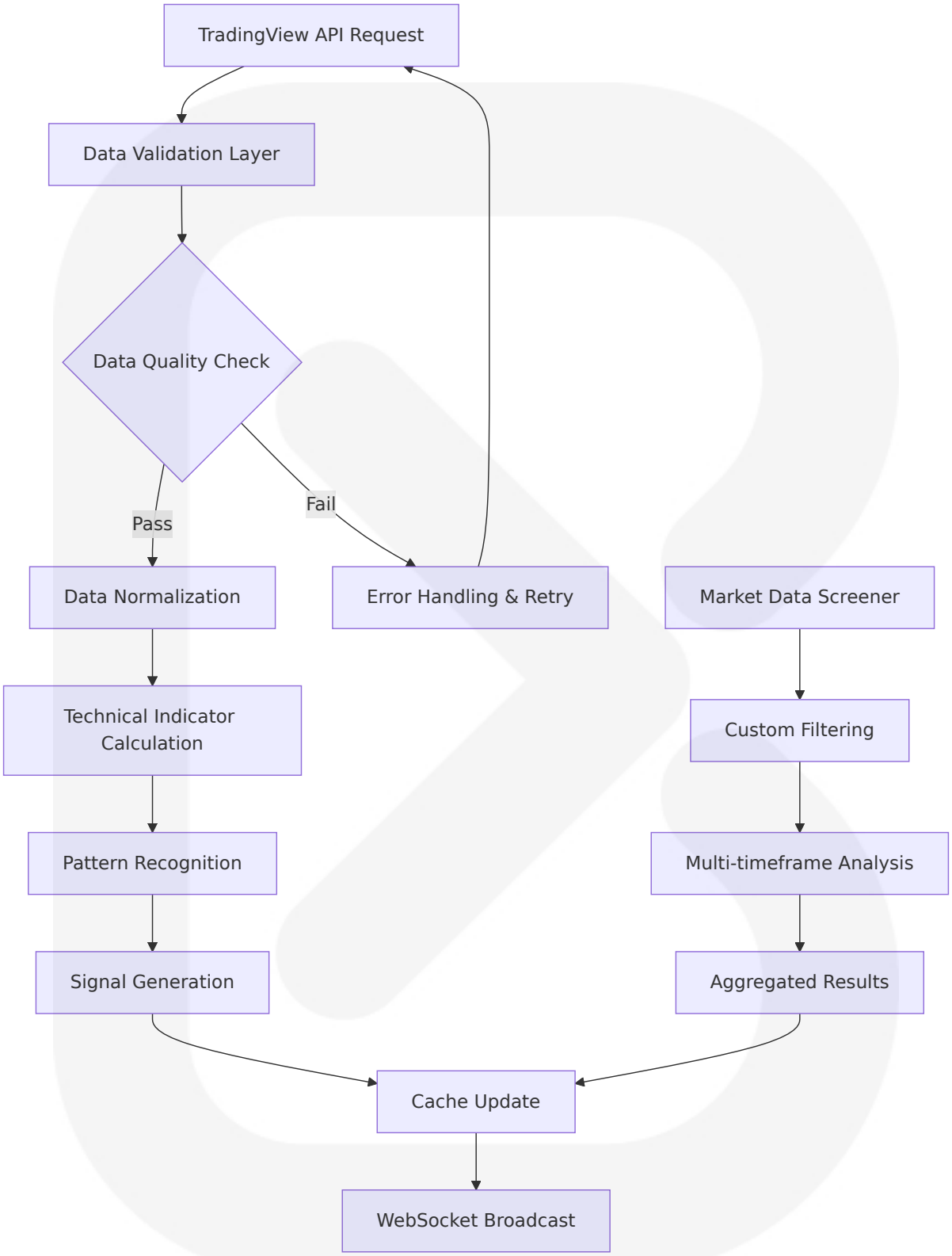
- **Synchronous Communication**: REST APIs for request-response operations
- **Asynchronous Communication**: Message queues for background processing
- **Event-Driven Architecture**: Pub/sub patterns for real-time updates
- **Circuit Breaker Pattern**: Fault tolerance and service isolation

6.2.2 TradingView API Integration Layer

API Wrapper Implementation:
The system utilizes unofficial python API wrapper to retrieve technical analysis from TradingView, allowing fetch of technical analysis data. The TA_Handler system provides technical analysis with recommendation outputs like "BUY", "NEUTRAL", "SELL".

Technical Analysis Integration:
The system processes technical analysis data with summary outputs containing recommendation, buy, neutral, and sell signals. Additionally, tradingview-screener allows creation of custom stock screeners using TradingView's official API, retrieving data directly without web scraping.

Data Processing Pipeline:



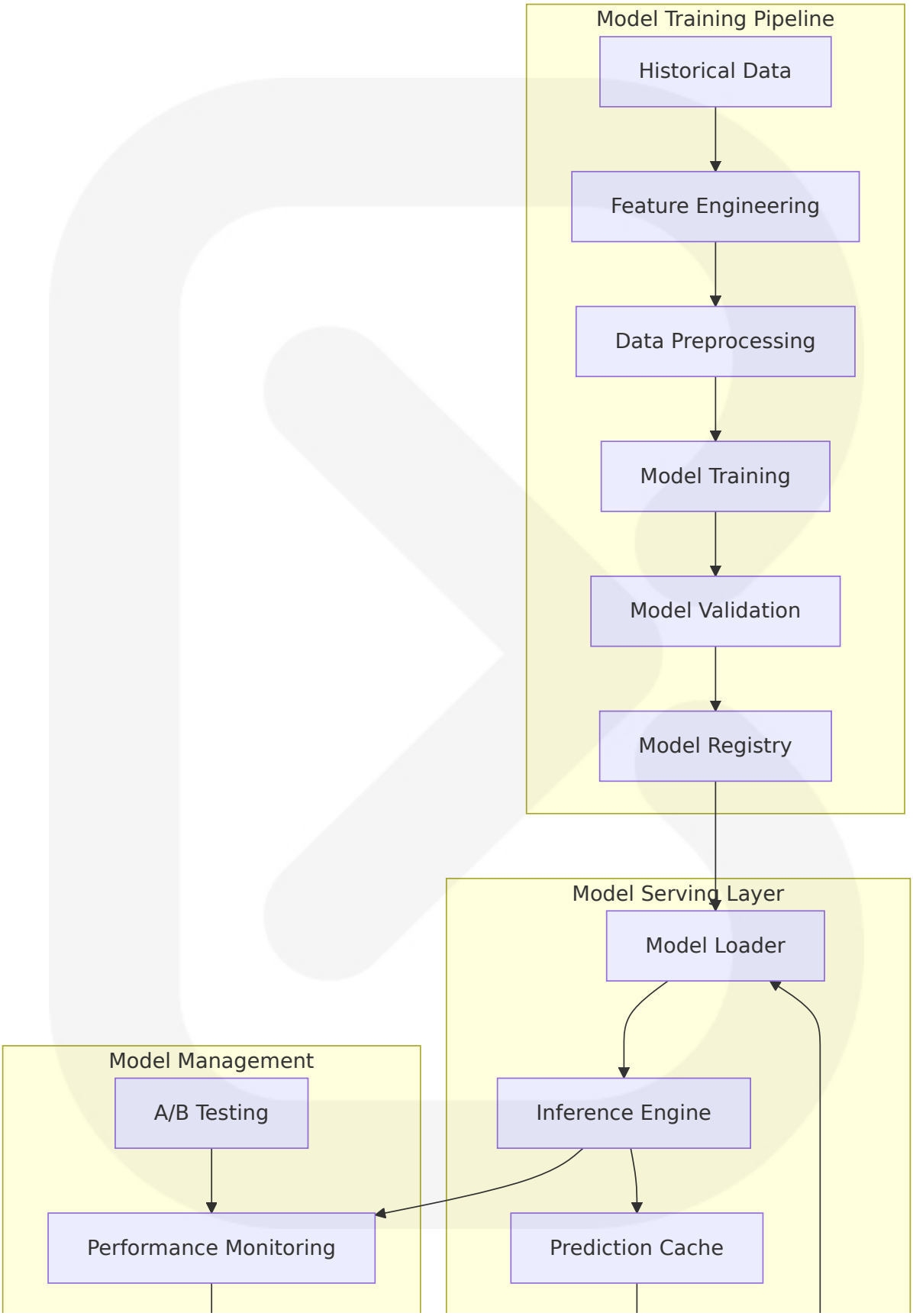
API Rate Limiting and Optimization:

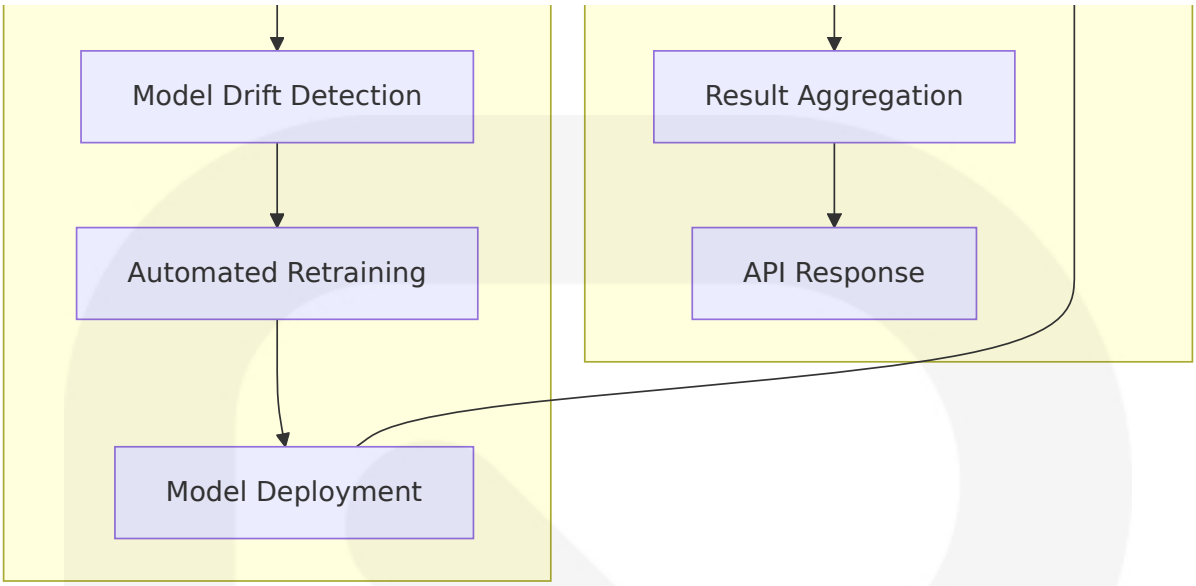
- **Request Batching**: Combine multiple symbol requests
- **Intelligent Caching**: TTL-based caching with smart invalidation
- **Fallback Mechanisms**: Alternative data sources for redundancy
- **Connection Pooling**: Efficient HTTP connection management

6.2.3 Machine Learning Service Architecture

ML Model Serving Infrastructure:
The system provides machine learning models applied to financial market predictions using TensorFlow, Keras, and Sci-kit Learn, utilizing past 500 days of data for forex pairs with technical indicators. Python's popularity and rich ecosystem of libraries, coupled with simplicity of implementing Machine Learning have made machine learning for algorithmic trading in Python a popular choice.

Model Architecture Design:


Model Training Pipeline
Historical Data
Feature Engineering
Data Preprocessing
Model Training
Model Validation
Model Registry
Model Serving Layer
Model Loader
Inference Engine
Prediction Cache
Model Management
A/B Testing
Performance Monitoring



Supported ML Frameworks:

Framework	Use Case	Model Types	Performance Characteristics
Scikit-learn	Traditional ML algorithms	Classification, Regression, Clustering	Fast inference, interpretable
TensorFlow	Deep learning models	Neural networks, LSTM, CNN	GPU acceleration, complex patterns
Keras	High-level neural networks	Sequential, Functional, Custom	Rapid prototyping, easy deployment
LightGBM	Gradient boosting	Tree-based models	Memory efficient, fast training

Model Inference Optimization:

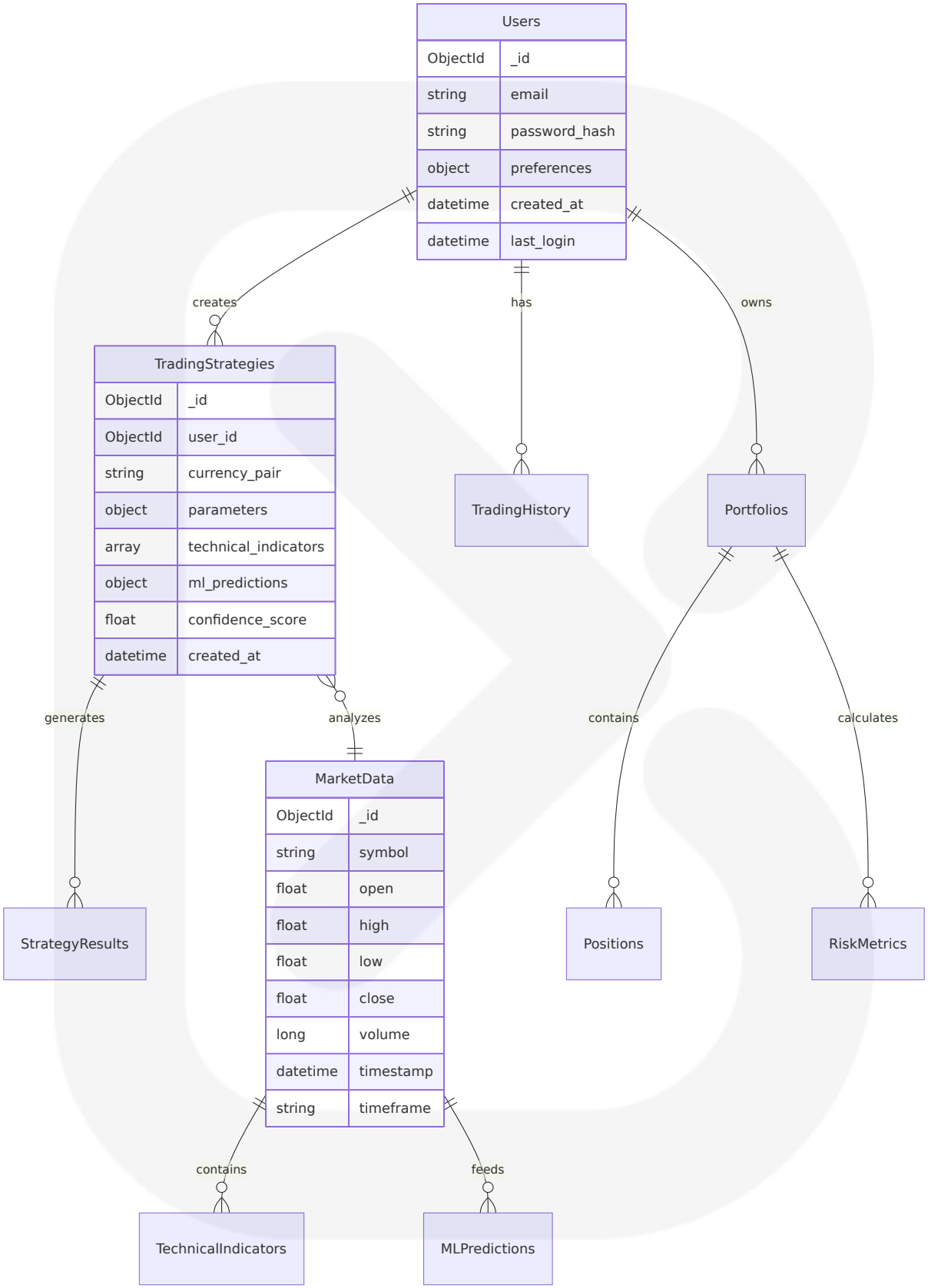
- **Batch Processing**: Group predictions for efficiency
- **Model Caching**: In-memory model storage for fast access
- **GPU Utilization**: CUDA acceleration for deep learning models
- **Async Processing**: Non-blocking inference with result queuing

6.2.4 Database and Storage Architecture

MongoDB Implementation:

PyMongo is a native Python driver for MongoDB, offering both synchronous and asynchronous APIs, supporting MongoDB 4.0, 4.2, 4.4, 5.0, 6.0, 7.0, and 8.0. The bson package provides BSON format implementation, while gridfs package offers gridfs implementation on top of pymongo.

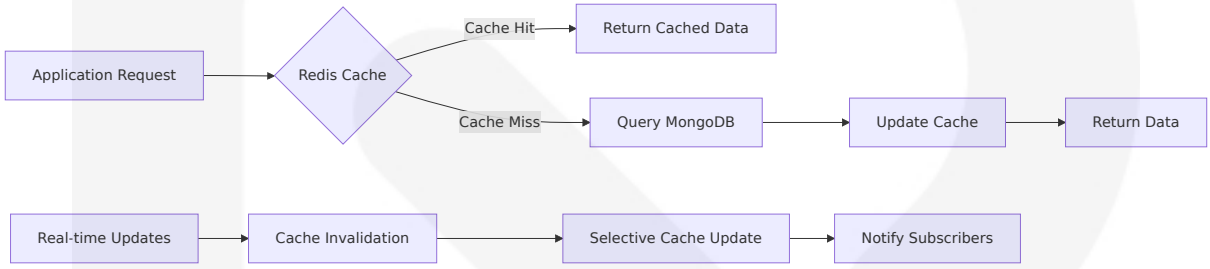
Database Schema Design:


Users
ObjectId _id
string email
string password_hash
object preferences
datetime created_at
datetime last_login
creates
has
owns
TradingStrategies
ObjectId _id
ObjectId user_id
string currency_pair
object parameters
array technical_indicators
object ml_predictions
float confidence_score
datetime created_at
TradingHistory
Portfolios
generates
analyzes
contains
calculates
MarketData
ObjectId _id
string symbol
float open
float high
float low
float close
long volume
datetime timestamp
string timeframe
StrategyResults
Positions
RiskMetrics
contains
feeds
TechnicalIndicators
MLPredictions

Data Partitioning Strategy:

- **Horizontal Partitioning**: Shard by currency pair and time range
- **Vertical Partitioning**: Separate hot and cold data
- **Index Optimization**: Compound indexes for query performance
- **Data Archival**: Automated archival of historical data

Redis Caching Architecture:



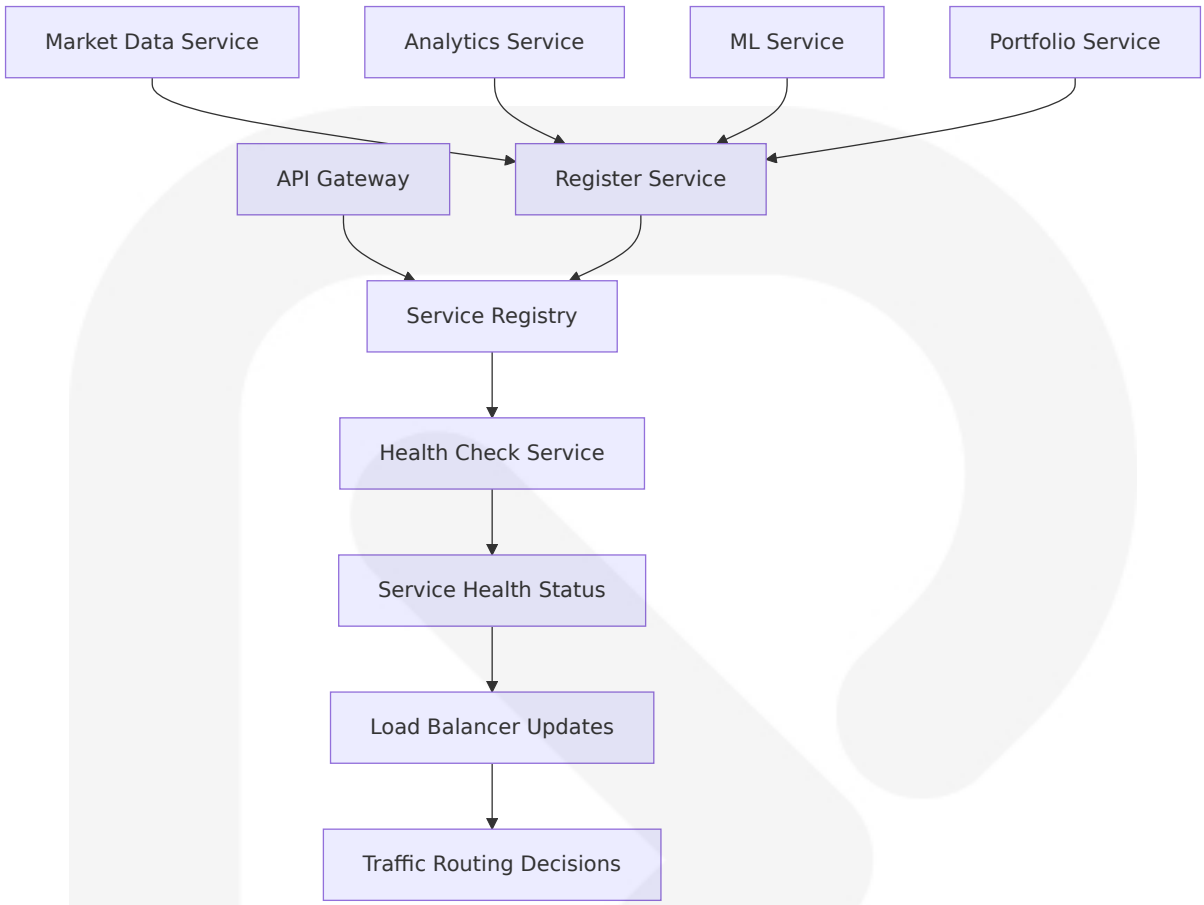
6.3 INTEGRATION PATTERNS AND COMMUNICATION

6.3.1 API Gateway Design Pattern

Gateway Responsibilities:

- **Request Routing**: Intelligent routing based on service availability and load
- **Authentication & Authorization**: Centralized security enforcement
- **Rate Limiting**: Per-user and per-endpoint throttling
- **Request/Response Transformation**: Data format standardization
- **Monitoring & Logging**: Centralized observability

Service Discovery Mechanism:



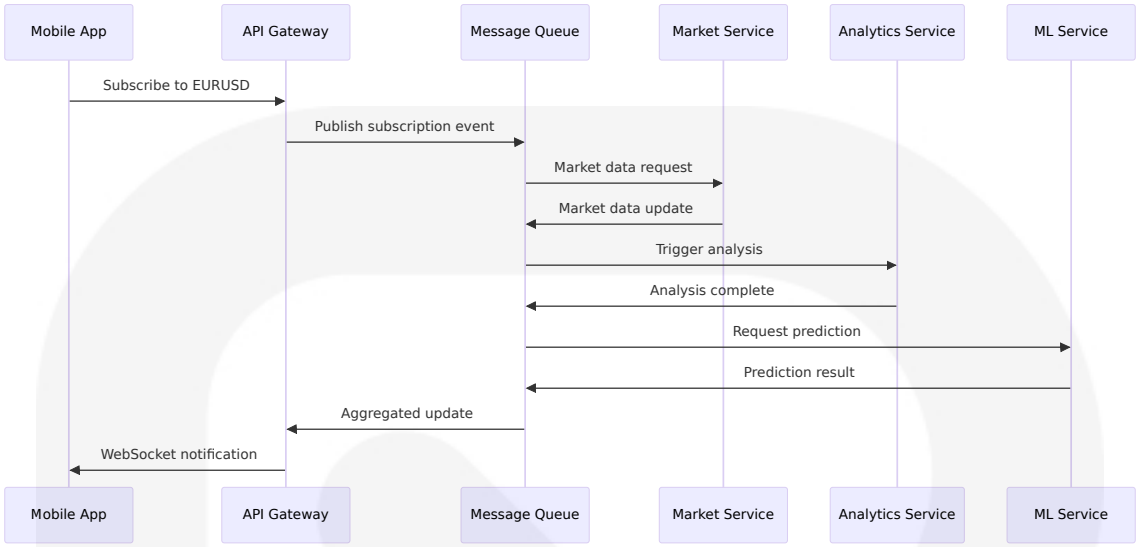
Circuit Breaker Implementation:

- **Failure Threshold**: Configurable failure rate triggers
- **Timeout Management**: Request timeout with exponential backoff
- **Fallback Strategies**: Cached responses and alternative services
- **Recovery Testing**: Automated health checks for service restoration

6.3.2 Event-Driven Architecture

Message Queue Implementation:
Redis serves as the primary message broker for asynchronous communication between services, supporting pub/sub patterns for real-time data distribution and task queues for background processing.

Event Flow Architecture:



Event Types and Handlers:

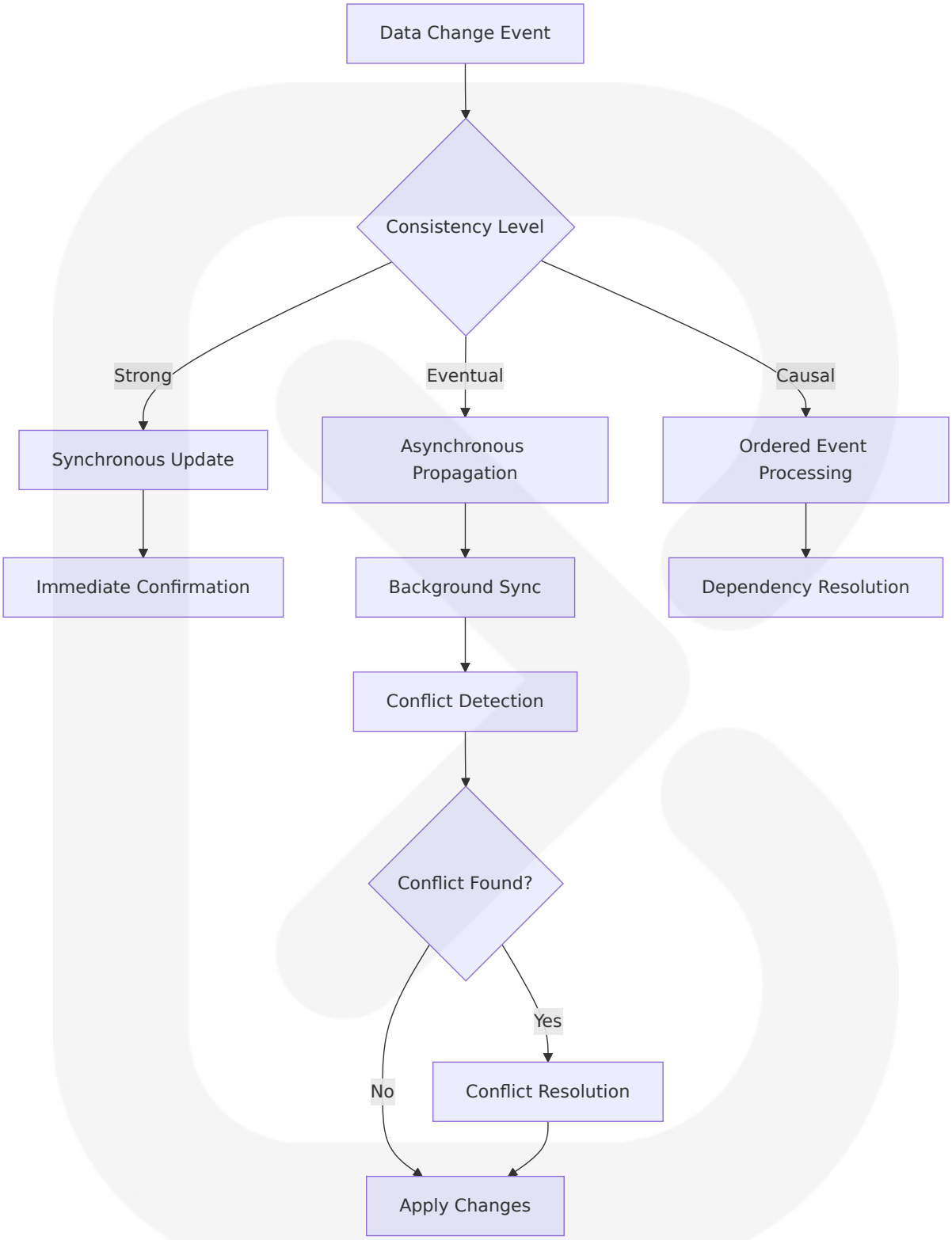
Event Type	Producer	Consumer	Processing Pattern
Market Data Update	Market Data Service	Analytics, ML Services	Fan-out
Analysis Complete	Analytics Service	ML Service, API Gateway	Pipeline
Prediction Ready	ML Service	API Gateway, Portfolio Service	Broadcast
User Action	API Gateway	Relevant Services	Direct routing

6.3.3 Data Consistency and Synchronization

Consistency Models:

- **Eventual Consistency**: For non-critical data like historical analysis
- **Strong Consistency**: For user accounts and trading positions
- **Causal Consistency**: For related trading events and decisions
- **Session Consistency**: For user-specific data within sessions

Synchronization Strategies:



Conflict Resolution Mechanisms:

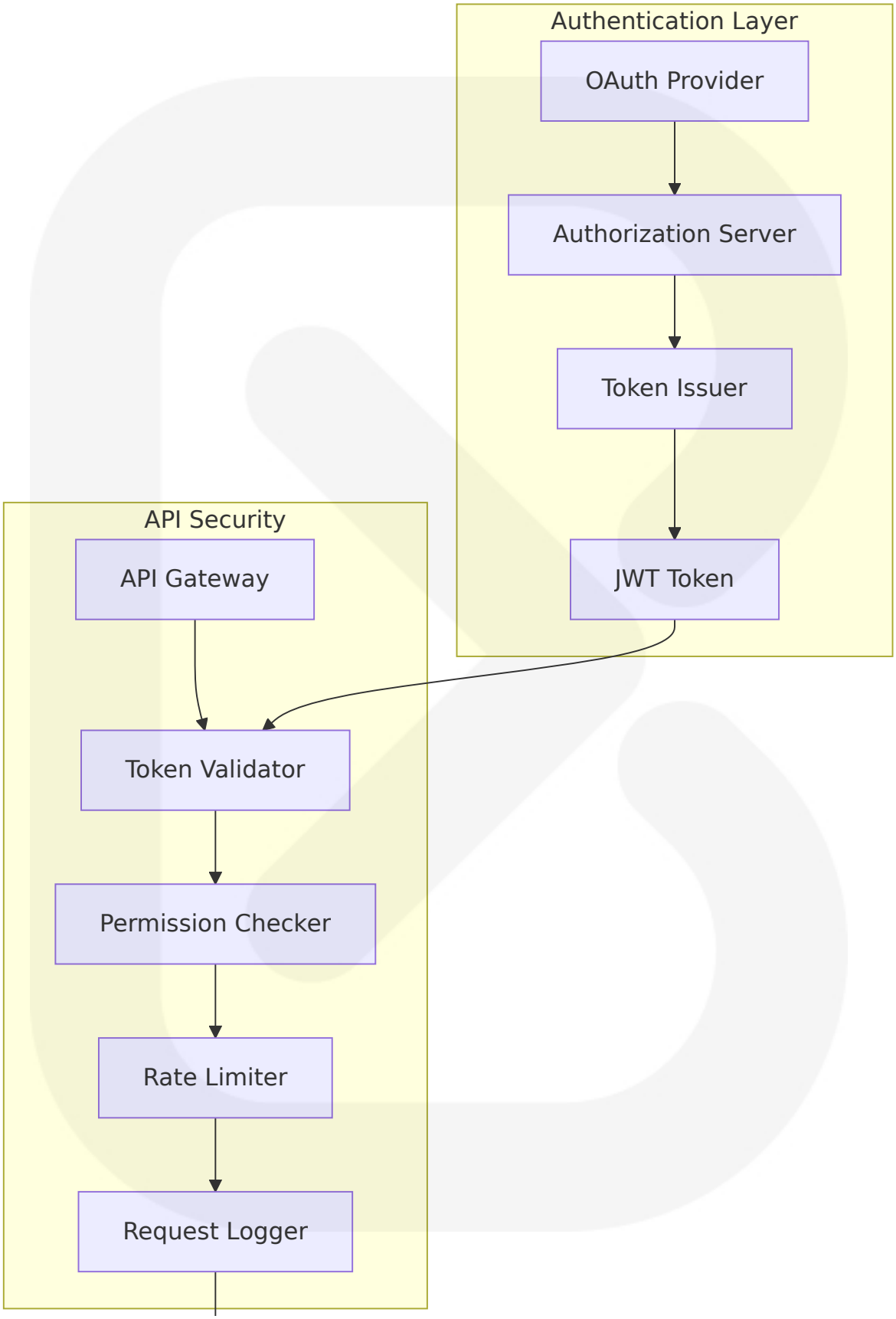
- **Last-Write-Wins**: For user preferences and settings

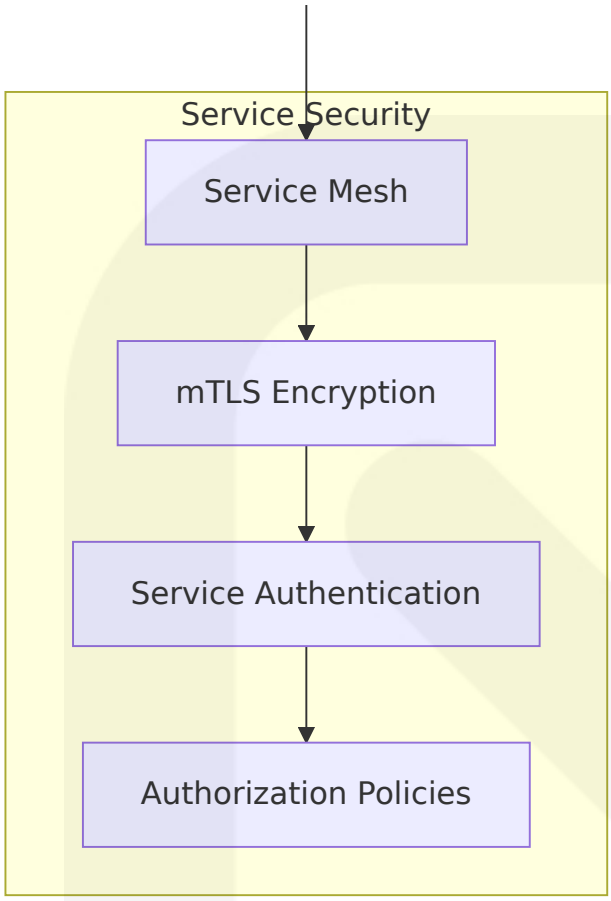
- **Merge Strategies**: For portfolio updates and position changes
- **User Intervention**: For critical trading decisions
- **Rollback Procedures**: For system errors and data corruption

6.3.4 Security and Authentication Integration

OAuth 2.0 + JWT Implementation:
The system implements a hybrid authentication approach combining OAuth 2.0 for initial authentication with JWT tokens for subsequent API access, providing both security and performance optimization.

Security Architecture:


Authentication Layer
OAuth Provider
Authorization Server
Token Issuer
JWT Token
API Security
API Gateway
Token Validator
Permission Checker
Rate Limiter
Request Logger



Security Measures:

- **Token Rotation**: Automatic refresh token rotation for enhanced security
- **API Rate Limiting**: Per-user and per-endpoint request throttling
- **Request Signing**: HMAC-based request integrity verification
- **Audit Logging**: Comprehensive security event logging and monitoring

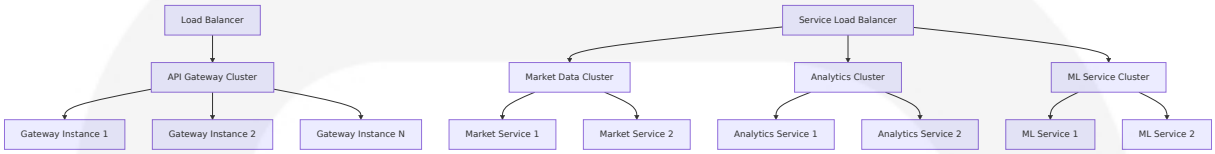
6.4 SCALABILITY AND PERFORMANCE OPTIMIZATION

6.4.1 Horizontal Scaling Architecture

Service Scaling Strategy:
Each microservice is designed for independent horizontal scaling based on

load patterns and resource requirements. The system employs container orchestration for dynamic scaling and load distribution.

Load Balancing Configuration:



Auto-Scaling Triggers:

Metric	Threshold	Scaling Action	Cool-down Period
CPU Utilization	>70%	Scale up	5 minutes
Memory Usage	>80%	Scale up	3 minutes
Request Queue Length	>100	Scale up	2 minutes
Response Time	>2 seconds	Scale up	5 minutes

6.4.2 Caching and Performance Optimization

Multi-Level Caching Strategy:

Cache Configuration:

- **L1 Cache (Application)**: In-memory caching with 1-5 second TTL
- **L2 Cache (Redis)**: Distributed caching with 1-60 minute TTL
- **L3 Cache (CDN)**: Static content caching with 24-hour TTL
- **Database Cache**: Query result caching with intelligent invalidation

Performance Monitoring:

- **Response Time Tracking**: P95, P99 latency monitoring
- **Throughput Measurement**: Requests per second by endpoint
- **Error Rate Monitoring**: 4xx/5xx error tracking and alerting

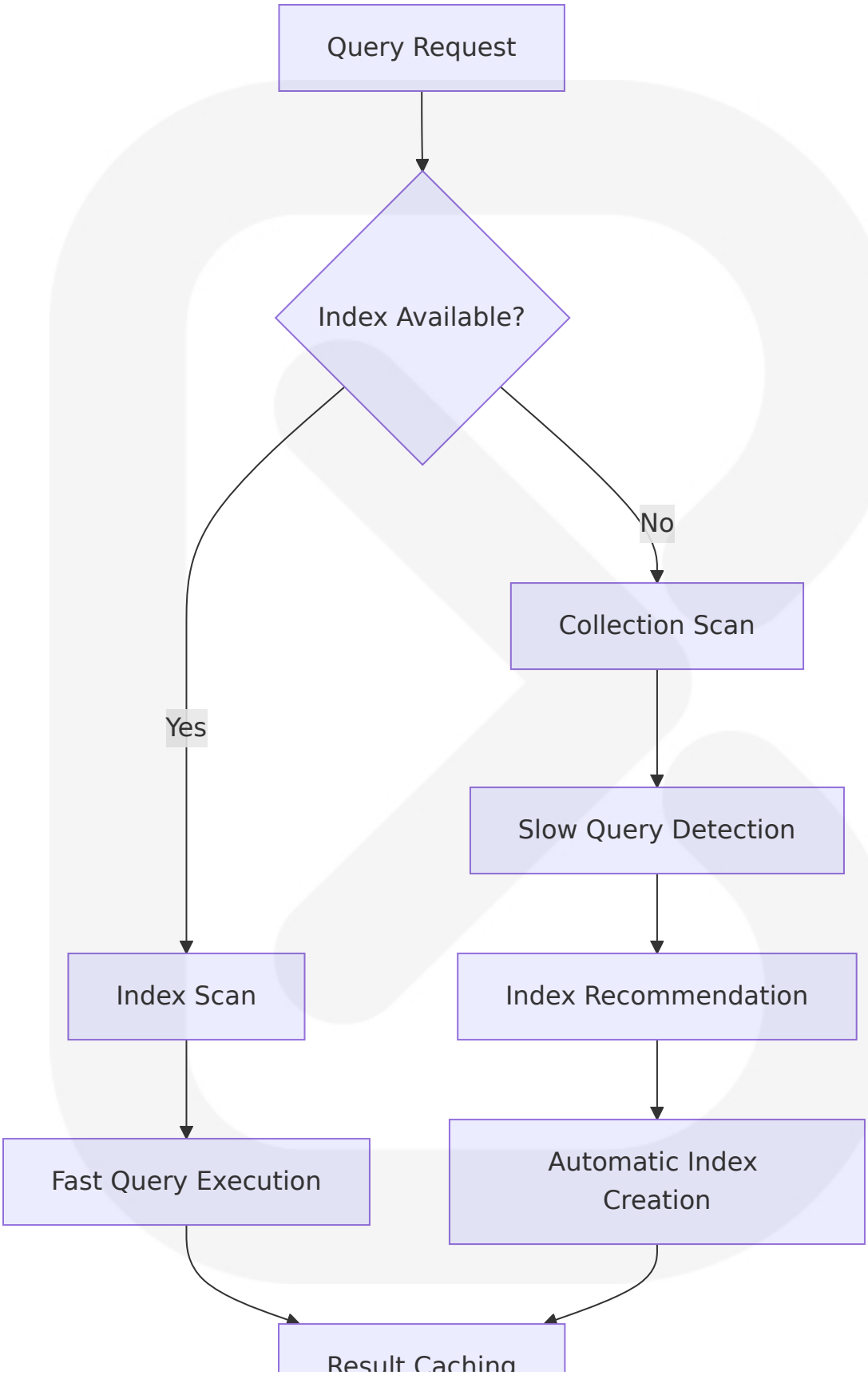
- **Resource Utilization**: CPU, memory, and network usage metrics

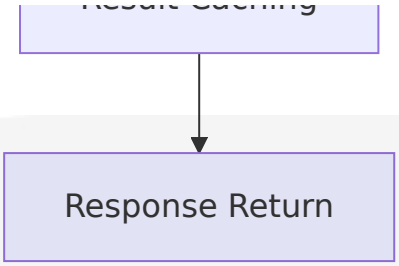
6.4.3 Database Performance Optimization

MongoDB Optimization Strategies:

- **Index Optimization**: Compound indexes for complex queries
- **Sharding Strategy**: Horizontal partitioning by currency pair and time
- **Read Replicas**: Separate read and write operations
- **Connection Pooling**: Efficient connection management

Query Optimization Patterns:


Query Request
Index Available?
No
Yes
Collection Scan
Slow Query Detection
Index Scan
Index Recommendation
Fast Query Execution
Automatic Index Creation
Result Caching



Data Archival Strategy:

- **Hot Data**: Recent 30 days in primary cluster
- **Warm Data**: 30-365 days in secondary cluster
- **Cold Data**: >365 days in archival storage
- **Automated Migration**: Time-based data lifecycle management

This comprehensive system components design provides a robust, scalable architecture for the forex trading application, leveraging the latest technologies including Expo SDK 54 with React Native 0.81, Flask-based microservices, and advanced machine learning capabilities. The design ensures high performance, reliability, and maintainability while supporting real-time trading operations and complex analytical workloads.

6.1 CORE SERVICES ARCHITECTURE

6.1.1 SERVICE COMPONENTS

6.1.1.1 Service Boundaries and Responsibilities

The forex trading application employs a microservices architecture where an application is divided into smaller sub-parts that are independent of each other i.e., they are independently deployable and scalable. Microservices are a software architectural style where a large application is broken down into smaller, independent services, with each service designed to serve a specific business function.

Service Name	Primary Responsibility	Business Domain	Technology Stack
API Gateway Service	Request routing, authentication, rate limiting	Cross-cutting concerns	Flask 3.1.2, Redis
Market Data Service	Real-time data acquisition and processing	Market data management	Python, TradingView APIs
Analytics Service	Technical analysis and indicator calculations	Market analysis	Python, TA-Lib, NumPy
ML Prediction Service	Machine learning model inference	Predictive analytics	Python, scikit-learn, TensorFlow

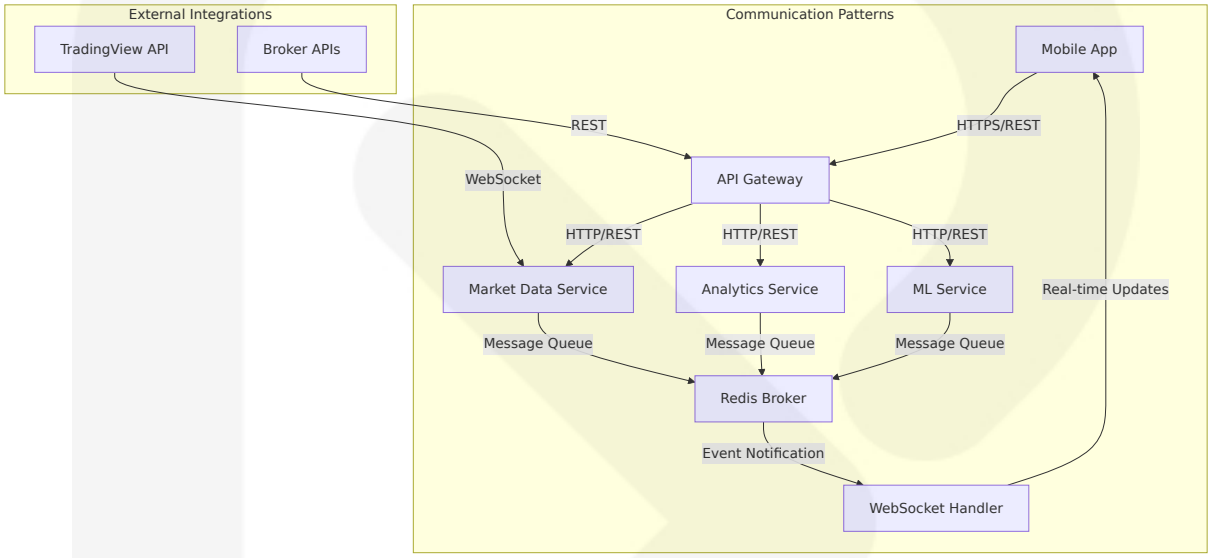
Service Decomposition Strategy:
The system follows domain-driven design principles with clear service boundaries based on business capabilities. The application will get alerts from TradingView regarding the signal(s) of your choice of strategy, with technical analysis indicators including MA, EMA, MACD, Supertrend for signal generation.

Service Autonomy:
Each microservice maintains its own data store and business logic, ensuring services are highly maintainable, testable, loosely coupled, independently deployable, and precisely focused. This approach enables teams to work independently while maintaining system cohesion.

6.1.1.2 Inter-service Communication Patterns

Synchronous Communication:
Services communicate using HTTP requests and RESTful APIs, with the API Gateway serving as the primary orchestration layer. Flask app receiving alerts from TradingView and automatically sends a POST order to an integrated exchange API demonstrates the request-response pattern implementation.

Asynchronous Communication:
There are various approaches to facilitate communication between microservices. In this case, the communication is synchronous, but asynchronous methods can also be employed as an alternative. The system implements message queuing using Redis for background processing and event-driven updates.



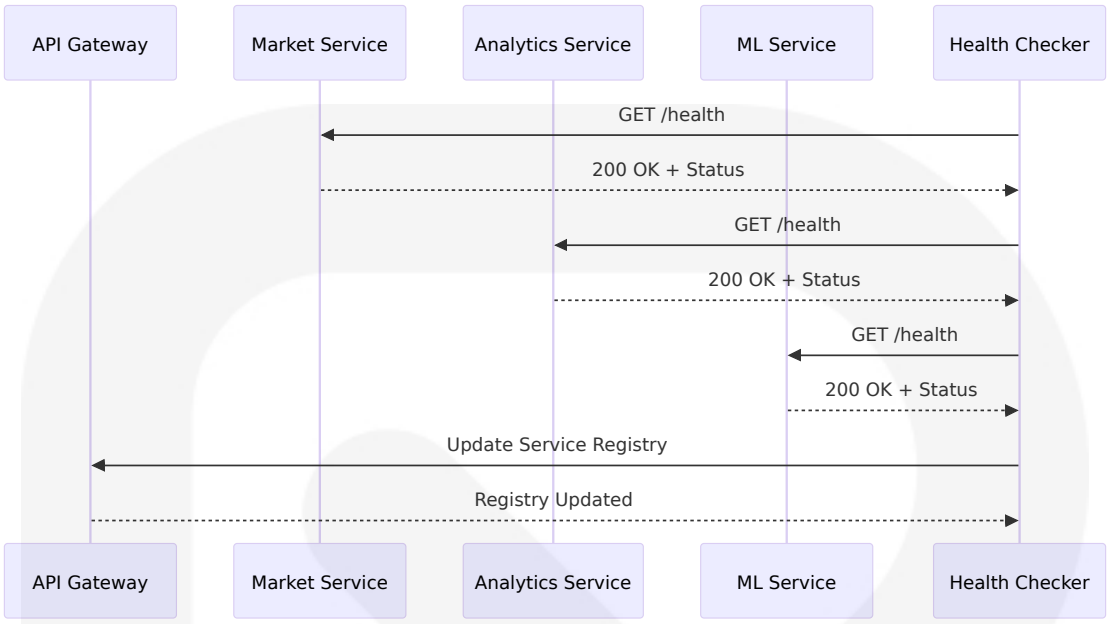
Communication Protocols:

- **REST APIs**: Primary synchronous communication protocol
- **WebSocket**: Real-time data streaming for market updates
- **Message Queues**: Asynchronous processing via Redis pub/sub
- **Event Streaming**: Real-time notifications and alerts

6.1.1.3 Service Discovery Mechanisms

Static Service Discovery:
The system employs configuration-based service discovery with environment variables and configuration files managing service endpoints. Each service registers its location and health status with the API Gateway during startup.

Health Check Implementation:



Service Registry Pattern:

- **Centralized Registry**: API Gateway maintains service locations
- **Health Monitoring**: Periodic health checks every 30 seconds
- **Load Balancing**: Round-robin distribution across healthy instances
- **Failover Logic**: Automatic routing to backup instances

6.1.1.4 Load Balancing Strategy

Application-Level Load Balancing:

The API Gateway implements intelligent load balancing with multiple algorithms based on service characteristics and current load patterns.

Load Balancing Algorithm	Use Case	Implementation
Round Robin	General API requests	Default for stateless operations
Least Connections	ML model inference	Resource-intensive operations
Weighted Round Robin	Service capacity differences	Based on service performance metrics

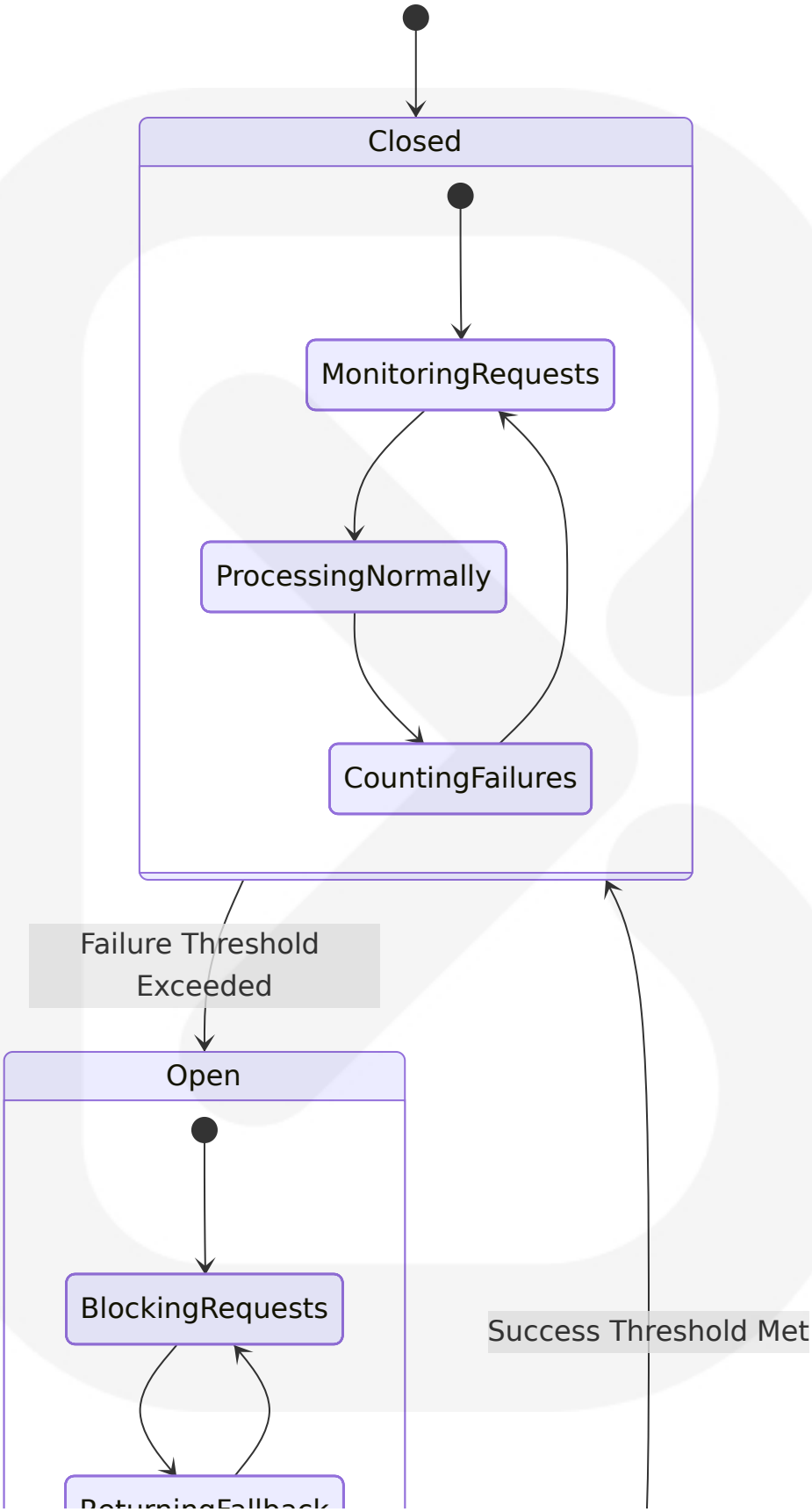
Load Balancing Configuration:

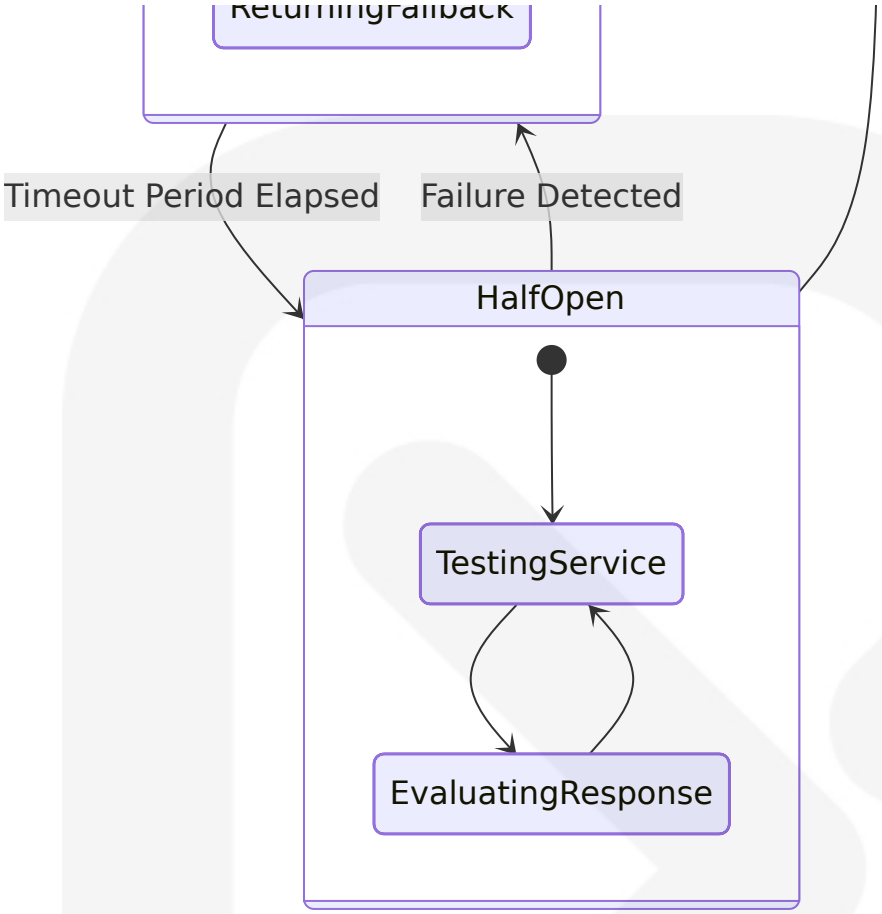
- **Health-based Routing**: Exclude unhealthy instances automatically
- **Geographic Distribution**: Route based on data center proximity
- **Resource-aware Balancing**: Consider CPU and memory utilization
- **Session Affinity**: Maintain user session consistency where required

6.1.1.5 Circuit Breaker Patterns

The Circuit Breaker pattern serves a critical role in inter-service communication, aiming to prevent the worsening of a service's performance. When one API attempts to communicate synchronously with another microservice, the Circuit Breaker acts akin to an electrical circuit: if a failure is detected, it "opens," temporarily halting access to the service for a predefined duration.

Circuit Breaker Implementation:
Pybreaker is a Python library that simplifies the implementation of the Circuit Breaker pattern for managing inter-service communication in distributed systems. The system implements circuit breakers for all external service calls.


Closed
MonitoringRequests
ProcessingNormally
CountingFailures
Failure Threshold Exceeded
Open
BlockingRequests
Success Threshold Met



Circuit Breaker Configuration:

- **Failure Threshold**: 5 consecutive failures trigger circuit opening
- **Reset Timeout**: 60 seconds before attempting service recovery
- **Success Threshold**: 3 successful requests required to close circuit
- **Fallback Mechanisms**: Cached data or alternative service routing

6.1.1.6 Retry and Fallback Mechanisms

Exponential Backoff Retry:
By "tripping" and breaking the circuit when repeated failures occur, the circuit breaker allows the system to respond quickly to issues, limit retry attempts, and improve overall stability.

Retry Strategy Implementation:

- **Initial Delay**: 100ms base delay for first retry

- **Exponential Multiplier**: 2x delay increase per retry attempt
- **Maximum Retries**: 3 attempts before circuit breaker activation
- **Jitter**: Random delay variation to prevent thundering herd

Fallback Mechanisms:

Service	Primary Fallback	Secondary Fallback	Degraded Mode
Market Data	Cached data (5min TTL)	Alternative data source	Historical data only
Analytics	Simplified indicators	Basic trend analysis	Manual analysis mode
ML Predictions	Previous model results	Statistical averages	Confidence warnings

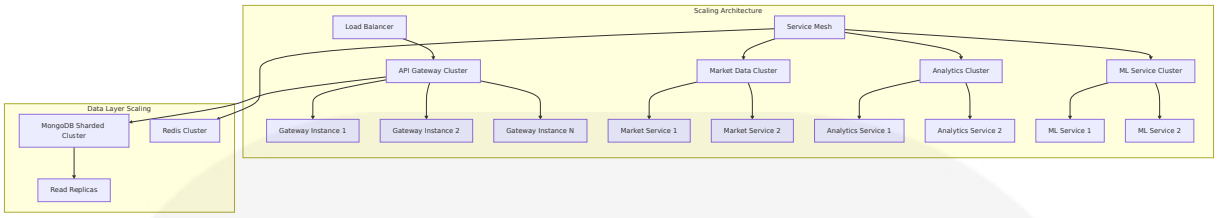
Graceful Degradation:
Even when external dependencies are down, the circuit breaker allows the system to continue running by failing fast. Users experience quick failure responses rather than waiting for timeouts.

6.1.2 SCALABILITY DESIGN

6.1.2.1 Horizontal/Vertical Scaling Approach

Horizontal Scaling Strategy:
Microservices are small, independent applications that perform specific tasks, allowing for flexible deployment and scaling. This modular approach to software design loosens the coupling between components, enhancing flexibility and manageability throughout development.

Service-Specific Scaling Patterns:



Scaling Characteristics by Service:

- **API Gateway**: Stateless horizontal scaling with session affinity
- **Market Data Service**: Vertical scaling for memory-intensive operations
- **Analytics Service**: Horizontal scaling with data partitioning
- **ML Service**: GPU-based vertical scaling with model parallelization

6.1.2.2 Auto-scaling Triggers and Rules

Metric-Based Auto-scaling:

The system implements comprehensive auto-scaling based on multiple performance indicators and business metrics.

Metric Type	Threshold	Scale-up Action	Scale-down Action	Cool-down Period
CPU Utilization	>70%	Add 1 instance	Remove 1 instance	5 minutes
Memory Usage	>80%	Add 1 instance	Remove 1 instance	3 minutes
Request Queue Length	>100 requests	Add 2 instances	Remove 1 instance	2 minutes
Response Time	>2 seconds	Add 1 instance	Remove 1 instance	5 minutes

Business Metric Scaling:

- **Market Volatility**: Scale analytics services during high volatility periods
- **User Activity**: Scale based on active trading sessions

- **Data Volume**: Scale market data services based on symbol subscriptions
- **Prediction Requests**: Scale ML services based on model inference demand

6.1.2.3 Resource Allocation Strategy

Container Resource Management:
Each microservice runs in containerized environments with defined resource limits and requests to ensure optimal resource utilization and prevent resource contention.

Resource Allocation Matrix:

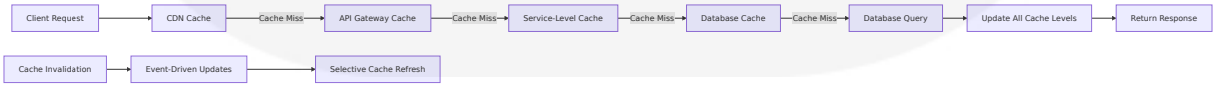
- **API Gateway**: 512MB RAM, 0.5 CPU cores per instance
- **Market Data Service**: 1GB RAM, 1 CPU core per instance
- **Analytics Service**: 2GB RAM, 1.5 CPU cores per instance
- **ML Service**: 4GB RAM, 2 CPU cores, 1 GPU per instance

Resource Optimization Techniques:

- **Memory Pooling**: Shared memory pools for similar operations
- **Connection Pooling**: Database connection reuse across requests
- **Caching Strategies**: Multi-level caching to reduce computation load
- **Lazy Loading**: On-demand resource allocation for non-critical components

6.1.2.4 Performance Optimization Techniques

Caching Strategy:
Multi-level caching implementation reduces latency and improves system responsiveness across all service layers.



Performance Optimization Strategies:

- **Database Optimization**: Index optimization and query performance tuning
- **Asynchronous Processing**: Background job processing for non-critical operations
- **Data Compression**: Reduce network bandwidth usage
- **Connection Multiplexing**: HTTP/2 and connection reuse

6.1.2.5 Capacity Planning Guidelines

Capacity Planning Methodology:

- **Baseline Metrics**: Establish performance baselines under normal load
- **Growth Projections**: Plan for 3x user growth over 12 months
- **Peak Load Planning**: Handle 10x normal load during market events
- **Resource Buffers**: Maintain 20% resource headroom for unexpected spikes

Capacity Monitoring:

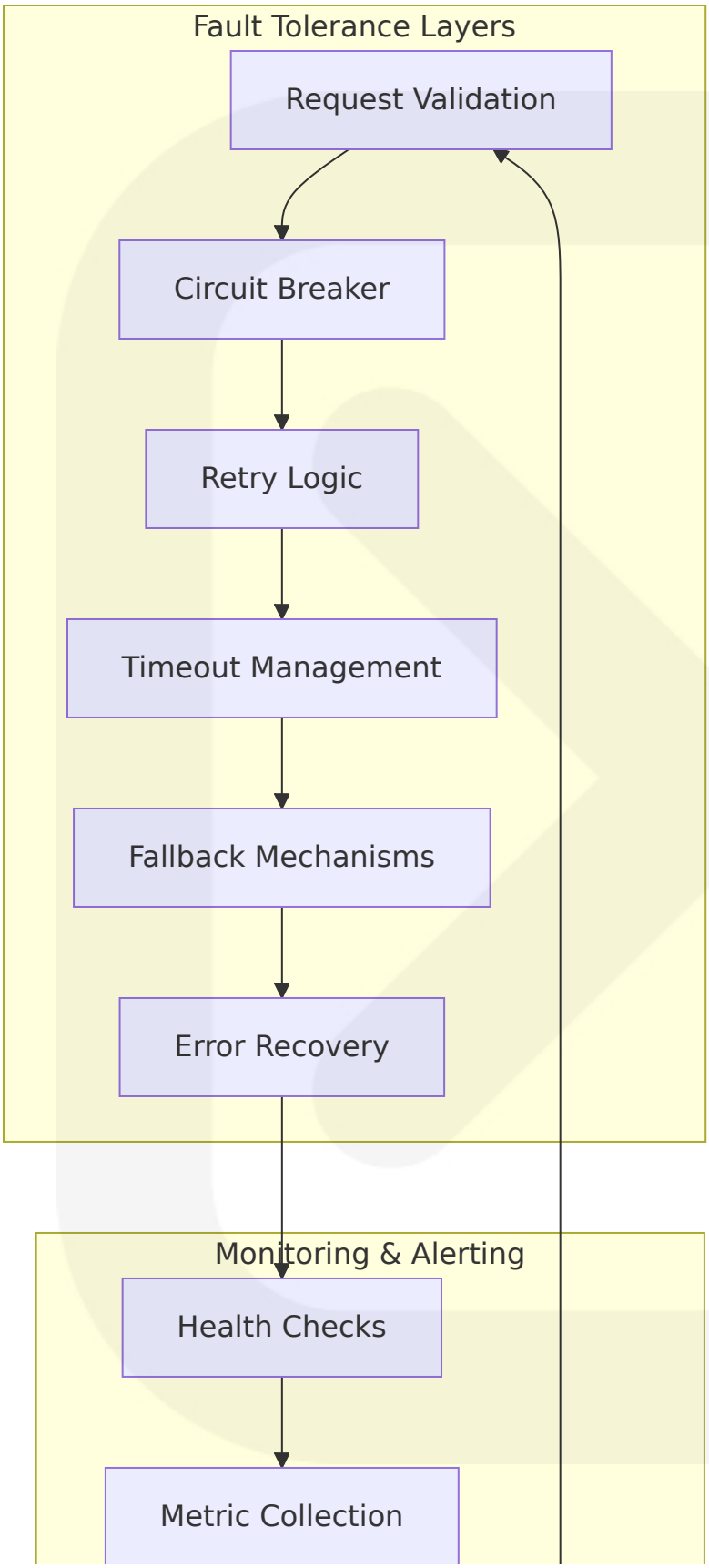
- **Real-time Metrics**: CPU, memory, network, and disk utilization
- **Business Metrics**: Active users, trading volume, API requests
- **Predictive Analytics**: Forecast resource needs based on historical patterns
- **Alert Thresholds**: Proactive scaling before resource exhaustion

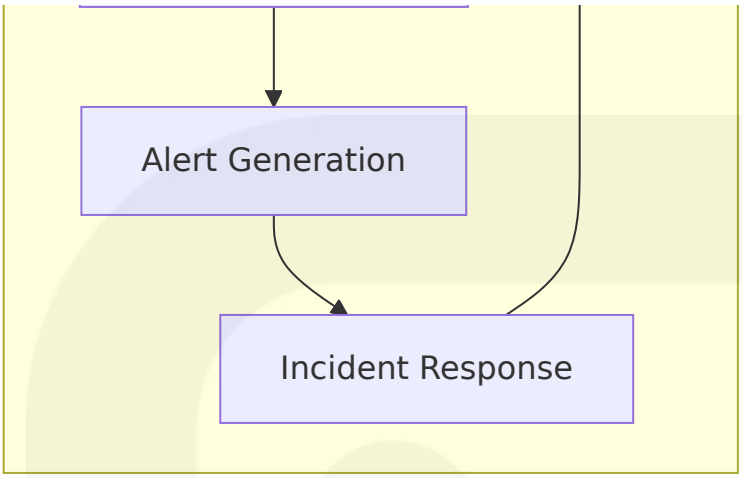
6.1.3 RESILIENCE PATTERNS

6.1.3.1 Fault Tolerance Mechanisms

Multi-layered Fault Tolerance:
The circuit breaker prevents cascading failures by stopping one failing service from overloading others, allows requests to fail quickly improving user experience, and helps the system self-heal by periodically reattempting connections to downed services.

Fault Tolerance Implementation:


Fault Tolerance Layers
Request Validation
Circuit Breaker
Retry Logic
Timeout Management
Fallback Mechanisms
Error Recovery
Monitoring & Alerting
Health Checks
Metric Collection



Fault Detection and Recovery:

- **Health Check Endpoints**: Comprehensive service health monitoring
- **Dependency Checks**: Verify external service availability
- **Resource Monitoring**: Track system resource utilization
- **Automated Recovery**: Self-healing mechanisms for common failures

6.1.3.2 Disaster Recovery Procedures

Recovery Time and Point Objectives:
The system implements comprehensive disaster recovery procedures with defined RTO (Recovery Time Objective) and RPO (Recovery Point Objective) targets.

Service Tier	RTO Target	RPO Target	Recovery Strategy
Critical Services	<1 hour	<15 minutes	Hot standby with real-time replication
Important Services	<4 hours	<1 hour	Warm standby with periodic backups
Supporting Services	<24 hours	<4 hours	Cold standby with daily backups

Disaster Recovery Workflow:

1. **Incident Detection**: Automated monitoring identifies system failures

2. **Impact Assessment**: Evaluate scope and severity of the incident
3. **Recovery Activation**: Initiate appropriate recovery procedures
4. **Service Restoration**: Restore services in priority order
5. **Validation Testing**: Verify system functionality post-recovery
6. **Post-incident Review**: Analyze incident and improve procedures

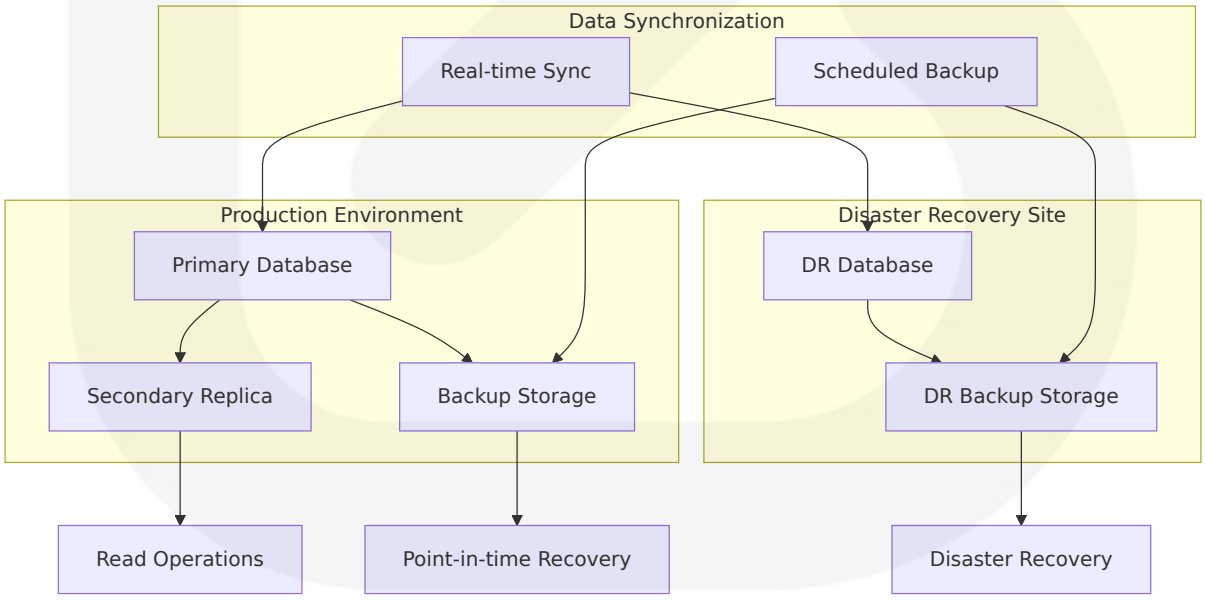
6.1.3.3 Data Redundancy Approach

Multi-level Data Redundancy:
The system implements comprehensive data redundancy across multiple layers to ensure data availability and consistency.

Data Replication Strategy:

- **Primary-Secondary Replication**: Real-time data replication for critical data
- **Multi-region Backup**: Geographic distribution of data copies
- **Point-in-time Recovery**: Granular recovery capabilities
- **Cross-service Data Sync**: Consistent data across service boundaries

Backup and Recovery Architecture:



6.1.3.4 Failover Configurations

Automatic Failover Mechanisms:
The system implements intelligent failover configurations that automatically redirect traffic from failed services to healthy alternatives.

Failover Strategies:

- **Active-Passive**: Primary service with standby backup
- **Active-Active**: Load distribution across multiple active instances
- **Geographic Failover**: Cross-region failover for disaster scenarios
- **Service Mesh Failover**: Intelligent routing based on service health

Failover Decision Matrix:

Failure Type	Detection Time	Failover Time	Recovery Action
Service Crash	<30 seconds	<1 minute	Restart service instance
Database Failure	<1 minute	<2 minutes	Switch to replica database
Network Partition	<2 minutes	<3 minutes	Route via alternative path
Data Center Outage	<5 minutes	<10 minutes	Activate DR site

6.1.3.5 Service Degradation Policies

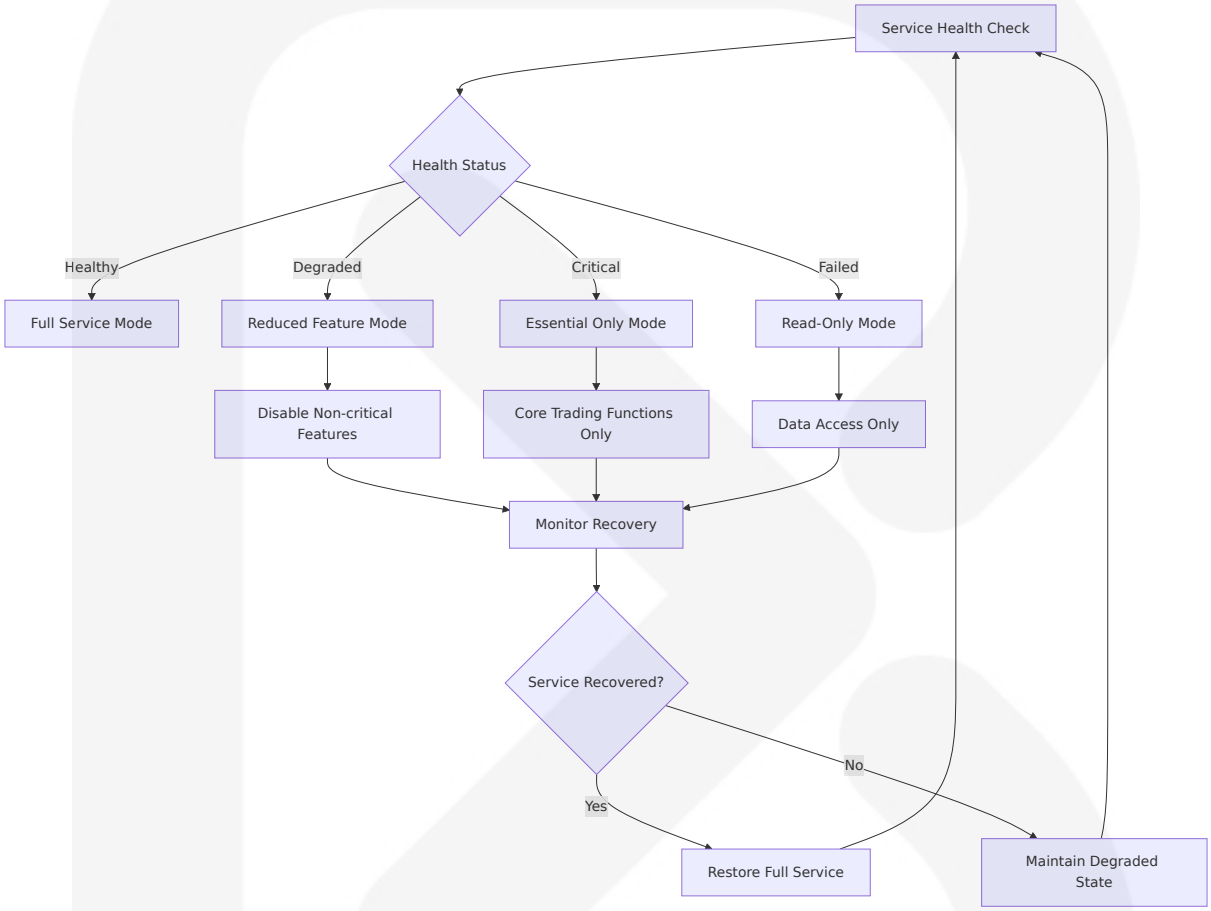
Graceful Degradation Framework:
This ensures other parts of our system remain responsive, providing a more reliable overall experience through intelligent service degradation policies.

Degradation Levels:

1. **Full Service**: All features available with optimal performance
2. **Reduced Features**: Non-critical features disabled, core functionality maintained

3. **Essential Only**: Only critical trading functions available
4. **Read-Only Mode**: Data access only, no trading operations
5. **Maintenance Mode**: System unavailable with status updates

Service Degradation Rules:



User Communication Strategy:

- **Status Page**: Real-time system status updates
- **In-app Notifications**: Service degradation alerts
- **Email Notifications**: Critical system updates
- **API Response Headers**: Service status indicators

The Core Services Architecture provides a robust, scalable foundation for the forex trading application, implementing proven microservices patterns with React Native 0.81 with Expo SDK 54 on the frontend and Flask-based microservices on the backend. The architecture ensures high availability,

fault tolerance, and optimal performance through comprehensive resilience patterns and intelligent scaling mechanisms.

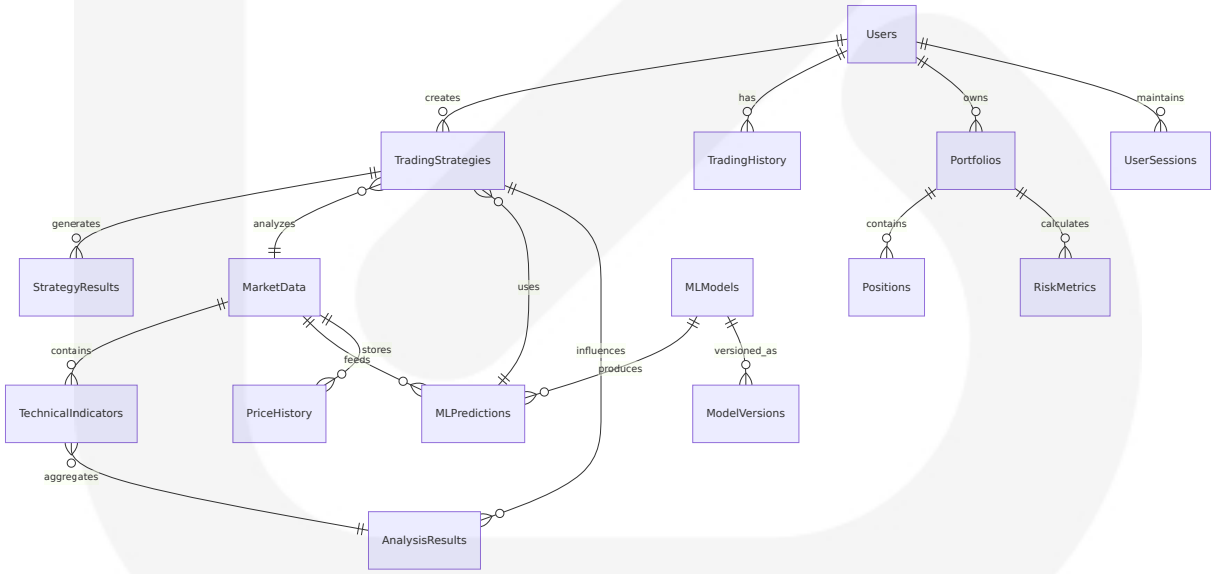
6.2 DATABASE DESIGN

6.2.1 SCHEMA DESIGN

6.2.1.1 Entity Relationships

The forex trading application requires a comprehensive database design to handle real-time market data, user management, trading strategies, and machine learning model artifacts. When you design your data model in MongoDB, consider the structure of your documents and the ways your application uses data from related entities. Embed related data within a single document.

Core Entity Relationships:



Entity Relationship Matrix:

Primary Entity	Related Entity	Relationship Type	Cardinality
Users	TradingStrategies	One-to-Many	1:N
Users	Portfolios	One-to-Many	1:N
MarketData	TechnicalIndicators	One-to-Many	1:N
TradingStrategies	MLPredictions	Many-to-One	N:1

6.2.1.2 Data Models and Structures

MongoDB Time Series Collections for Market Data:

Compared to normal collections, storing time series data in time series collections improves query efficiency and reduces the disk usage for time series data and secondary indexes. Time series collections use an underlying columnar storage format and store data in time-order.

Market Data Collection (Time Series):

```
// Time Series Collection Configuration
db.createCollection("market_data", {
  timeseries: {
    timeField: "timestamp",
    metaField: "metadata",
    granularity: "seconds"
  },
  expireAfterSeconds: 31536000 // 1 year retention
})

// Document Structure
{
  _id: ObjectId("..."),
  timestamp: ISODate("2025-01-15T10:30:00.000Z"),
  metadata: {
    symbol: "EURUSD",
    source: "tradingview",
```

```
    timeframe: "1m"
  },
  open: 1.0845,
  high: 1.0847,
  low: 1.0843,
  close: 1.0846,
  volume: 125000,
  spread: 0.0002
}
```

Users Collection:

```
{
  _id: ObjectId("..."),
  email: "trader@example.com",
  password_hash: "bcrypt_hash",
  profile: {
    first_name: "John",
    last_name: "Doe",
    timezone: "UTC",
    preferred_currency: "USD"
  },
  preferences: {
    risk_tolerance: "moderate",
    notification_settings: {
      email: true,
      push: true,
      trading_alerts: true
    },
    default_position_size: 0.01
  },
  subscription: {
    tier: "premium",
    expires_at: ISODate("2025-12-31T23:59:59.000Z")
  },
  created_at: ISODate("2025-01-01T00:00:00.000Z"),
  last_login: ISODate("2025-01-15T10:30:00.000Z"),
  is_active: true
}
```

Trading Strategies Collection:

```
{
  _id: ObjectId("..."),
  user_id: ObjectId("..."),
  strategy_name: "EURUSD Scalping Strategy",
  currency_pair: "EURUSD",
  timeframe: "5m",
  parameters: {
    entry_conditions: {
      rsi_oversold: 30,
      rsi_overbought: 70,
      macd_signal: "bullish_crossover"
    },
    exit_conditions: {
      take_profit: 0.0020,
      stop_loss: 0.0010,
      trailing_stop: true
    }
  },
  technical_indicators: [
    {
      name: "RSI",
      period: 14,
      values: [65.2, 67.1, 69.3]
    },
    {
      name: "MACD",
      fast_period: 12,
      slow_period: 26,
      signal_period: 9
    }
  ],
  ml_predictions: {
    model_id: ObjectId("..."),
    confidence_score: 0.78,
    prediction: "BUY",
    probability: {
      buy: 0.78,
      sell: 0.15,
      hold: 0.07
    }
  },
  risk_assessment: {
    risk_score: 3.2,
```

```
    max_drawdown: 0.05,
    position_size: 0.02,
    risk_reward_ratio: 2.0
  },
  backtest_results: {
    total_trades: 150,
    winning_trades: 105,
    win_rate: 0.70,
    profit_factor: 1.85,
    max_drawdown: 0.08
  },
  status: "active",
  created_at: ISODate("2025-01-15T10:30:00.000Z"),
  updated_at: ISODate("2025-01-15T10:30:00.000Z")
}
```

6.2.1.3 Indexing Strategy

Time Series Collection Indexing:

MongoDB automatically creates a compound index on both the metaField and timeField of a time series collection. MongoDB 6.3 and later automatically creates a compound index on the time and metadata fields for new time series collections.

Index Configuration Matrix:

Collection	Index Type	Fields	Purpose	Performance Impact
market_data	Compound (Auto)	metadata.symbol, timestamp	Time series queries	High
market_data	Single	metadata.timeframe	Timeframe filtering	Medium
users	Unique	email	User authentication	High
trading_strategies	Compound	user_id, currency_pair	User strategy queries	High

Custom Index Definitions:

```
// Market Data Indexes (Time Series)
db.market_data.createIndex(
  { "metadata.symbol": 1, "timestamp": -1 },
  { name: "symbol_time_desc" }
)

// Users Collection Indexes
db.users.createIndex(
  { "email": 1 },
  { unique: true, name: "email_unique" }
)

db.users.createIndex(
  { "created_at": 1 },
  { name: "created_at_asc" }
)

// Trading Strategies Indexes
db.trading_strategies.createIndex(
  { "user_id": 1, "currency_pair": 1 },
  { name: "user_pair_compound" }
)

db.trading_strategies.createIndex(
  { "status": 1, "created_at": -1 },
  { name: "status_created_desc" }
)

// ML Predictions Indexes
db.ml_predictions.createIndex(
  { "model_id": 1, "created_at": -1 },
  { name: "model_created_desc" }
)
```

6.2.1.4 Partitioning Approach

Time-Based Partitioning Strategy:

The bucketing design pattern is one MongoDB design pattern that groups raw data from multiple documents into one document rather than keeping separate documents for each and every raw piece of data. Therefore, we see performance benefits in terms of index size savings and read/write speed.

Partitioning Configuration:

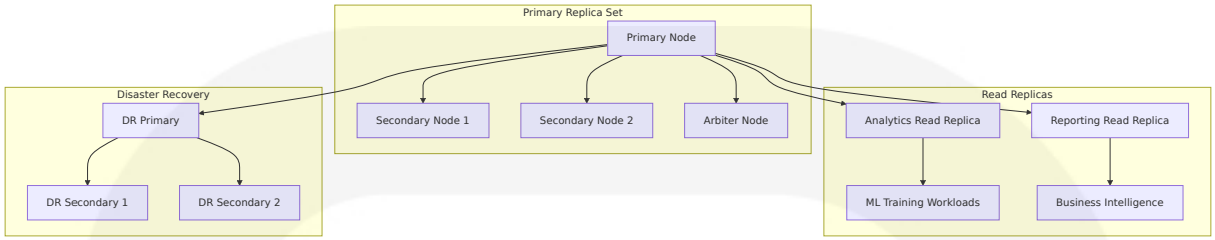
Data Type	Partitioning Method	Partition Key	Retention Policy
Real-time Market Data	Time Series Bucketing	metadata.symbol + timestamp	1 year
Historical Market Data	Date-based Sharding	timestamp (monthly)	5 years
User Data	Hash-based Sharding	user_id	Permanent
Trading Strategies	Compound Sharding	user_id + currency_pair	2 years

Sharding Configuration:

```
// Enable sharding for trading_strategies collection
sh.enableSharding("forex_trading")

// Shard trading_strategies by user_id and currency_pair
sh.shardCollection(
  "forex_trading.trading_strategies",
  { "user_id": 1, "currency_pair": 1 }
)

// Shard market_data time series by metadata.symbol
sh.shardCollection(
  "forex_trading.market_data",
  { "metadata.symbol": 1 }
)
```

6.2.1.5 Replication Configuration

MongoDB Replica Set Architecture:



Replication Configuration:

```
// Replica Set Configuration
rs.initiate({
  _id: "forex-rs",
  members: [
    { _id: 0, host: "primary:27017", priority: 2 },
    { _id: 1, host: "secondary1:27017", priority: 1 },
    { _id: 2, host: "secondary2:27017", priority: 1 },
    { _id: 3, host: "arbiter:27017", arbiterOnly: true }
  ]
})

// Read Preference Configuration
db.getMongo().setReadPref("secondaryPreferred", [
  { "datacenter": "analytics" }
])
```

6.2.1.6 Backup Architecture

Multi-Tier Backup Strategy:

Backup Type	Frequency	Retention	Storage Location	Recovery Time
Point-in-Time	Continuous	7 days	Local SSD	<5 minutes
Daily Snapshot	Daily	30 days	Cloud Storage	<30 minutes
Weekly Archive	Weekly	1 year	Cold Storage	<4 hours

Backup Type	Frequency	Retention	Storage Location	Recovery Time
Monthly Archive	Monthly	5 years	Glacier Storage	<24 hours

Backup Configuration:

```
// MongoDB Atlas Backup Configuration
{
  "backupPolicy": {
    "pointInTimeRecovery": true,
    "snapshotRetention": {
      "daily": 30,
      "weekly": 12,
      "monthly": 60
    },
    "continuousBackup": {
      "enabled": true,
      "retentionDays": 7
    }
  }
}
```

6.2.2 DATA MANAGEMENT

6.2.2.1 Migration Procedures

Database Migration Strategy:

The system implements a comprehensive migration framework supporting both schema evolution and data transformation requirements for the forex trading application.

Migration Framework:

```
// Migration Script Template
const migration = {
  version: "2025.01.15.001",
```

```
  description: "Add ML prediction confidence thresholds",
  up: async function(db) {
    // Forward migration
    await db.collection('trading_strategies').updateMany(
      { "ml_predictions.confidence_threshold": { $exists: false } },
      { $set: { "ml_predictions.confidence_threshold": 0.6 } }
    )
  },
  down: async function(db) {
    // Rollback migration
    await db.collection('trading_strategies').updateMany(
      {},
      { $unset: { "ml_predictions.confidence_threshold": "" } }
    )
  }
}
```

Migration Execution Process:

Phase	Action	Validation	Rollback Plan
Pre-migration	Schema validation	Data integrity checks	Snapshot creation
Migration	Execute transformation	Progress monitoring	Automatic rollback triggers
Post-migration	Performance testing	Data consistency verification	Manual rollback procedures
Cleanup	Remove deprecated fields	Index optimization	Archive old schemas

6.2.2.2 Versioning Strategy

Schema Versioning Framework:

The application implements semantic versioning for database schemas with backward compatibility support for gradual migration of client applications.

Version Control Structure:

```
// Schema Version Document
{
  _id: "schema_version",
  current_version: "2.1.0",
  supported_versions: ["2.0.0", "2.1.0"],
  migration_history: [
    {
      version: "2.1.0",
      applied_at: ISODate("2025-01-15T10:30:00.000Z"),
      description: "Added ML model confidence scoring",
      rollback_available: true
    }
  ],
  compatibility_matrix: {
    "mobile_app": {
      "1.0.0": ["2.0.0"],
      "1.1.0": ["2.0.0", "2.1.0"]
    }
  }
}
```

API Versioning Strategy:

- **Header-based versioning**: `API-Version: 2.1.0`
- **Backward compatibility**: Support for 2 previous major versions
- **Deprecation timeline**: 6-month notice for breaking changes
- **Feature flags**: Gradual rollout of new schema features

6.2.2.3 Archival Policies

Data Lifecycle Management:

Time series data is almost always appended in comparison to updates or deletion. That means databases can have huge workloads, and even indexes may not be enough for optimization.

Archival Configuration:

Data Category	Hot Storage	Warm Storage	Cold Storage	Archive Storage
Real-time Market Data	7 days	30 days	1 year	5 years
Trading Strategies	90 days	1 year	2 years	Permanent
User Activity Logs	30 days	90 days	1 year	3 years
ML Model Artifacts	30 days	6 months	2 years	5 years

Automated Archival Process:

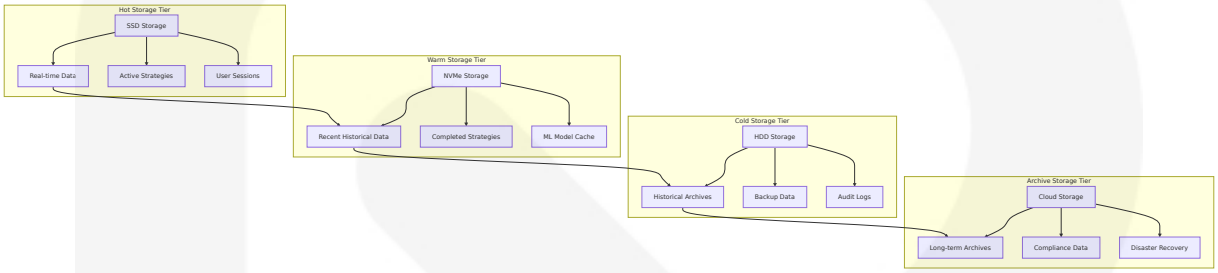
```
// TTL Index for Automatic Expiration
db.market_data.createIndex(
  { "timestamp": 1 },
  { expireAfterSeconds: 31536000 } // 1 year
)

// Archival Pipeline
const archivalPipeline = [
  {
    $match: {
      timestamp: {
        $lt: new Date(Date.now() - 90 * 24 * 60 * 60 * 1000) // 90 days a
      }
    }
  },
  {
    $out: {
      db: "forex_archive",
      coll: "market_data_archive"
    }
  }
]
```

6.2.2.4 Data Storage and Retrieval Mechanisms

Storage Optimization Strategy:

With MongoDB 8.0, Time Series Collections now directly write into a column-compressed format, reducing cache usage, lowering write I/O, improving insert performance and storage efficiency.

Storage Architecture:



Retrieval Optimization:

- **Query routing**: Automatic routing based on data age
- **Caching layers**: Multi-level caching with Redis
- **Compression**: Column-store compression for time series data
- **Indexing**: Optimized indexes for time-based queries

6.2.2.5 Caching Policies

Redis Caching Architecture:

Update prices of securities in the portfolio in real-time as the brokerage receives the latest prices from the exchanges.

Cache Configuration Matrix:

Data Type	Cache Strategy	TTL	Eviction Policy	Consistency Model
Real-time Prices	Write-through	5 seconds	LRU	Strong consistency
Technical Indicators	Cache-aside	1 minute	LFU	Eventual consistency

Data Type	Cache Strategy	TTL	Eviction Policy	Consistency Model
ML Predictions	Write-behind	15 minutes	TTL-based	Eventual consistency
User Sessions	Write-through	24 hours	TTL-based	Strong consistency

Redis Configuration:

```
// Cache-aside Pattern Implementation
const getCachedMarketData = async (symbol, timeframe) => {
  const cacheKey = `market:${symbol}:${timeframe}`

  // Check cache first
  let data = await redis.get(cacheKey)
  if (data) {
    return JSON.parse(data)
  }

  // Cache miss - fetch from database
  data = await db.collection('market_data').findOne({
    'metadata.symbol': symbol,
    'metadata.timeframe': timeframe
  }, { sort: { timestamp: -1 } })

  // Update cache
  await redis.setex(cacheKey, 60, JSON.stringify(data))
  return data
}

// Write-through Pattern for Real-time Updates
const updateMarketData = async (marketData) => {
  // Update database
  await db.collection('market_data').insertOne(marketData)

  // Update cache immediately
  const cacheKey = `market:${marketData.metadata.symbol}:${marketData.me
  await redis.setex(cacheKey, 5, JSON.stringify(marketData))

  // Publish to subscribers
```

```
    await redis.publish(`market_updates:${marketData.metadata.symbol}`, JS
}
```

6.2.3 COMPLIANCE CONSIDERATIONS

6.2.3.1 Data Retention Rules

Regulatory Compliance Framework:

The forex trading application must comply with multiple financial regulations including MiFID II, GDPR, and regional financial data protection requirements.

Retention Policy Matrix:

Data Category	Regulatory Requirement	Retention Period	Deletion Trigger	Compliance Standard
Trading Records	MiFID II	5 years	Automatic	Financial regulation
Personal Data	GDPR	User consent based	User request	Privacy regulation
Audit Logs	SOX Compliance	7 years	Automatic	Financial audit
ML Model Data	Internal Policy	2 years	Model retirement	Business requirement

Automated Retention Implementation:

```
// Retention Policy Configuration
const retentionPolicies = {
  trading_history: {
    retention_days: 1825, // 5 years
    compliance: "MiFID II",
    auto_delete: true
  },
  user_personal_data: {
```

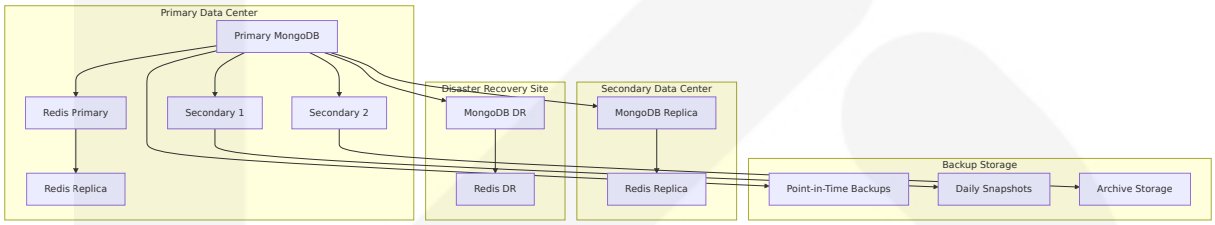
```
      retention_days: null, // User consent based
      compliance: "GDPR",
      auto_delete: false,
      deletion_triggers: ["user_request", "account_closure"]
    },
    audit_logs: {
      retention_days: 2555, // 7 years
      compliance: "SOX",
      auto_delete: true
    }
  }
```

6.2.3.2 Backup and Fault Tolerance Policies

High Availability Configuration:

The system implements comprehensive backup and fault tolerance mechanisms to ensure 99.9% uptime and data durability.

Fault Tolerance Architecture:



Recovery Procedures:

Failure Type	Detection Time	Recovery Time	Data Loss	Procedure
Primary Node Failure	<30 seconds	<2 minutes	None	Automatic failover
Data Center Outage	<2 minutes	<10 minutes	<1 minute	Geographic failover
Corruption/Disaster	<5 minutes	<4 hours	<15 minutes	Point-in-time recovery
Complete System Loss	<10 minutes	<24 hours	<1 hour	Disaster recovery site

6.2.3.3 Privacy Controls

GDPR Compliance Implementation:

The system implements comprehensive privacy controls including data minimization, purpose limitation, and user consent management.

Privacy Control Framework:

```
// User Consent Management
const privacyControls = {
  data_categories: {
    essential: {
      description: "Account and authentication data",
      legal_basis: "contract",
      retention: "account_lifetime",
      user_control: false
    },
    analytics: {
      description: "Trading behavior analysis",
      legal_basis: "consent",
      retention: "2_years",
      user_control: true
    },
    marketing: {
      description: "Marketing communications",
      legal_basis: "consent",
      retention: "until_withdrawn",
      user_control: true
    }
  },
  user_rights: {
    access: true,
    rectification: true,
    erasure: true,
    portability: true,
    restriction: true,
    objection: true
  }
}

// Data Anonymization Pipeline
```

```
const anonymizeUserData = async (userId) => {
  const anonymizedId = generateAnonymousId()

  // Replace personal identifiers
  await db.collection('trading_history').updateMany(
    { user_id: userId },
    {
      $set: { user_id: anonymizedId },
      $unset: {
        "personal_info": "",
        "contact_details": ""
      }
    }
  )
}
```

6.2.3.4 Audit Mechanisms

Comprehensive Audit Trail:

The system maintains detailed audit logs for all data access, modifications, and system events to support regulatory compliance and security monitoring.

Audit Log Structure:

```
// Audit Log Document
{
  _id: ObjectId("..."),
  timestamp: ISODate("2025-01-15T10:30:00.000Z"),
  event_type: "data_access",
  user_id: ObjectId("..."),
  session_id: "sess_123456",
  resource: {
    collection: "trading_strategies",
    document_id: ObjectId("..."),
    operation: "read"
  },
  metadata: {
    ip_address: "192.168.1.100",
    user_agent: "Mozilla/5.0...",
```

```
    api_version: "2.1.0"
  },
  compliance_tags: ["MiFID II", "audit_trail"],
  retention_until: ISODate("2032-01-15T10:30:00.000Z")
}
```

Audit Event Categories:

Event Category	Logging Level	Retention	Monitoring	Alerting
Authentication	All events	7 years	Real-time	Failed attempts
Data Access	Sensitive data only	5 years	Daily review	Unusual patterns
System Changes	All changes	7 years	Real-time	All changes
Trading Operations	All operations	5 years	Real-time	High-risk trades

6.2.3.5 Access Controls

Role-Based Access Control (RBAC):

The system implements fine-grained access controls with role-based permissions and attribute-based access control for sensitive operations.

Access Control Matrix:

Role	Market Data	Trading Strategies	User Management	System Admin
Basic User	Read own	CRUD own	Read own profile	None
Premium User	Read all	CRUD own + templates	Read own profile	None
Analyst	Read all	Read all	None	None

Role	Market Data	Trading Strategies	User Management	System Admin
Administrator	Full access	Full access	Full access	Full access

MongoDB Access Control Configuration:

```
// Database User Roles
db.createRole({
  role: "tradingUser",
  privileges: [
    {
      resource: { db: "forex_trading", collection: "market_data" },
      actions: ["find"]
    },
    {
      resource: { db: "forex_trading", collection: "trading_strategies" }
      actions: ["find", "insert", "update", "remove"],
      condition: { "user_id": "$$USER_ID" }
    }
  ],
  roles: []
})

// Field-Level Security
db.users.createIndex(
  { "email": 1 },
  {
    partialFilterExpression: { "role": { $in: ["admin", "analyst"] } },
    name: "admin_email_access"
  }
)
```

6.2.4 PERFORMANCE OPTIMIZATION

6.2.4.1 Query Optimization Patterns

MongoDB Query Optimization Strategy:

To improve performance for queries that your application runs frequently, create indexes on commonly queried fields. As your application grows, monitor your deployment's index use to ensure that your indexes are still supporting relevant queries.

Query Pattern Analysis:

Query Pattern	Frequency	Optimization Strategy	Index Requirements
Real-time price lookup	Very High	Compound index + caching	symbol + timestamp
User strategy retrieval	High	Compound index	user_id + status
Historical data analysis	Medium	Time series optimization	timestamp range
ML model predictions	Medium	Caching + batch processing	model_id + created_at

Optimized Query Examples:

```
// Optimized Real-time Price Query
db.market_data.find({
  "metadata.symbol": "EURUSD",
  "timestamp": {
    $gte: ISODate("2025-01-15T10:00:00.000Z"),
    $lte: ISODate("2025-01-15T11:00:00.000Z")
  }
}).sort({ "timestamp": -1 }).limit(100)

// Optimized Strategy Aggregation
db.trading_strategies.aggregate([
  {
    $match: {
      user_id: ObjectId("..."),
      status: "active"
    }
  },
  {
    $lookup: {
```

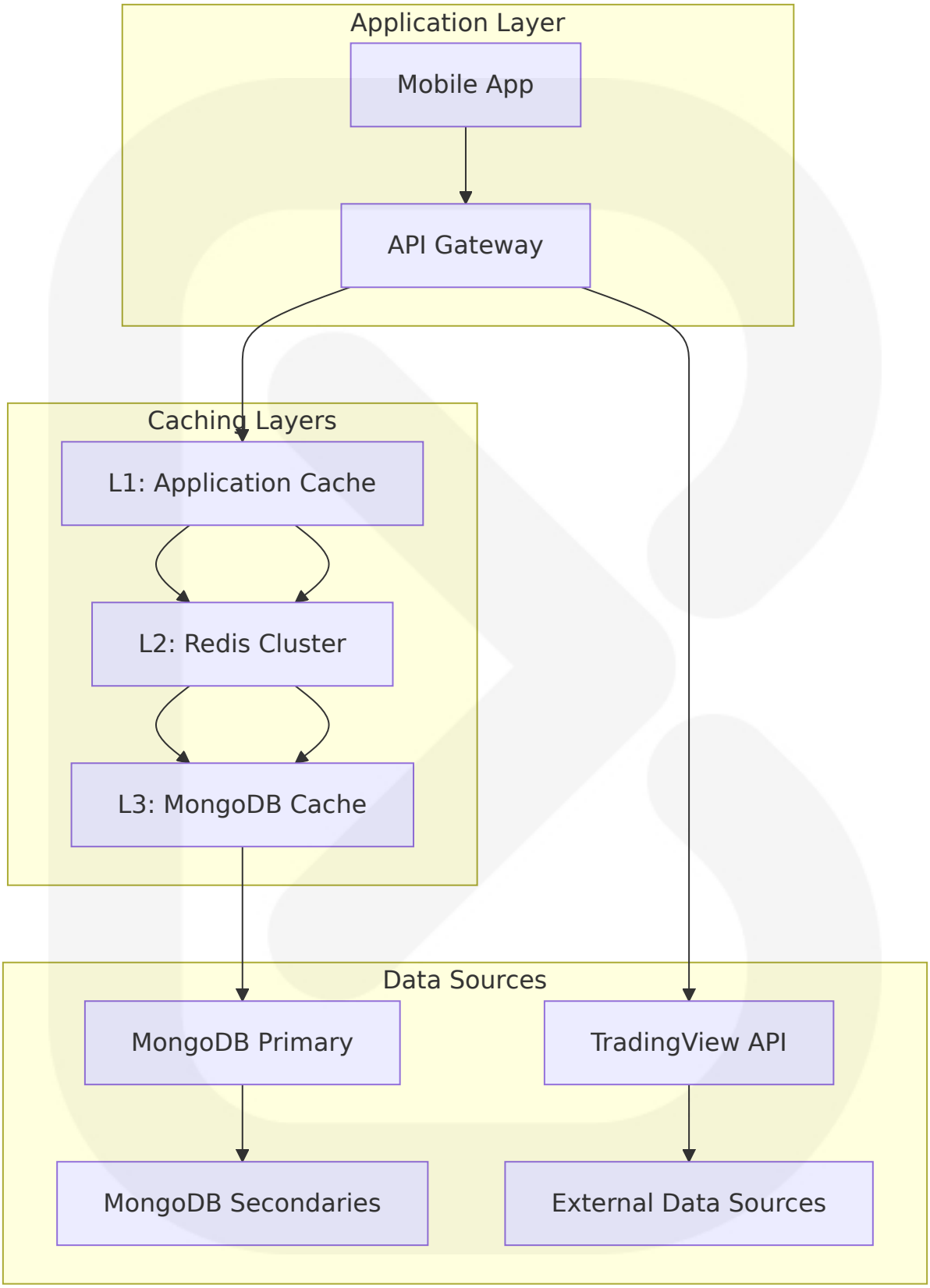
```
        from: "ml_predictions",
        localField: "_id",
        foreignField: "strategy_id",
        as: "predictions"
      }
    },
    {
      $project: {
        strategy_name: 1,
        currency_pair: 1,
        "predictions.confidence_score": 1
      }
    }
  ])
```

6.2.4.2 Caching Strategy

Multi-Level Caching Architecture:

With caching, data stored in slower databases can achieve sub-millisecond performance. That helps businesses to respond to the need for real-time applications.

Caching Layer Configuration:



Cache Performance Metrics:

Cache Level	Hit Ratio Target	Latency Target	Capacity	Eviction Policy
L1 Application	>95%	<1ms	100MB	LRU
L2 Redis	>90%	<5ms	10GB	TTL + LRU
L3 MongoDB	>80%	<50ms	100GB	Working Set

6.2.4.3 Connection Pooling

Database Connection Management:

The system implements intelligent connection pooling to optimize database resource utilization and minimize connection overhead.

Connection Pool Configuration:

```
// MongoDB Connection Pool Settings
const mongoOptions = {
  maxPoolSize: 100,
  minPoolSize: 10,
  maxIdleTimeMS: 30000,
  waitQueueTimeoutMS: 5000,
  serverSelectionTimeoutMS: 5000,
  heartbeatFrequencyMS: 10000,
  retryWrites: true,
  retryReads: true
}

// Redis Connection Pool Settings
const redisOptions = {
  host: 'redis-cluster',
  port: 6379,
  maxRetriesPerRequest: 3,
  retryDelayOnFailover: 100,
  enableOfflineQueue: false,
  lazyConnect: true,
  keepAlive: 30000,
  family: 4,
```

```
  connectTimeout: 10000,
  commandTimeout: 5000
}
```

Connection Pool Monitoring:

Metric	Threshold	Action	Monitoring Frequency
Active Connections	>80% of max	Scale pool size	Every 30 seconds
Wait Queue Length	>10 requests	Add connections	Every 10 seconds
Connection Errors	>1% error rate	Health check	Every 5 seconds
Response Time	>100ms average	Pool optimization	Every minute

6.2.4.4 Read/Write Splitting

Read/Write Separation Strategy:

The system implements intelligent read/write splitting to distribute load across replica set members and optimize performance for different workload patterns.

Read/Write Routing Configuration:

```
// Read Preference Configuration
const readPreferences = {
  realtime_data: {
    mode: 'primary',
    maxStalenessSeconds: 0
  },
  analytics_queries: {
    mode: 'secondaryPreferred',
    maxStalenessSeconds: 30,
    tags: [{ datacenter: 'analytics' }]
  },
```

```
  reporting_queries: {
    mode: 'secondary',
    maxStalenessSeconds: 300,
    tags: [{ workload: 'reporting' }]
  }
}

// Write Concern Configuration
const writeConcerns = {
  critical_data: {
    w: 'majority',
    j: true,
    wtimeout: 5000
  },
  analytics_data: {
    w: 1,
    j: false,
    wtimeout: 1000
  }
}
```

6.2.4.5 Batch Processing Approach

Efficient Batch Operations:

The system implements optimized batch processing for high-volume operations including market data ingestion, ML model training, and historical data analysis.

Batch Processing Pipeline:

```
// Market Data Batch Insert
const batchInsertMarketData = async (marketDataArray) => {
  const batchSize = 1000
  const batches = []

  for (let i = 0; i < marketDataArray.length; i += batchSize) {
    batches.push(marketDataArray.slice(i, i + batchSize))
  }

  const results = await Promise.all(
```

```
    batches.map(batch =>
      db.collection('market_data').insertMany(batch, {
        ordered: false,
        writeConcern: { w: 1, j: false }
      })
    )
  )

  return results
}

// Aggregation Pipeline Optimization
const optimizedAggregation = [
  {
    $match: {
      timestamp: {
        $gte: startDate,
        $lte: endDate
      }
    }
  },
  {
    $group: {
      _id: {
        symbol: "$metadata.symbol",
        hour: { $hour: "$timestamp" }
      },
      avgPrice: { $avg: "$close" },
      volume: { $sum: "$volume" },
      count: { $sum: 1 }
    }
  },
  {
    $sort: { "_id.symbol": 1, "_id.hour": 1 }
  }
]
```

Batch Processing Performance:

Operation Type	Batch Size	Throughput	Latency	Resource Usage
Market Data Insert	1000 docs	50K docs/sec	<100ms	CPU: 60%, Memory: 2GB
Technical Analysis	500 symbols	100 symbols/sec	<5s	CPU: 80%, Memory: 4GB
ML Predictions	100 strategies	20 predictions/sec	<5s	CPU: 90%, Memory: 8GB
Historical Aggregation	10K docs	5K docs/sec	<2s	CPU: 70%, Memory: 6GB

The database design provides a robust, scalable foundation for the forex trading application, leveraging MongoDB's time series collections for optimal market data storage and Redis for high-performance caching. The comprehensive indexing strategy, combined with intelligent caching and connection pooling, ensures sub-second response times for real-time trading operations while maintaining data consistency and regulatory compliance.

6.3 INTEGRATION ARCHITECTURE

6.3.1 API DESIGN

6.3.1.1 Protocol Specifications

The forex trading application implements a comprehensive API architecture supporting multiple communication protocols optimized for real-time market data processing and mobile application requirements.

Primary Communication Protocols:

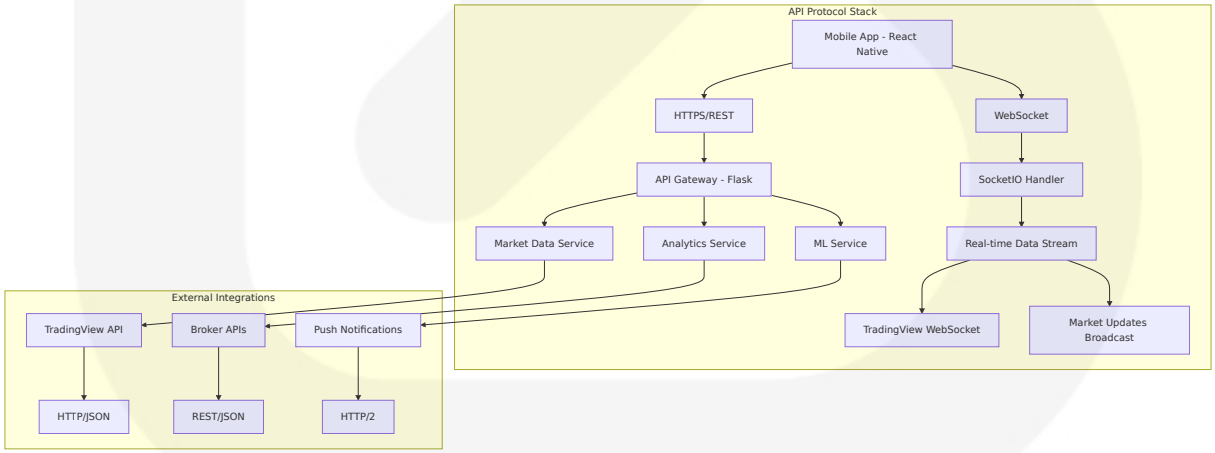
Protocol	Use Case	Implementation	Performance Characteristics
HTTPS/REST	Standard API operations	Flask 3.1.2 with Gunicorn	<500ms response time
WebSocket	Real-time market data	Flask-SocketIO with eventlet	<2 second latency
HTTP/2	Mobile optimization	Nginx reverse proxy	Multiplexed connections

REST API Specification:

The TradingView REST API acts as a frontend for brokers' backend systems, with the API specification designed for exclusive financial products including TradingView Web Platform & Trading Terminal. The system implements RESTful endpoints following OpenAPI 3.0 specification with JSON request/response format.

WebSocket Protocol Implementation:

WebSocket communication allows for real-time, bi-directional data transfers between the client and server, enabling the setup of a React + Flask application with two-way interactive communication using WebSockets with socket.io.

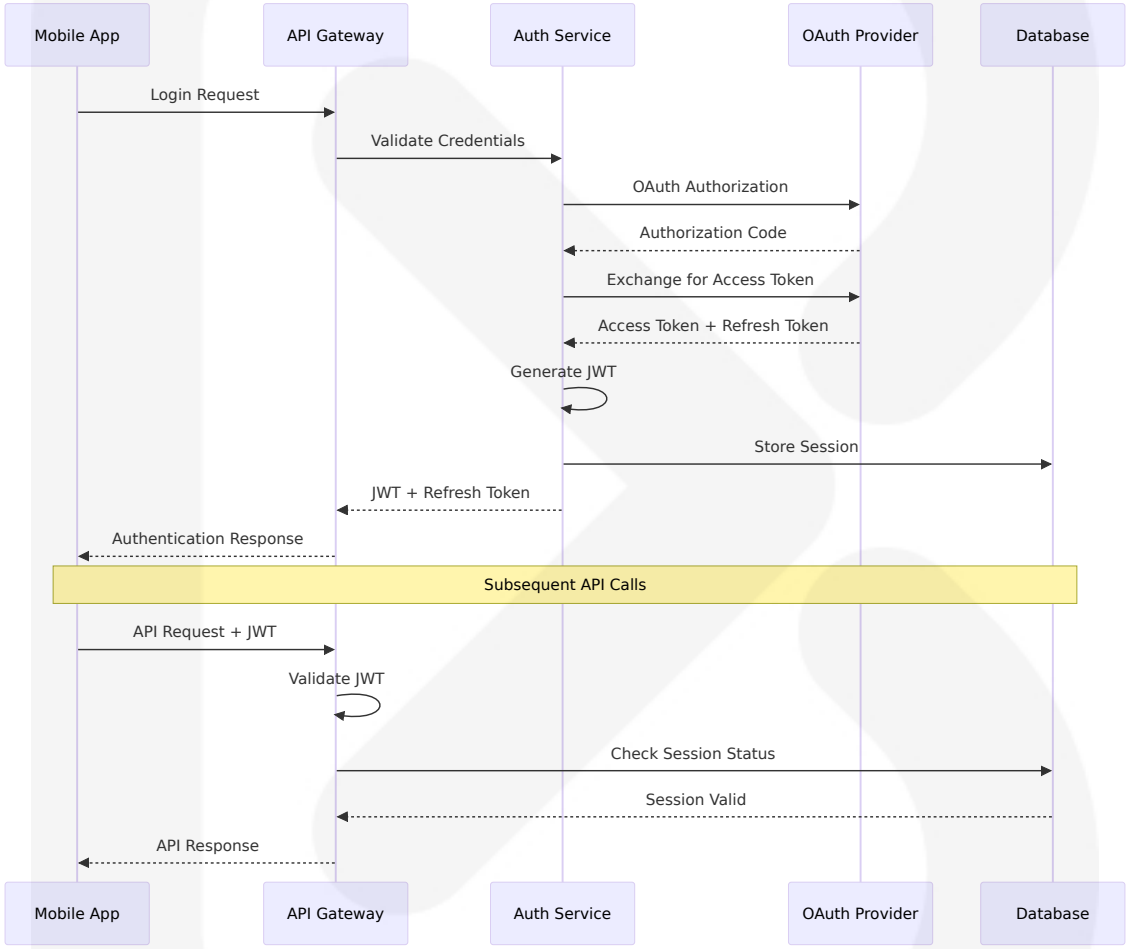


6.3.1.2 Authentication Methods

OAuth 2.0 + JWT Implementation:

The system implements a hybrid authentication approach combining OAuth 2.0 for initial authentication with JWT tokens for subsequent API access, providing both security and performance optimization for mobile applications.

Authentication Flow Architecture:



Authentication Configuration Matrix:

Authentication Method	Use Case	Token Lifetime	Security Level
OAuth 2.0	Initial authentication	N/A	High
JWT Access Token	API requests	15 minutes	Medium

Authentication Method	Use Case	Token Lifetime	Security Level
Refresh Token	Token renewal	7 days	High
API Key	Service-to-service	Permanent	Medium

6.3.1.3 Authorization Framework

Role-Based Access Control (RBAC):

The system implements fine-grained authorization with role-based permissions and attribute-based access control for sensitive trading operations.

Authorization Matrix:

Role	Market Data	Trading Strategies	Portfolio Management	Admin Functions
Basic User	Read own subscriptions	CRUD own strategies	Read own portfolio	None
Premium User	Read all pairs	CRUD + templates	Full portfolio access	None
Analyst	Read all data	Read all strategies	Read all portfolios	None
Administrator	Full access	Full access	Full access	Full access

Permission Validation Flow:

```
// Authorization Middleware Implementation
const authorizeEndpoint = (requiredPermission) => {
  return async (req, res, next) => {
    try {
      const token = req.headers.authorization?.split(' ')[1];
      const decoded = jwt.verify(token, process.env.JWT_SECRET);

      const user = await User.findById(decoded.userId);
```

```
      const hasPermission = await checkPermission(user.role, requiredPerr

      if (!hasPermission) {
        return res.status(403).json({ error: 'Insufficient permissions' }
      }

      req.user = user;
      next();
    } catch (error) {
      return res.status(401).json({ error: 'Invalid token' });
    }
  };
};
```

6.3.1.4 Rate Limiting Strategy

Multi-Tier Rate Limiting:

The system implements sophisticated rate limiting to protect against abuse while ensuring legitimate high-frequency trading operations can function effectively.

Rate Limiting Configuration:

Endpoint Category	Rate Limit	Window	Burst Allowance	Throttling Strategy
Authentication	5 requests/minute	1 minute	2 requests	Exponential backoff
Market Data	100 requests/minute	1 minute	20 requests	Token bucket
Trading Operations	50 requests/minute	1 minute	10 requests	Sliding window
ML Predictions	20 requests/minute	1 minute	5 requests	Fixed window

Rate Limiting Implementation:

```python
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
import redis

#### Redis-backed rate limiter
limiter = Limiter(
    app,
    key_func=get_remote_address,
    storage_uri="redis://localhost:6379",
    default_limits=["1000 per hour", "100 per minute"]
)

#### Endpoint-specific rate limiting
@app.route('/api/market-data')
@limiter.limit("100 per minute")
@limiter.limit("1000 per hour")
def get_market_data():
    return jsonify({"data": "market_data"})

@app.route('/api/ml-prediction')
@limiter.limit("20 per minute")
@limiter.limit("200 per hour")
def get_ml_prediction():
    return jsonify({"prediction": "result"})
```

6.3.1.5 Versioning Approach

Semantic API Versioning:
The system implements comprehensive API versioning supporting backward compatibility and gradual migration of client applications.

Versioning Strategy:

Versioning Method	Implementation	Use Case	Migration Timeline
Header-based	`API-Version: 2.1.0`	Primary versioning	6 months overlap
URL-based	`/api/v2/market-data`	Legacy support	12 months support

Versioning Method	Implementation	Use Case	Migration Timeline
Content negotiation	`Accept: application/vnd.api+json;version=2`	Advanced clients	Optional

Version Compatibility Matrix:

```
// API Version Management
const API_VERSIONS = {
  '1.0.0': {
    supported: false,
    deprecated: true,
    sunset_date: '2025-06-01'
  },
  '2.0.0': {
    supported: true,
    deprecated: false,
    features: ['basic_trading', 'market_data']
  },
  '2.1.0': {
    supported: true,
    deprecated: false,
    features: ['basic_trading', 'market_data', 'ml_predictions', 'advance
  }
};

// Version validation middleware
const validateApiVersion = (req, res, next) => {
  const version = req.headers['api-version'] || '2.0.0';

  if (!API_VERSIONS[version] || !API_VERSIONS[version].supported) {
    return res.status(400).json({
      error: 'Unsupported API version',
      supported_versions: Object.keys(API_VERSIONS).filter(v => API_VERSI
    });
  }

  req.apiVersion = version;
  next();
};
```

6.3.1.6 Documentation Standards

OpenAPI 3.0 Specification:
The system maintains comprehensive API documentation using OpenAPI 3.0 specification with automated generation from code annotations and interactive documentation.

Documentation Structure:

- **Interactive API Explorer**: Swagger UI for testing endpoints
- **Code Examples**: Multi-language client examples
- **Authentication Guide**: Step-by-step authentication setup
- **Rate Limiting Guide**: Usage limits and best practices
- **Error Handling**: Comprehensive error codes and responses

API Documentation Example:

```
openapi: 3.0.0
info:
  title: Forex Trading API
  version: 2.1.0
  description: Comprehensive forex market analysis and trading strategy A

paths:
  /api/v2/market-data/{symbol}:
    get:
      summary: Get real-time market data
      parameters:
        - name: symbol
          in: path
          required: true
          schema:
            type: string
            example: "EURUSD"
        - name: timeframe
          in: query
          schema:
            type: string
            enum: ["1m", "5m", "15m", "1h", "4h", "1d"]
            default: "1m"
```

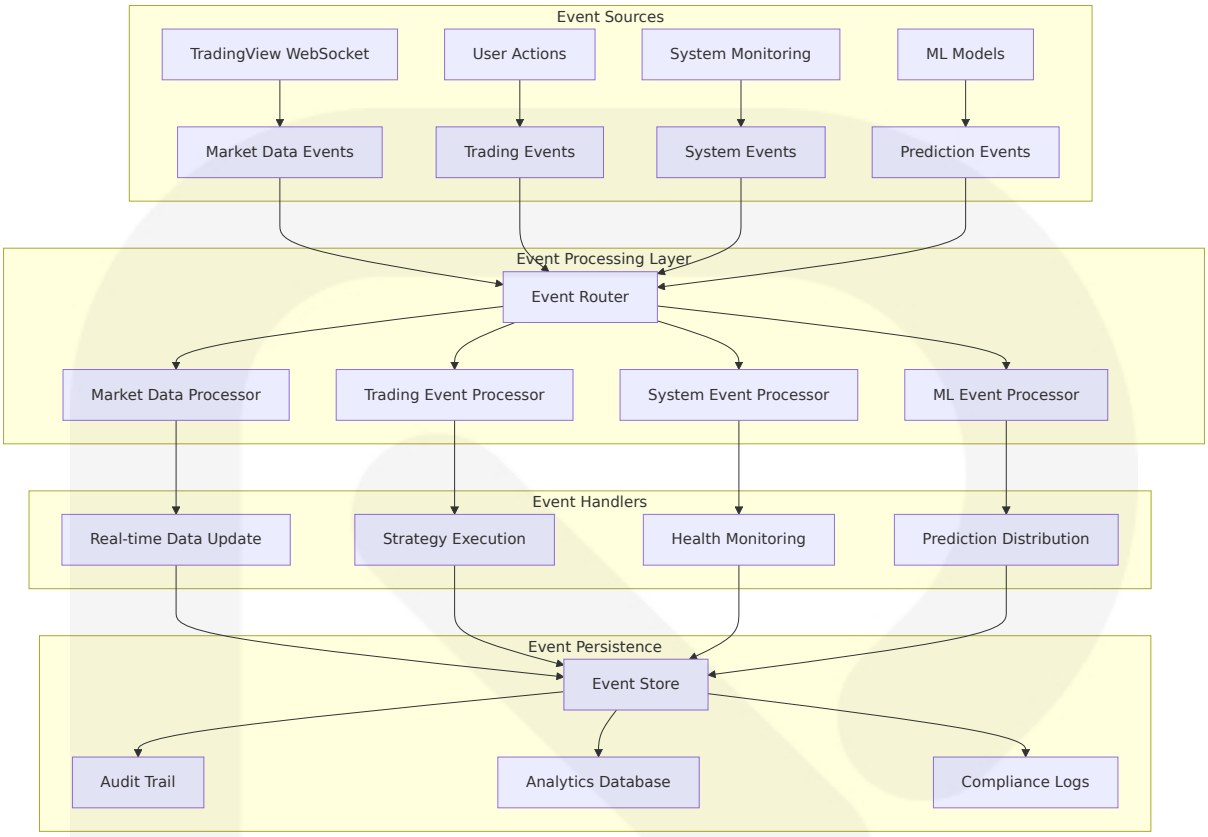
```
        responses:
          '200':
            description: Market data retrieved successfully
            content:
              application/json:
                schema:
                  $ref: '#/components/schemas/MarketData'
          '429':
            description: Rate limit exceeded
            headers:
              X-RateLimit-Remaining:
                schema:
                  type: integer
              X-RateLimit-Reset:
                schema:
                  type: integer
```

6.3.2 MESSAGE PROCESSING

6.3.2.1 Event Processing Patterns

Event-Driven Architecture Implementation:
The system implements comprehensive event processing patterns to handle real-time market data, user actions, and system events with high throughput and low latency requirements.

Event Processing Flow:



Event Processing Patterns:

Pattern	Use Case	Implementation	Performance
Publish-Subscribe	Market data distribution	Redis pub/sub	<100ms latency
Event Sourcing	Trading strategy audit	MongoDB event store	Eventual consistency
CQRS	Read/write separation	Separate read models	Optimized queries
Saga Pattern	Multi-step trading operations	State machine	Transactional consistency

6.3.2.2 Message Queue Architecture

Redis-Based Message Queue:

Flask-SocketIO integrates WebSocket support into Flask applications, with a SocketIO instance created and linked to the Flask application, allowing it to

handle WebSocket connections. The system utilizes Redis as the primary message broker for asynchronous processing and real-time communication.

Message Queue Configuration:

```
# Redis Message Queue Setup
import redis
from rq import Queue, Worker
import json

#### Redis connection
redis_conn = redis.Redis(
    host='localhost',
    port=6379,
    db=0,
    decode_responses=True
)

#### Queue definitions
market_data_queue = Queue('market_data', connection=redis_conn)
analytics_queue = Queue('analytics', connection=redis_conn)
ml_prediction_queue = Queue('ml_predictions', connection=redis_conn)
notification_queue = Queue('notifications', connection=redis_conn)

#### Message processing function
def process_market_data(symbol, data):
    """Process incoming market data"""
    try:
#### Validate and normalize data
        normalized_data = normalize_market_data(data)

#### Store in database
        store_market_data(symbol, normalized_data)

#### Trigger analytics processing
        analytics_queue.enqueue(
            'analytics.process_technical_indicators',
            symbol,
            normalized_data
        )
```

```
#### Broadcast to connected clients
        socketio.emit('market_update', {
            'symbol': symbol,
            'data': normalized_data
        }, room=f'market_{symbol}')

    except Exception as e:
        logger.error(f"Market data processing error: {e}")
        raise
```

Queue Management Strategy:

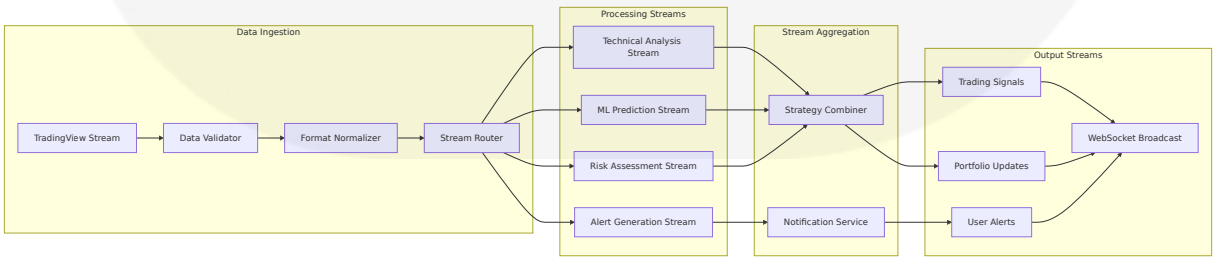
Queue Type	Priority	Workers	Retry Policy	Dead Letter Queue
Market Data	High	5 workers	3 retries, exponential backoff	Yes
Analytics	Medium	3 workers	2 retries, linear backoff	Yes
ML Predictions	Medium	2 workers	1 retry, immediate	Yes
Notifications	Low	2 workers	3 retries, exponential backoff	No

6.3.2.3 Stream Processing Design

Real-time Stream Processing:

The system implements stream processing for continuous market data analysis and real-time trading signal generation using event-driven patterns.

Stream Processing Architecture:



Stream Processing Implementation:

```python
# Stream processing with asyncio
import asyncio
import aioredis
from typing import AsyncGenerator

class MarketDataStream:
    def __init__(self):
        self.redis = None
        self.subscribers = {}

    async def initialize(self):
        self.redis = await aioredis.from_url("redis://localhost")

    async def market_data_stream(self, symbol: str) -> AsyncGenerator:
        """Stream market data for a specific symbol"""
        pubsub = self.redis.pubsub()
        await pubsub.subscribe(f'market_data:{symbol}')

        try:
            async for message in pubsub.listen():
                if message['type'] == 'message':
                    data = json.loads(message['data'])
                    yield data
        finally:
            await pubsub.unsubscribe(f'market_data:{symbol}')

    async def process_stream(self, symbol: str):
        """Process market data stream with analytics"""
        async for data in self.market_data_stream(symbol):
            # Technical analysis processing
            indicators = await self.calculate_indicators(data)

            # ML prediction processing
            prediction = await self.get_ml_prediction(data, indicators)

            # Risk assessment
            risk_score = await self.assess_risk(data, prediction)

            # Generate trading signal
            signal = await self.generate_signal(indicators, prediction,
```

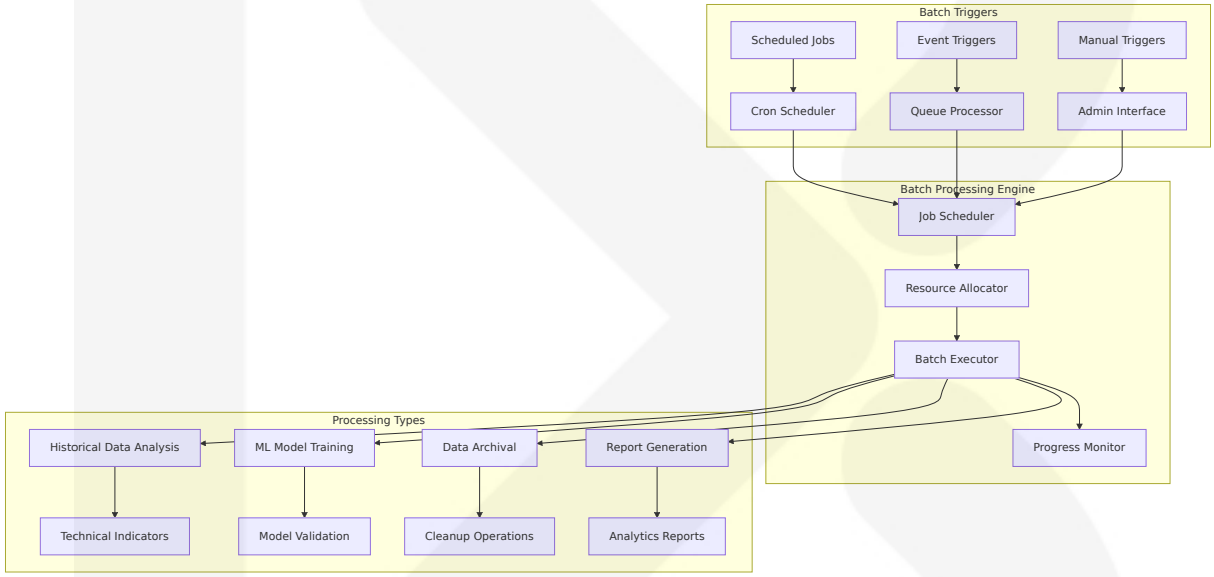
```
            # Broadcast to subscribers
            await self.broadcast_signal(symbol, signal)
```

6.3.2.4 Batch Processing Flows

Batch Processing Architecture:

The system implements efficient batch processing for historical data analysis, model training, and periodic maintenance operations.

Batch Processing Workflow:



Batch Job Configuration:

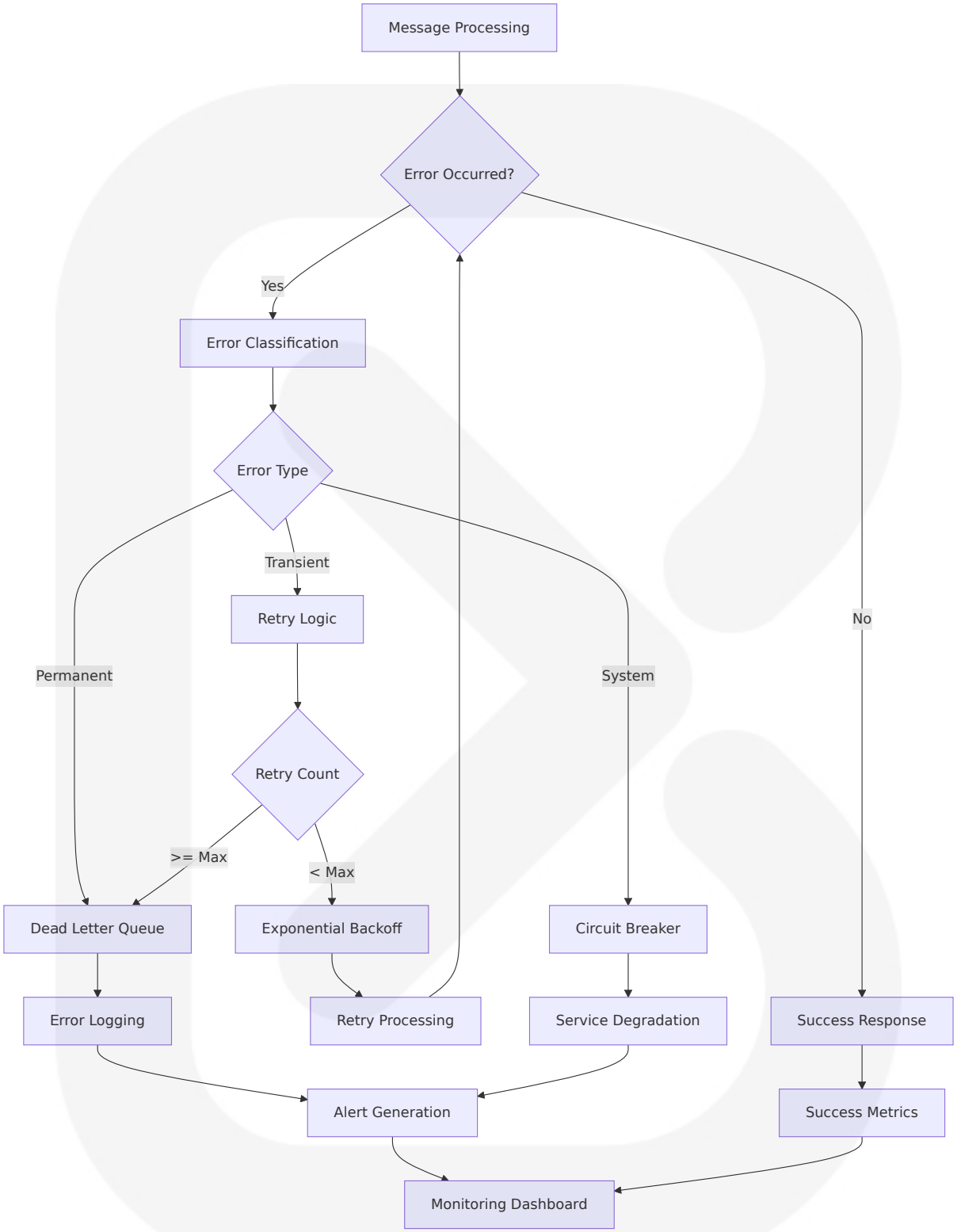
Job Type	Schedule	Duration	Resource Requirements	Retry Policy
Historical Analysis	Daily 2 AM	2-4 hours	4 CPU, 8GB RAM	2 retries
ML Model Training	Weekly Sunday	6-12 hours	8 CPU, 16GB RAM, GPU	1 retry
Data Archival	Monthly 1st	1-2 hours	2 CPU, 4GB RAM	3 retries
Report Generation	Daily 6 AM	30 minutes	2 CPU, 2GB RAM	2 retries

6.3.2.5 Error Handling Strategy

Comprehensive Error Handling Framework:

The system implements multi-layered error handling with automatic recovery, circuit breakers, and comprehensive logging for all message processing operations.

Error Handling Flow:



Error Handling Implementation:

```python
import logging
from enum import Enum
from typing import Optional
import traceback

class ErrorType(Enum):
    TRANSIENT = "transient"
    PERMANENT = "permanent"
    SYSTEM = "system"

class MessageProcessor:
    def __init__(self):
        self.logger = logging.getLogger(__name__)
        self.max_retries = 3
        self.circuit_breaker = CircuitBreaker()

    async def process_message(self, message: dict, retry_count: int = 0)
        """Process message with comprehensive error handling"""
        try:
            # Circuit breaker check
            if self.circuit_breaker.is_open():
                raise SystemError("Circuit breaker is open")

            # Process the message
            result = await self._process_message_logic(message)

            # Reset circuit breaker on success
            self.circuit_breaker.record_success()

            return result

        except Exception as e:
            error_type = self._classify_error(e)

            # Log error with context
            self.logger.error(
                f"Message processing error: {str(e)}",
                extra={
                    'message_id': message.get('id'),
                    'error_type': error_type.value,
                    'retry_count': retry_count,
                    'traceback': traceback.format_exc()
                }
```

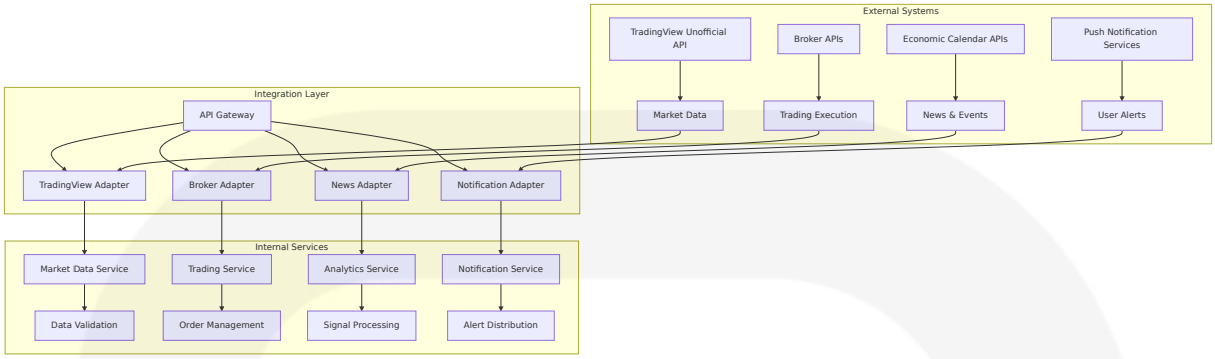
```
            )

            # Handle based on error type
            if error_type == ErrorType.TRANSIENT and retry_count < self.r
                # Exponential backoff retry
                delay = 2 ** retry_count
                await asyncio.sleep(delay)
                return await self.process_message(message, retry_count +

            elif error_type == ErrorType.SYSTEM:
                # Record failure for circuit breaker
                self.circuit_breaker.record_failure()

            # Send to dead letter queue for permanent errors or max retri
            await self._send_to_dead_letter_queue(message, str(e))
            raise

    def _classify_error(self, error: Exception) -> ErrorType:
        """Classify error type for appropriate handling"""
        if isinstance(error, (ConnectionError, TimeoutError)):
            return ErrorType.TRANSIENT
        elif isinstance(error, (ValueError, KeyError)):
            return ErrorType.PERMANENT
        else:
            return ErrorType.SYSTEM
```

6.3.3 EXTERNAL SYSTEMS

6.3.3.1 Third-party Integration Patterns

TradingView API Integration:
TradingView doesn't have an API that gives access to data as of now, but their REST API is meant for brokers who want to be supported on their trading platform. The system implements unofficial API wrappers and alternative integration patterns for market data access.

Integration Architecture:



Third-party Integration Configuration:

Service	Integration Type	Authentication	Rate Limits	Fallback Strategy
TradingView	Unofficial API	API Key	100 req/min	Alternative data sources
Interactive Brokers	Official API	OAuth 2.0	50 req/min	Order queuing
Alpha Vantage	REST API	API Key	5 req/min	Data caching
Firebase FCM	Push API	Service Account	1M msg/day	Email fallback

6.3.3.2 Legacy System Interfaces

Broker Integration Patterns:

Integrating TradingView with Interactive Brokers takes trading to the next level by merging advanced charting and analysis tools with seamless market access, requiring both accounts to be active and properly configured.

Legacy System Adapter Pattern:

```
from abc import ABC, abstractmethod
from typing import Dict, Any, Optional

class BrokerAdapter(ABC):
    """Abstract base class for broker integrations"""
```

```python
    @abstractmethod
    async def authenticate(self, credentials: Dict[str, str]) -> bool:
        pass

    @abstractmethod
    async def get_account_info(self) -> Dict[str, Any]:
        pass

    @abstractmethod
    async def place_order(self, order: Dict[str, Any]) -> Dict[str, Any]
        pass

    @abstractmethod
    async def get_positions(self) -> List[Dict[str, Any]]:
        pass

class InteractiveBrokersAdapter(BrokerAdapter):
    """Interactive Brokers API adapter"""

    def __init__(self, config: Dict[str, str]):
        self.host = config.get('host', 'localhost')
        self.port = config.get('port', 7497)
        self.client_id = config.get('client_id', 1)
        self.connection = None

    async def authenticate(self, credentials: Dict[str, str]) -> bool:
        """Authenticate with IB Gateway/TWS"""
        try:
            # Connect to IB Gateway
            self.connection = IBConnection(
                host=self.host,
                port=self.port,
                client_id=self.client_id
            )

            await self.connection.connect()
            return self.connection.is_connected()

        except Exception as e:
            logger.error(f"IB authentication failed: {e}")
            return False

    async def place_order(self, order: Dict[str, Any]) -> Dict[str, Any]
```

```
        """Place order through IB API"""
        try:
            ib_order = self._convert_to_ib_order(order)
            result = await self.connection.place_order(ib_order)
            return self._convert_from_ib_result(result)

        except Exception as e:
            logger.error(f"Order placement failed: {e}")
            raise BrokerError(f"Failed to place order: {e}")
```

6.3.3.3 API Gateway Configuration

Centralized API Gateway:

The system implements a comprehensive API gateway using Flask with advanced routing, authentication, rate limiting, and monitoring capabilities.

Gateway Configuration:

```
from flask import Flask, request, jsonify
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
import jwt
import redis

app = Flask(__name__)

#### Rate limiter configuration
limiter = Limiter(
    app,
    key_func=get_remote_address,
    storage_uri="redis://localhost:6379"
)

#### Redis for caching and session management
redis_client = redis.Redis(host='localhost', port=6379, db=0)

class APIGateway:
    def __init__(self):
        self.services = {
```

```
            'market-data': 'http://market-data-service:5001',
            'analytics': 'http://analytics-service:5002',
            'ml-predictions': 'http://ml-service:5003',
            'trading': 'http://trading-service:5004'
        }
        self.circuit_breakers = {}

    async def route_request(self, service: str, endpoint: str, **kwargs)
        """Route request to appropriate microservice"""
        service_url = self.services.get(service)
        if not service_url:
            raise ValueError(f"Unknown service: {service}")

        # Check circuit breaker
        if self._is_circuit_open(service):
            raise ServiceUnavailableError(f"Service {service} is unavaila

        try:
            # Forward request to service
            response = await self._forward_request(service_url, endpoint
            self._record_success(service)
            return response

        except Exception as e:
            self._record_failure(service)
            raise

#### API Gateway routes
@app.route('/api/v2/<service>/<path:endpoint>', methods=['GET', 'POST',
@limiter.limit("100 per minute")
def gateway_route(service, endpoint):
    """Main gateway routing function"""
    try:
#### Authentication check
        if not authenticate_request(request):
            return jsonify({'error': 'Authentication required'}), 401

#### Authorization check
        if not authorize_request(request, service, endpoint):
            return jsonify({'error': 'Insufficient permissions'}), 403

#### Route to service
        gateway = APIGateway()
```

```
        result = gateway.route_request(service, endpoint, request=reques

        return jsonify(result)

    except Exception as e:
        logger.error(f"Gateway routing error: {e}")
        return jsonify({'error': 'Internal server error'}), 500
```

6.3.3.4 External Service Contracts

Service Level Agreements (SLAs):
The system maintains formal contracts with external service providers defining performance expectations, availability requirements, and failure handling procedures.

External Service SLA Matrix:

Service Provider	Availability SLA	Response Time SLA	Data Accuracy	Support Level
TradingView (Unofficial)	95%	<5 seconds	99%	Community
Interactive Brokers	99.5%	<1 second	99.9%	Professional
Alpha Vantage	99%	<2 seconds	99.5%	Standard
Firebase FCM	99.9%	<500ms	99.9%	Enterprise

Contract Monitoring Implementation:

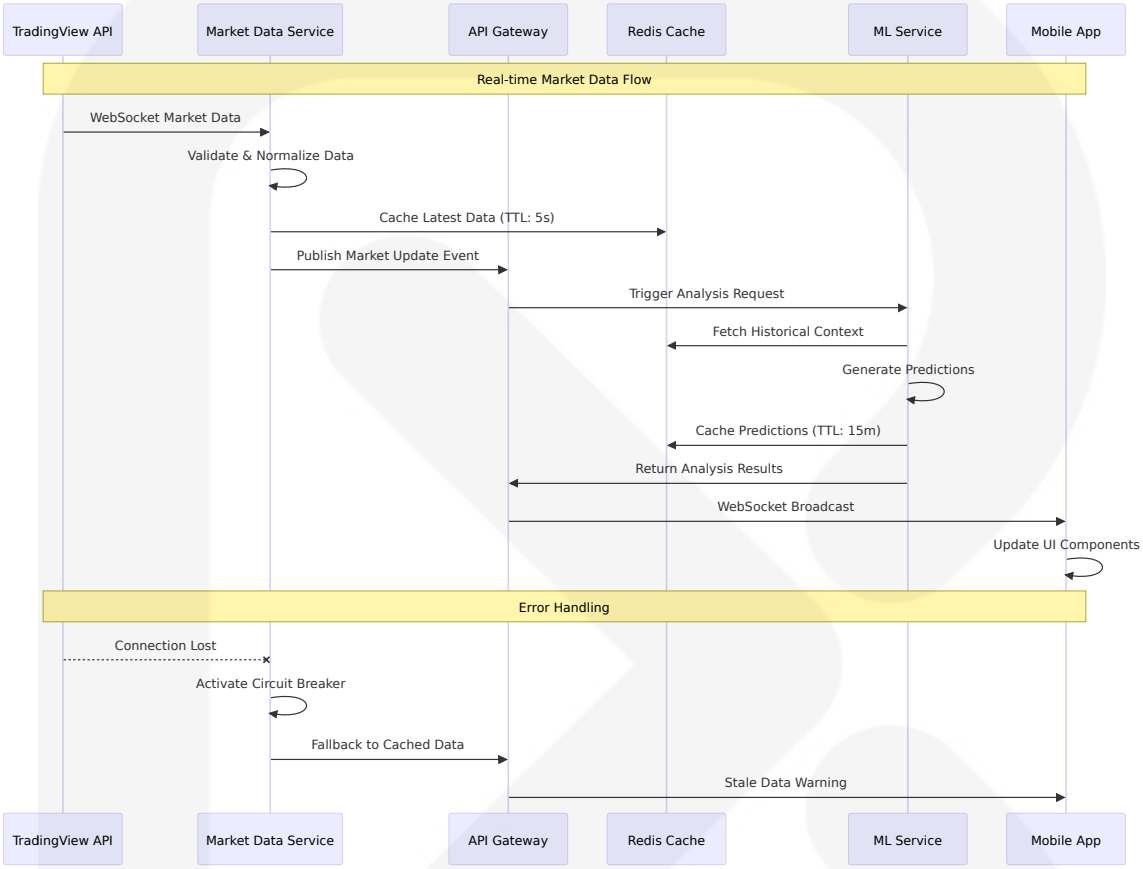
```
import time
from dataclasses import dataclass
from typing import Dict, List
import asyncio

@dataclass
class ServiceContract:
```

```
    name: str
    availability_sla: float  # Percentage
    response_time_sla: float  # Seconds
    data_accuracy_sla: float  # Percentage

class ContractMonitor:
    def __init__(self):
        self.contracts = {}
        self.metrics = {}

    def register_contract(self, contract: ServiceContract):
        """Register a service contract for monitoring"""
        self.contracts[contract.name] = contract
        self.metrics[contract.name] = {
            'total_requests': 0,
            'successful_requests': 0,
            'total_response_time': 0,
            'sla_violations': []
        }

    async def monitor_request(self, service_name: str, request_func):
        """Monitor a request against SLA"""
        if service_name not in self.contracts:
            return await request_func()

        start_time = time.time()
        metrics = self.metrics[service_name]
        contract = self.contracts[service_name]

        try:
            result = await request_func()
            response_time = time.time() - start_time

            # Update metrics
            metrics['total_requests'] += 1
            metrics['successful_requests'] += 1
            metrics['total_response_time'] += response_time

            # Check SLA violations
            if response_time > contract.response_time_sla:
                violation = {
                    'type': 'response_time',
                    'expected': contract.response_time_sla,
```

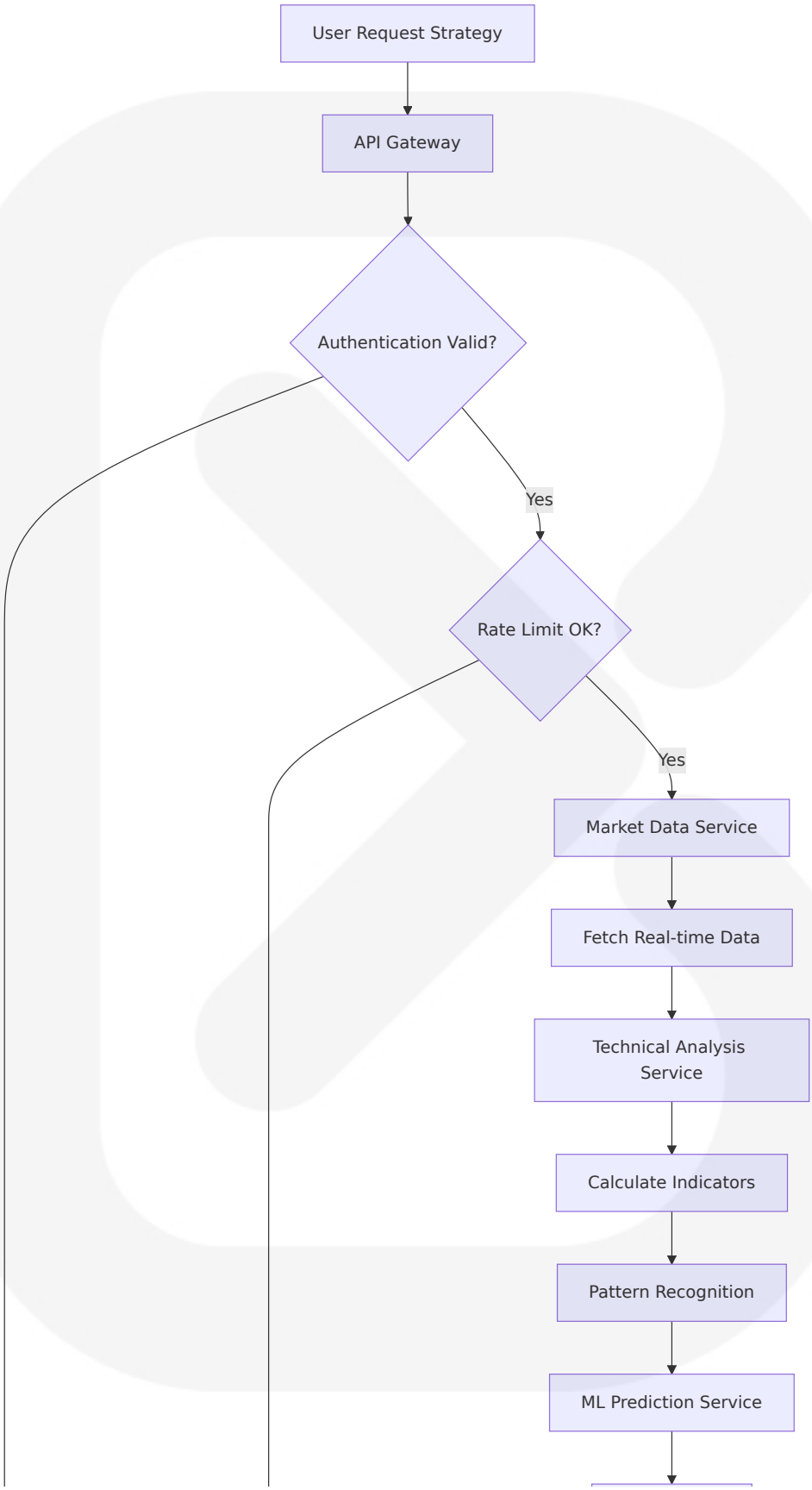
```
                    'actual': response_time,
                    'timestamp': time.time()
                }
                metrics['sla_violations'].append(violation)
                await self._handle_sla_violation(service_name, violation)

            return result

        except Exception as e:
            metrics['total_requests'] += 1
            # Record availability violation
            violation = {
                'type': 'availability',
                'error': str(e),
                'timestamp': time.time()
            }
            metrics['sla_violations'].append(violation)
            await self._handle_sla_violation(service_name, violation)
            raise

    def get_sla_compliance(self, service_name: str) -> Dict[str, float]:
        """Calculate SLA compliance metrics"""
        if service_name not in self.metrics:
            return {}

        metrics = self.metrics[service_name]
        contract = self.contracts[service_name]

        # Calculate availability
        availability = (metrics['successful_requests'] / metrics['total_
                        if metrics['total_requests'] > 0 else 0)

        # Calculate average response time
        avg_response_time = (metrics['total_response_time'] / metrics['su
                             if metrics['successful_requests'] > 0 else 0)

        return {
            'availability': availability,
            'availability_sla': contract.availability_sla,
            'avg_response_time': avg_response_time,
            'response_time_sla': contract.response_time_sla,
            'sla_violations': len(metrics['sla_violations'])
        }
```

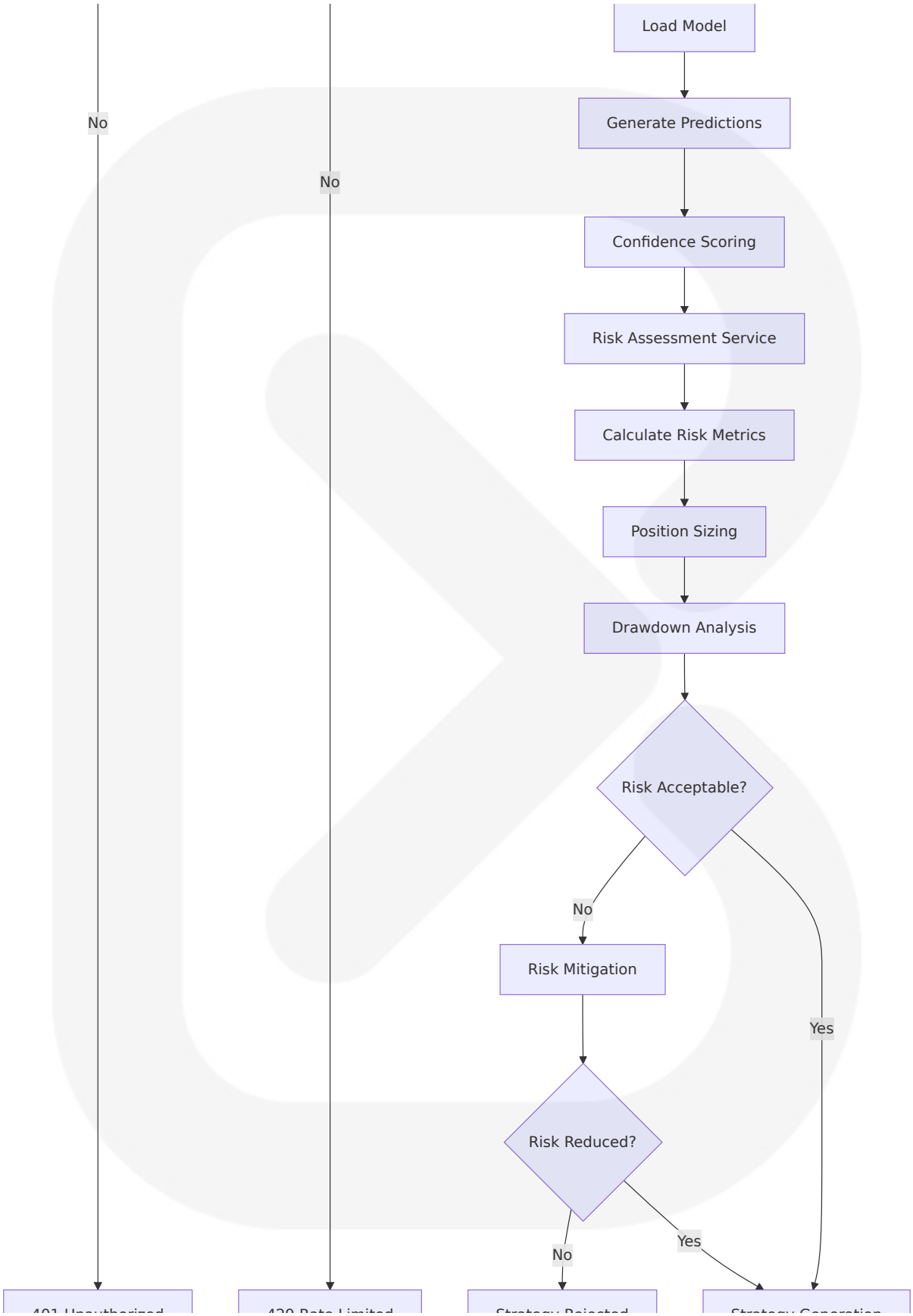
6.3.4 INTEGRATION FLOW DIAGRAMS

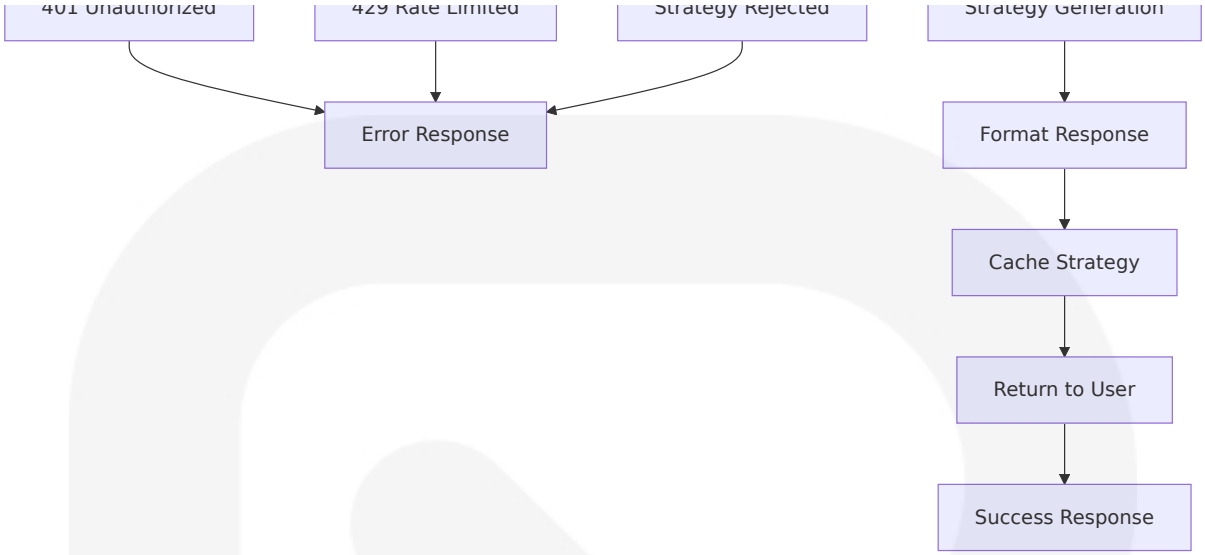
6.3.4.1 Real-time Market Data Integration Flow



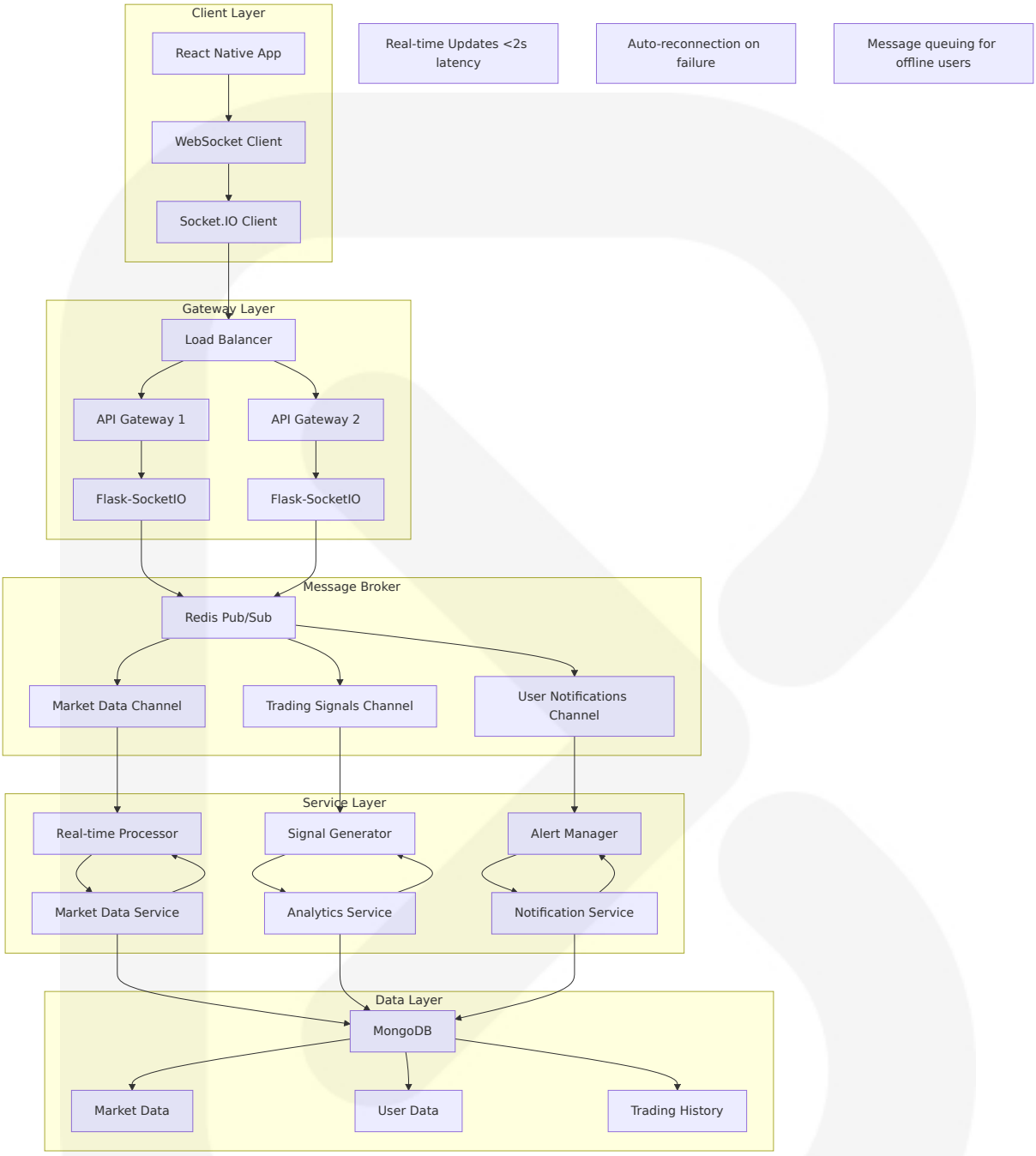
6.3.4.2 Trading Strategy Generation Flow


User Request Strategy
API Gateway
Authentication Valid?
Yes
Rate Limit OK?
Yes
Market Data Service
Fetch Real-time Data
Technical Analysis Service
Calculate Indicators
Pattern Recognition
ML Prediction Service

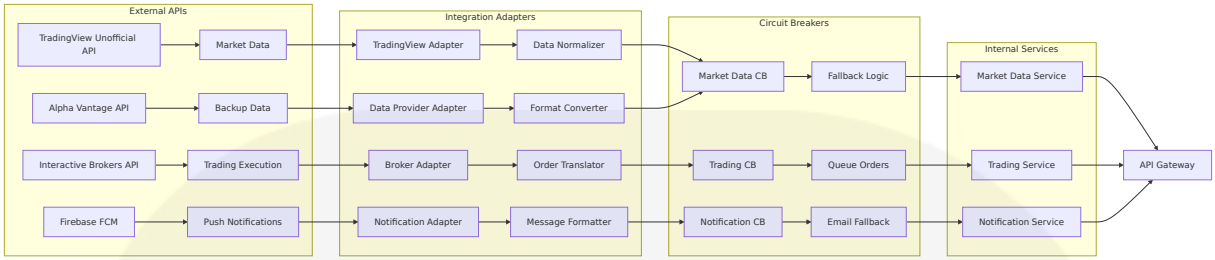

Load Model
Generate Predictions
Confidence Scoring
Risk Assessment Service
Calculate Risk Metrics
Position Sizing
Drawdown Analysis
Risk Acceptable?
No
Yes
Risk Mitigation
Risk Reduced?
No
Yes
No
No



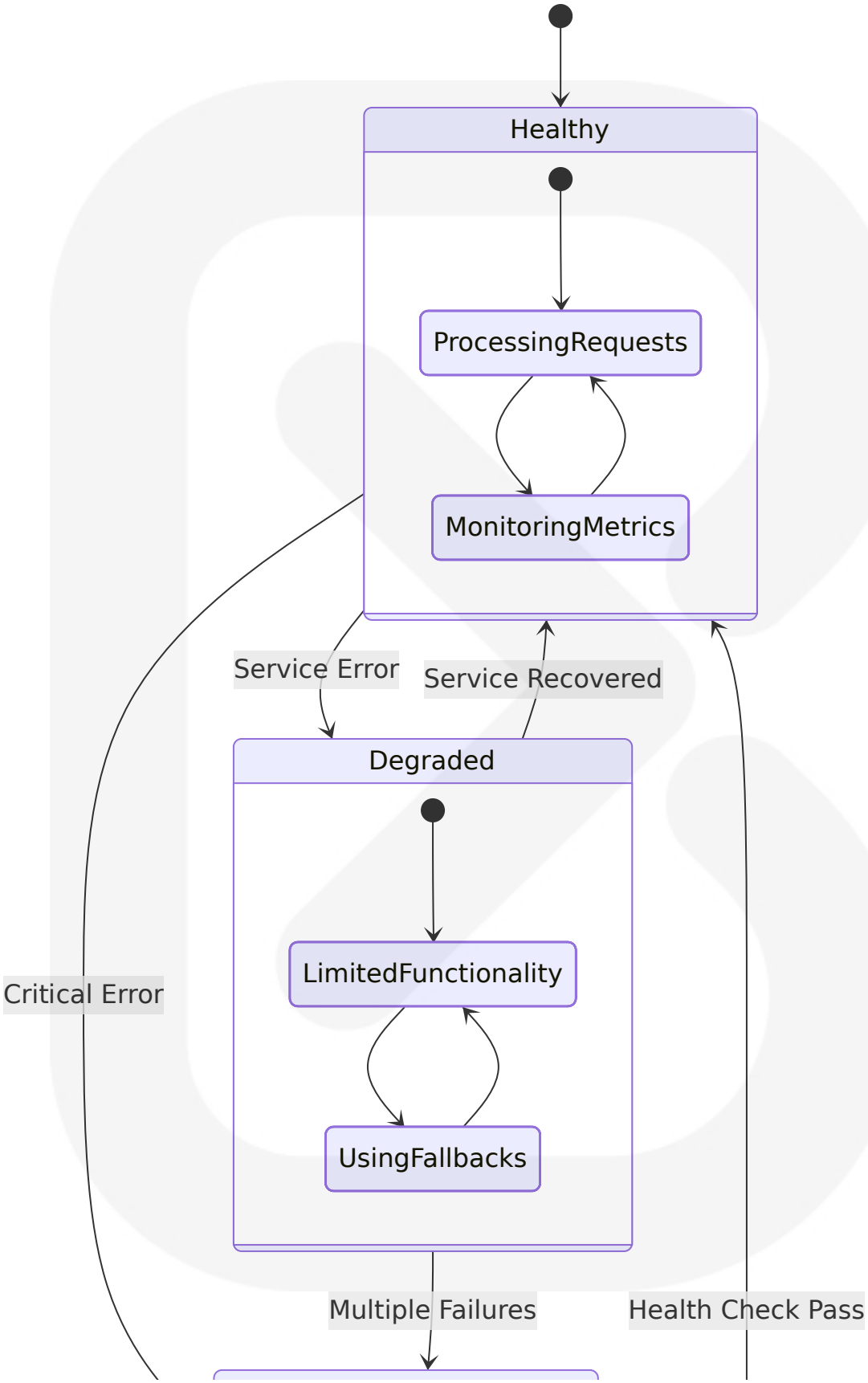
6.3.4.3 WebSocket Communication Architecture

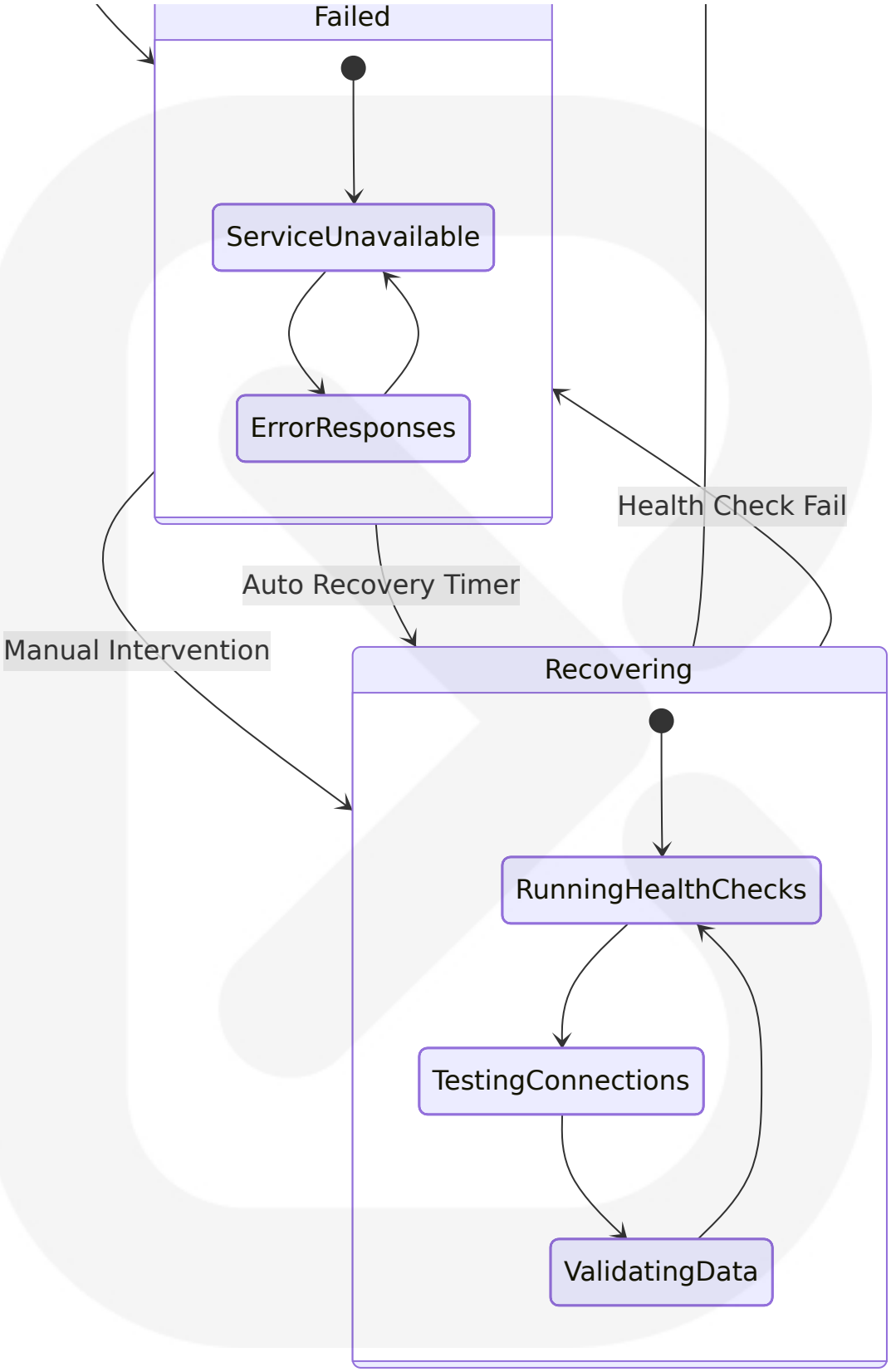


6.3.4.4 External API Integration Pattern



6.3.4.5 Error Handling and Recovery Flow


Healthy
ProcessingRequests
MonitoringMetrics
Service Error
Service Recovered
Degraded
LimitedFunctionality
UsingFallbacks
Critical Error
Multiple Failures
Health Check Pass



The Integration Architecture provides a comprehensive framework for

connecting the React Native mobile application with Flask-based backend services, external APIs, and real-time data sources. The system leverages modern React + Flask project patterns from 2025, combining Vite for frontend development with Flask for backend API services, while implementing robust error handling, circuit breaker patterns, and comprehensive monitoring to ensure reliable operation in production environments.

The architecture supports high-frequency trading requirements with sub-2-second latency for real-time market data, comprehensive rate limiting to prevent abuse, and sophisticated authentication and authorization mechanisms. The WebSocket-based real-time communication ensures traders receive immediate updates on market conditions and trading opportunities, while the comprehensive error handling and fallback mechanisms maintain system availability even when external services experience issues.

6.4 SECURITY ARCHITECTURE

6.4.1 AUTHENTICATION FRAMEWORK

6.4.1.1 Identity Management

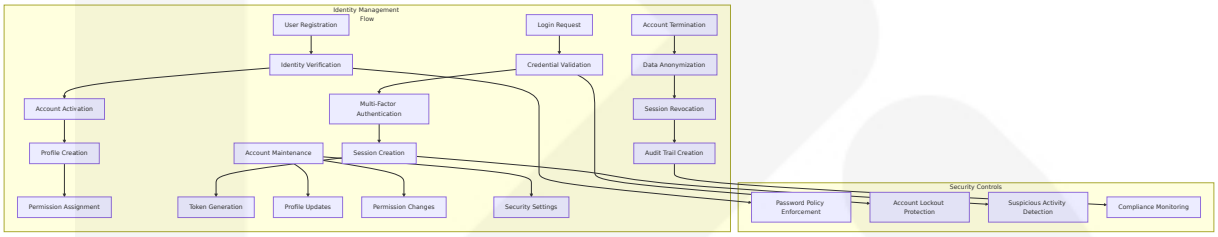
The forex trading application implements a comprehensive identity management system designed to handle the unique security requirements of financial trading platforms while ensuring compliance with regulatory standards including GDPR (General Data Protection Regulation) which continues to guide data protection, requiring companies to secure personal data, document usage, and respond to user requests within strict timeframes.

Identity Provider Architecture:

Component	Technology	Purpose	Security Level
Primary Authentication	OAuth 2.0 + JWT	User identity verification	High
Session Management	Redis-based tokens	Stateless session handling	High
Identity Storage	MongoDB with encryption	User profile persistence	Critical

User Identity Lifecycle Management:

The system manages user identities through a complete lifecycle from registration to account termination, ensuring proper logout mechanisms that thoroughly clear user-specific data from local storage when a user logs out, since there's no guarantee that the same user will log back in during the next session, failing to clear or safeguard the data could potentially expose sensitive information to unauthorised access.



6.4.1.2 Multi-Factor Authentication

MFA Implementation Strategy:

The application implements robust multi-factor authentication leveraging react-native-app-auth, an SDK for communicating with OAuth2 providers that wraps the native AppAuth-iOS and AppAuth-Android libraries and can support PKCE for enhanced security.

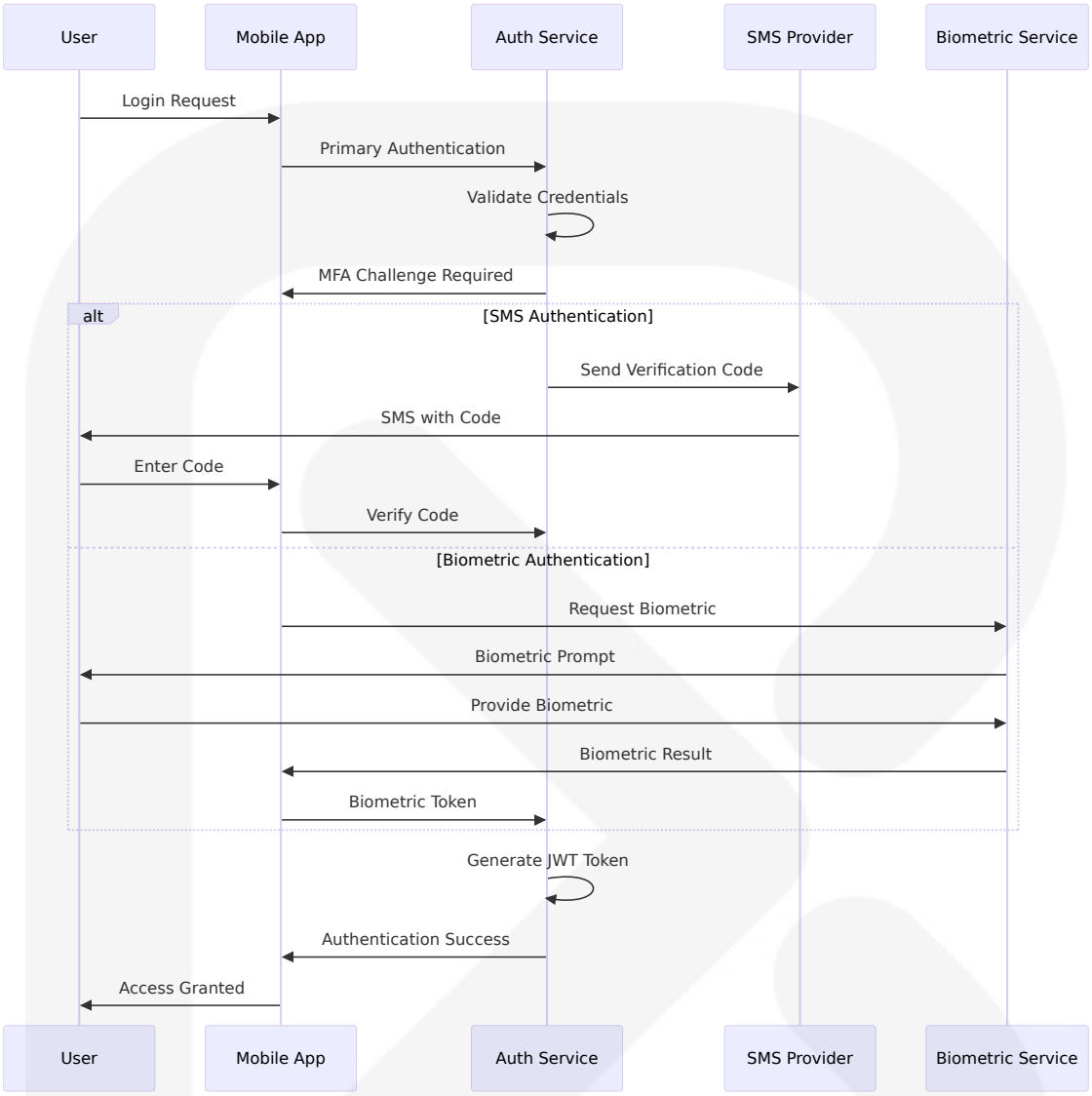
Authentication Factors Configuration:

Factor Type	Implementation	Use Case	Security Strength
Knowledge Factor	Password + PIN	Primary authentication	Medium
Possession Factor	Mobile device + SMS/App	Secondary verification	High
Inherence Factor	Biometric authentication	Convenience authentication	High

Biometric Authentication Integration:

The system utilizes expo-secure-store which provides a way to encrypt and securely store key-value pairs locally on the device, with each Expo project having a separate storage system and no access to the storage of other Expo projects for secure biometric data handling.

MFA Flow Implementation:



6.4.1.3 Session Management

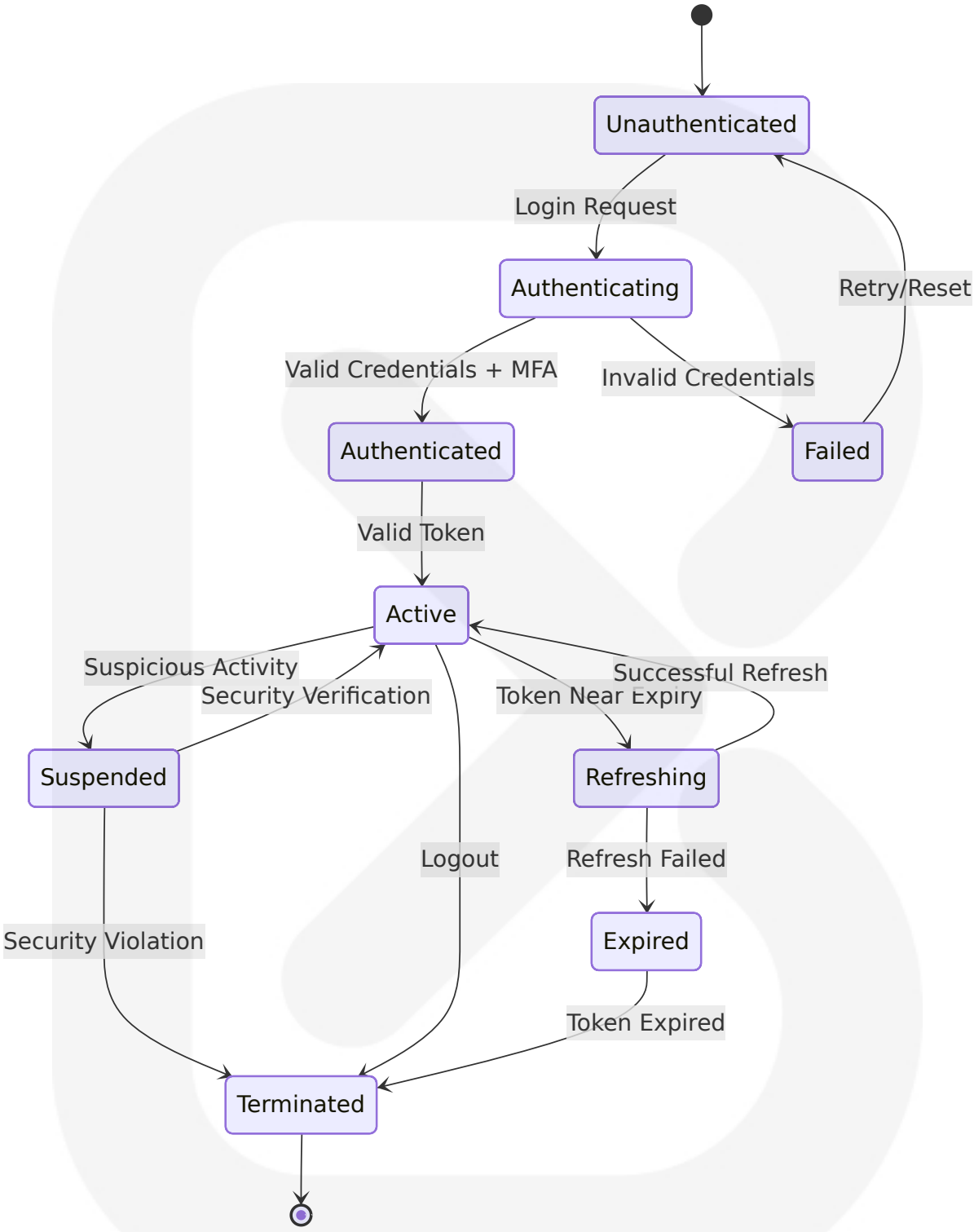
Stateless Session Architecture:

The system implements JWT (JSON Web Tokens) which provide a robust mechanism for authentication and authorization in web applications, enabling secure and scalable web services for stateless session management.

Session Security Configuration:

Session Attribute	Configuration	Security Rationale
Token Expiration	15 minutes access, 7 days refresh	Minimize exposure window
Token Rotation	Automatic refresh token rotation	Prevent token replay attacks
Session Binding	Device fingerprinting	Prevent session hijacking

Session Lifecycle Management:



6.4.1.4 Token Handling

JWT Token Architecture:
The application implements comprehensive token handling following JWT structure with Header containing metadata (algorithm used for signing), Payload storing user information (claims like user ID, roles), and Signature ensuring token integrity using a secret key, commonly used in Flask for authenticating users by issuing tokens upon login and verifying them for protected routes.

Token Security Implementation:

Token Type	Lifetime	Storage Location	Security Measures
Access Token	15 minutes	Memory only	Short-lived, frequent rotation
Refresh Token	7 days	Secure storage	Encrypted, single-use
API Key	Permanent	Secure storage	Rate limited, scope restricted

Secure Token Storage:
The system leverages expo-secure-store library which functions similarly to AsyncStorage but with added security, and for storing encryption keys safely using libraries like expo-secure-store or react-native-keychain for secure token persistence.

6.4.1.5 Password Policies

Password Security Framework:
The application enforces comprehensive password policies aligned with financial industry security standards and fair access, clear disclosures, and explainable decisions which are central to compliance in financial services, requiring pricing to be transparent and credit decisions to be supported by logic that teams can explain and regulators can test.

Password Policy Matrix:

Policy Component	Requirement	Enforcement Method	Compliance Standard
Minimum Length	12 characters	Client + server validation	Industry standard
Complexity	Mixed case, numbers, symbols	Pattern matching	Regulatory requirement
History	Last 12 passwords	Encrypted storage	Security best practice
Expiration	90 days for admin accounts	Automated notifications	Financial regulation

Password Security Implementation:

```
// Password Policy Enforcement
const passwordPolicy = {
  minLength: 12,
  requireUppercase: true,
  requireLowercase: true,
  requireNumbers: true,
  requireSymbols: true,
  preventCommonPasswords: true,
  preventUserInfoInPassword: true,
  historyCount: 12,
  maxAge: 90 * 24 * 60 * 60 * 1000, // 90 days in milliseconds

  validate: function(password, userInfo) {
    const errors = [];

    if (password.length < this.minLength) {
      errors.push(`Password must be at least ${this.minLength} characters
    }

    if (this.requireUppercase && !/[A-Z]/.test(password)) {
      errors.push('Password must contain uppercase letters');
    }

    if (this.requireLowercase && !/[a-z]/.test(password)) {
      errors.push('Password must contain lowercase letters');
    }
```

```
    if (this.requireNumbers && !/\d/.test(password)) {
      errors.push('Password must contain numbers');
    }

    if (this.requireSymbols && !/[!@#$%^&*(),.?":{}|<>]/.test(password))
      errors.push('Password must contain special characters');
    }

    return errors;
  }
};
```

6.4.2 AUTHORIZATION SYSTEM

6.4.2.1 Role-Based Access Control

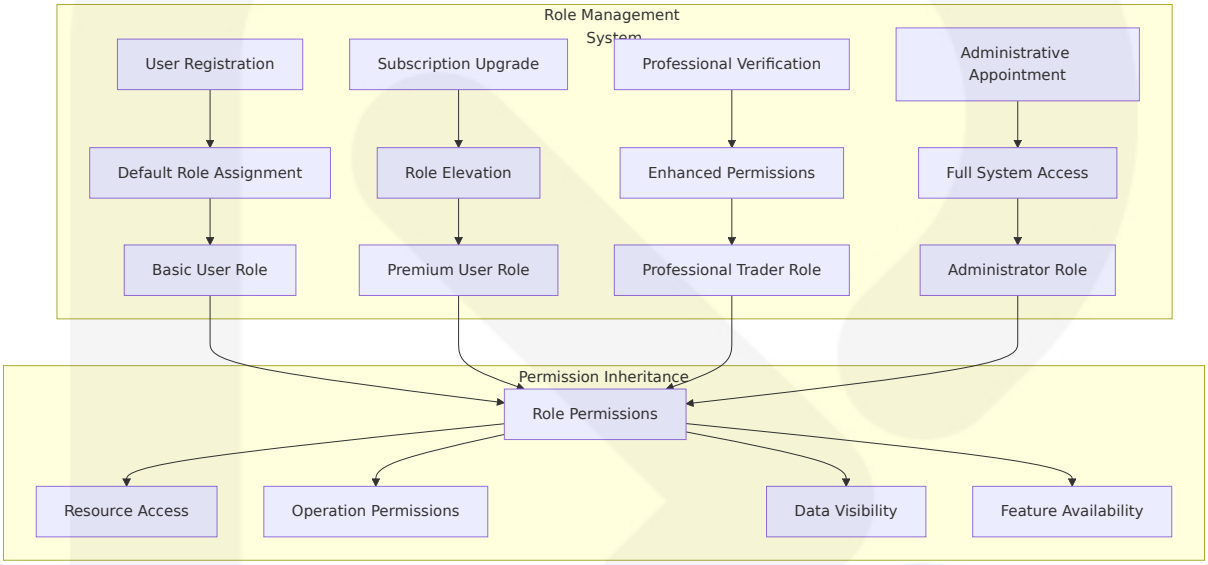
RBAC Architecture:
The system implements comprehensive role-based access control designed for financial trading applications, ensuring MiFID II requirements for collection and retention of large volumes of client and counterparty information, including all electronic communications data, which must be made available to regulators within 72 hours of a request.

Role Hierarchy Matrix:

Role	Market Data Access	Trading Operations	Portfolio Management	Administrative Functions
Basic User	Real-time quotes only	Paper trading	Personal portfolio	Profile management
Premium User	Full market data	Live trading	Advanced analytics	Subscription management
Professional Trader	Institutional data	High-frequency trading	Multi-portfolio	Risk management

Role	Market Data Access	Trading Operations	Portfolio Management	Administrative Functions
Administrator	All data sources	System monitoring	All portfolios	User management

Role Assignment Flow:



6.4.2.2 Permission Management

Granular Permission System:

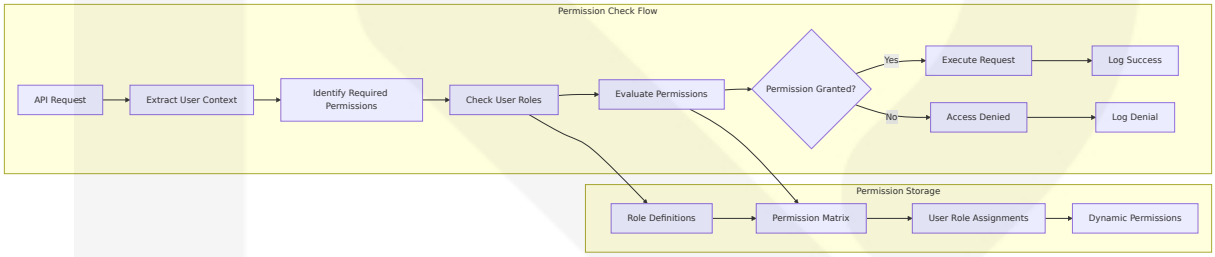
The application implements fine-grained permissions aligned with MiFID II rules affecting Direct Electronic Access (DEA), algorithmic trading, high frequency trading (HFT) and market making, which define algorithmic trading as trading where a computer algorithm automatically determines individual parameters of orders with limited or no human intervention.

Permission Categories:

Permission Category	Scope	Granularity Level	Regulatory Compliance
Market Data Access	Real-time/delayed data	Per exchange/instrument	Market data licensing

Permission Category	Scope	Granularity Level	Regulatory Compliance
Trading Operations	Order types/sizes	Per account/strategy	MiFID II compliance
Portfolio Analytics	Performance metrics	Per portfolio/timeframe	Privacy regulations
System Administration	User/system management	Per function/resource	Audit requirements

Permission Enforcement Architecture:



6.4.2.3 Resource Authorization

Resource-Level Security:

The system implements comprehensive resource authorization ensuring the Least Privilege Principle, a fundamental concept in app security that aims to minimise the risk of unauthorised access and data breaches.

Resource Authorization Matrix:

Resource Type	Authorization Method	Access Control	Audit Requirements
Market Data	Subscription-based	Real-time validation	Access logging
Trading Accounts	Ownership verification	Account binding	Transaction audit
Portfolio Data	User association	Privacy controls	Data access logs

Resource Type	Authorization Method	Access Control	Audit Requirements
System Configuration	Administrative rights	Change approval	Configuration audit

6.4.2.4 Policy Enforcement Points

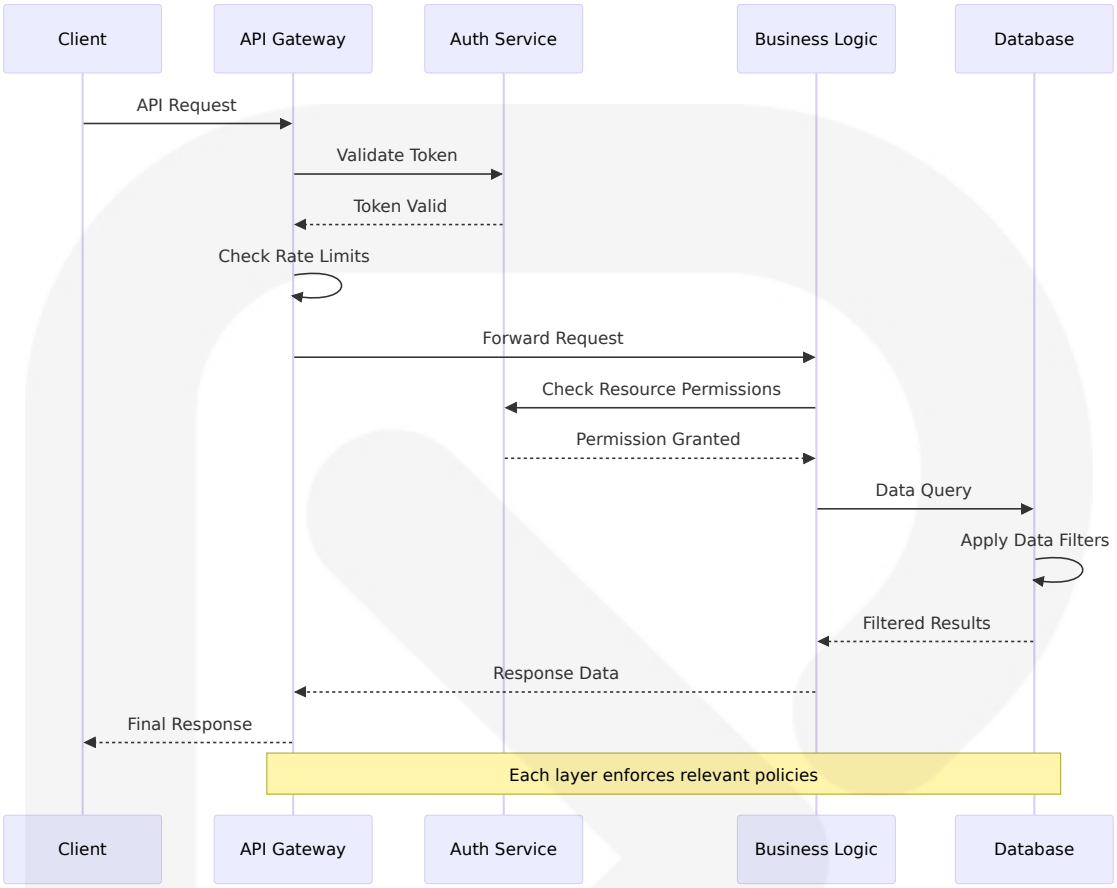
Distributed Authorization Architecture:

The system implements multiple policy enforcement points throughout the application architecture to ensure comprehensive security coverage.

Enforcement Point Configuration:

Enforcement Point	Location	Policy Type	Response Time
API Gateway	Entry point	Authentication/rate limiting	<100ms
Service Layer	Business logic	Resource authorization	<50ms
Data Layer	Database access	Data privacy/compliance	<25ms
Client Layer	UI components	Feature visibility	<10ms

Policy Enforcement Flow:



6.4.2.5 Audit Logging

Comprehensive Audit Framework:

The system maintains detailed audit logs to support MiFID II requirements for collection and retention of large volumes of client and counterparty information, with data that must be made available to regulators within 72 hours of a request, and client email correspondence recorded and archived for up to five years, telephone calls for as long as seven years.

Audit Event Categories:

Event Category	Retention Period	Log Level	Regulatory Requirement
Authentication Events	7 years	All attempts	Financial audit compliance

Event Category	Retention Period	Log Level	Regulatory Requirement
Trading Operations	5 years	All transactions	MiFID II compliance
Data Access	3 years	Sensitive data only	Privacy regulations
System Changes	7 years	All modifications	SOX compliance

Audit Log Structure:

```
// Comprehensive Audit Log Entry
const auditLogEntry = {
  timestamp: new Date().toISOString(),
  eventId: generateUniqueId(),
  userId: user.id,
  sessionId: session.id,
  eventType: 'TRADING_OPERATION',
  eventCategory: 'ORDER_PLACEMENT',
  resource: {
    type: 'TRADING_ACCOUNT',
    id: account.id,
    symbol: 'EURUSD'
  },
  action: 'CREATE_ORDER',
  outcome: 'SUCCESS',
  details: {
    orderType: 'MARKET',
    quantity: 10000,
    price: null,
    stopLoss: 1.0850,
    takeProfit: 1.0950
  },
  ipAddress: request.ip,
  userAgent: request.headers['user-agent'],
  geolocation: {
    country: 'US',
    region: 'NY'
  },
  riskScore: 2.5,
```

```
    complianceFlags: ['MiFID_II_REPORTABLE'],
    retentionUntil: new Date(Date.now() + 5 * 365 * 24 * 60 * 60 * 1000) //
};
```

6.4.3 DATA PROTECTION

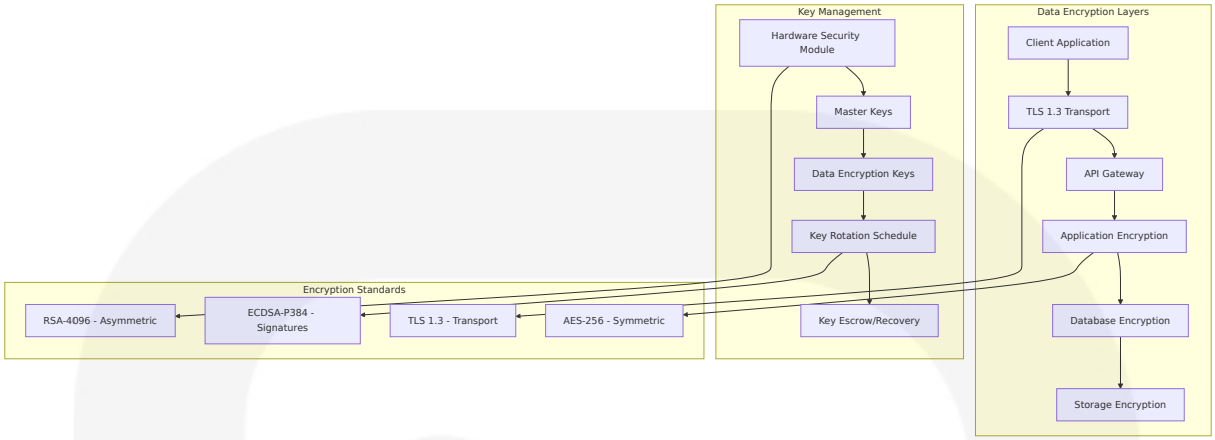
6.4.3.1 Encryption Standards

Multi-Layer Encryption Architecture:
The system implements comprehensive encryption standards following TLS encryption technology which protects from having information intercepted during transmission to TradingView or payment processors, with secure pages displaying the "lock" icon confirming a secure and encrypted connection has been established.

Encryption Implementation Matrix:

Data State	Encryption Standard	Key Management	Performance Impact
Data in Transit	TLS 1.3 with AES-256-GCM	Certificate-based	Minimal (<5ms)
Data at Rest	AES-256-CBC	Hardware Security Module	Low (<10ms)
Application Layer	ChaCha20-Poly1305	Key derivation functions	Minimal (<2ms)
Database Encryption	Transparent Data Encryption	Automated key rotation	Medium (<20ms)

Encryption Architecture Flow:



6.4.3.2 Key Management

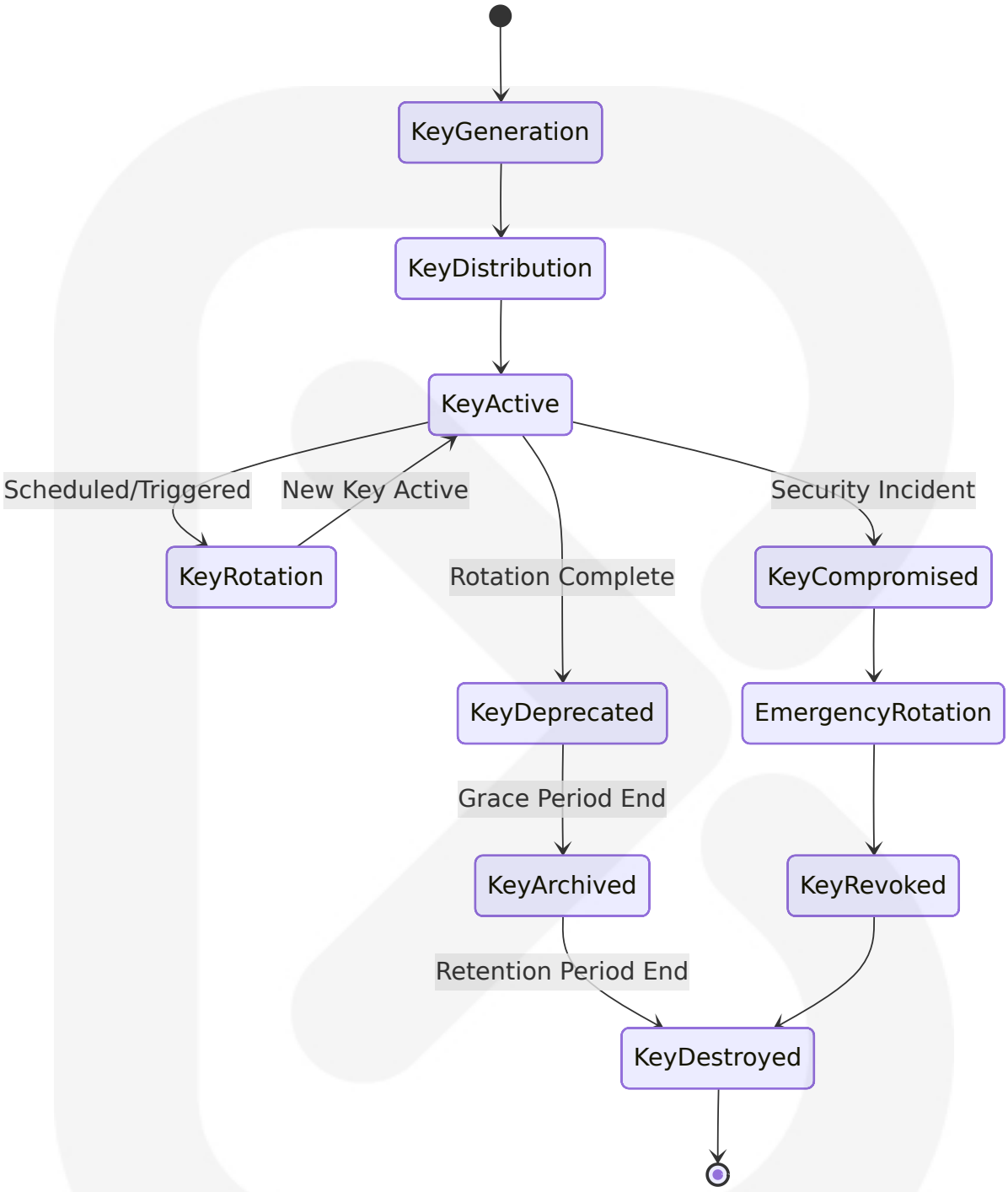
Enterprise Key Management System:

The system implements robust key management following platform-specific secure storage using libraries like react-native-keychain or expo-secure-store, which provide unified, cross-platform API to interact directly with iOS Keychain and Android Keystore, with encryption of larger data sets using AES-256 and secure generation and storage of encryption keys within the Keychain/Keystore.

Key Lifecycle Management:

Key Type	Generation Method	Rotation Schedule	Storage Location
Master Keys	Hardware Security Module	Annual	HSM secure enclave
Data Encryption Keys	PBKDF2 derivation	Quarterly	Encrypted key store
API Keys	Cryptographically secure random	Monthly	Secure configuration
Session Keys	Ephemeral generation	Per session	Memory only

Key Management Workflow:



6.4.3.3 Data Masking Rules

Dynamic Data Masking Framework:

The system implements comprehensive data masking to protect sensitive information while maintaining application functionality, ensuring

compliance with GDPR requirements for companies to secure personal data, document usage, and respond to user requests within strict timeframes.

Data Masking Configuration:

Data Type	Masking Method	Visibility Rules	Compliance Requirement
Personal Identifiers	Partial masking	Role-based visibility	GDPR Article 32
Financial Data	Format-preserving encryption	Need-to-know basis	PCI DSS Level 1
Trading Positions	Aggregation masking	Portfolio-level only	MiFID II transparency
Communication Data	Redaction masking	Compliance team only	Financial regulations

Data Masking Implementation:

```
// Dynamic Data Masking Service
class DataMaskingService {
  constructor() {
    this.maskingRules = {
      email: (email, userRole) => {
        if (userRole === 'admin') return email;
        const [local, domain] = email.split('@');
        return `${local.substring(0, 2)}***@${domain}`;
      },

      accountNumber: (account, userRole) => {
        if (userRole === 'owner' || userRole === 'admin') return account
        return `****${account.slice(-4)}`;
      },

      tradingBalance: (balance, userRole) => {
        if (userRole === 'owner' || userRole === 'admin') return balance
        return '***.**';
      },
```

```
      personalId: (id, userRole) => {
        if (userRole === 'compliance' || userRole === 'admin') return id
        return `***-**-${id.slice(-4)}`;
      }
    };
  }

  maskData(data, dataType, userRole, context) {
    const maskingRule = this.maskingRules[dataType];
    if (!maskingRule) return data;

    // Apply contextual masking based on user role and access context
    return maskingRule(data, userRole, context);
  }
}
```

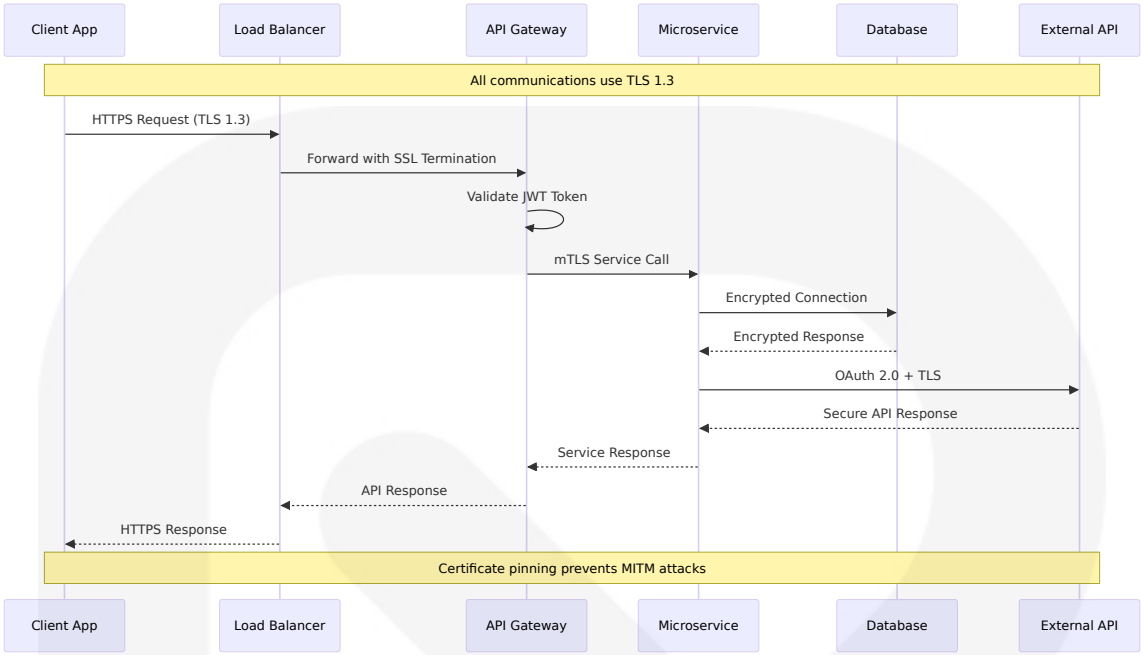
6.4.3.4 Secure Communication

End-to-End Security Architecture:
The system ensures secure communication across all channels, implementing SSL encryption for all APIs, which protects against requested data being read in plain text between when it leaves the server and before it reaches the client.

Communication Security Matrix:

Communication Channel	Security Protocol	Authentication Method	Monitoring Level
Client-Server API	TLS 1.3 + Certificate Pinning	JWT + API Keys	Full logging
Server-to-Server	mTLS + Service Mesh	Service certificates	Transaction logging
Database Connections	TLS + Connection Pooling	Certificate-based	Query logging
External API Calls	TLS + API Key Rotation	OAuth 2.0 + PKCE	Rate limit monitoring

Secure Communication Flow:



6.4.3.5 Compliance Controls

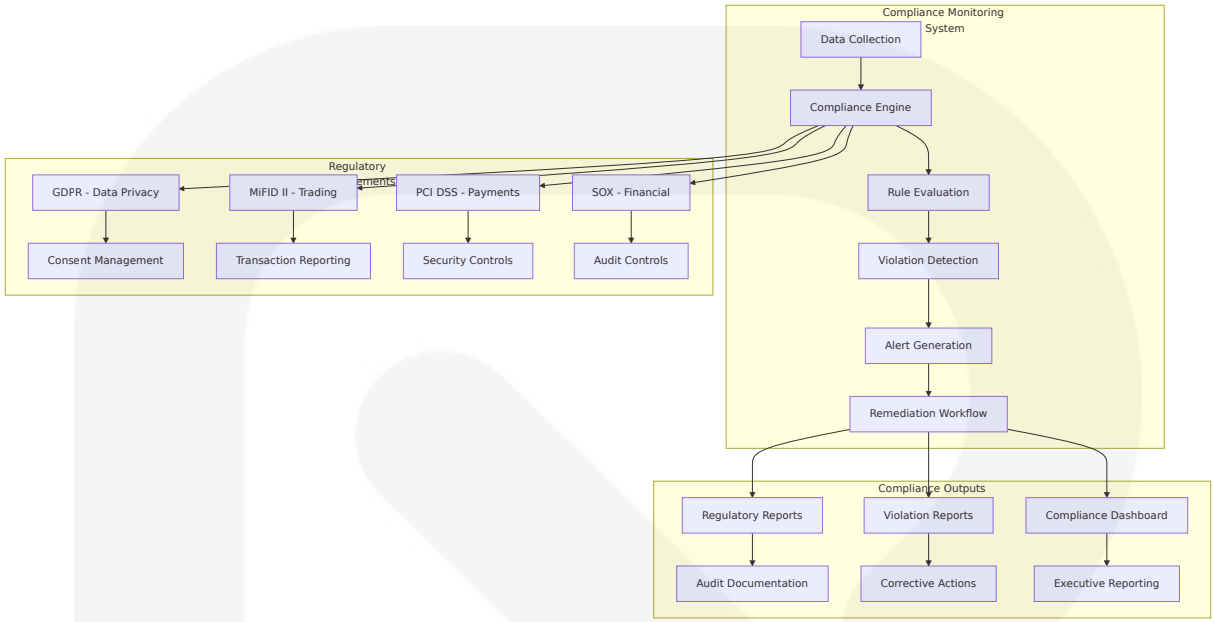
Regulatory Compliance Framework:

The system implements comprehensive compliance controls addressing GDPR penalties reaching up to â‚¬20 million or 4% of global turnover for significant non-compliance, with MiFID II requiring robust new reporting, operational and technical infrastructures, and GDPR having vast scope open to interpretation in certain instances.

Compliance Control Matrix:

Regulation	Control Type	Implementation	Monitoring Frequency
GDPR	Data privacy controls	Consent management, data minimization	Continuous
MiFID II	Transaction reporting	Automated regulatory reporting	Real-time
PCI DSS	Payment security	Tokenization, secure processing	Daily
SOX	Financial controls	Audit trails, change management	Monthly

Compliance Monitoring Architecture:



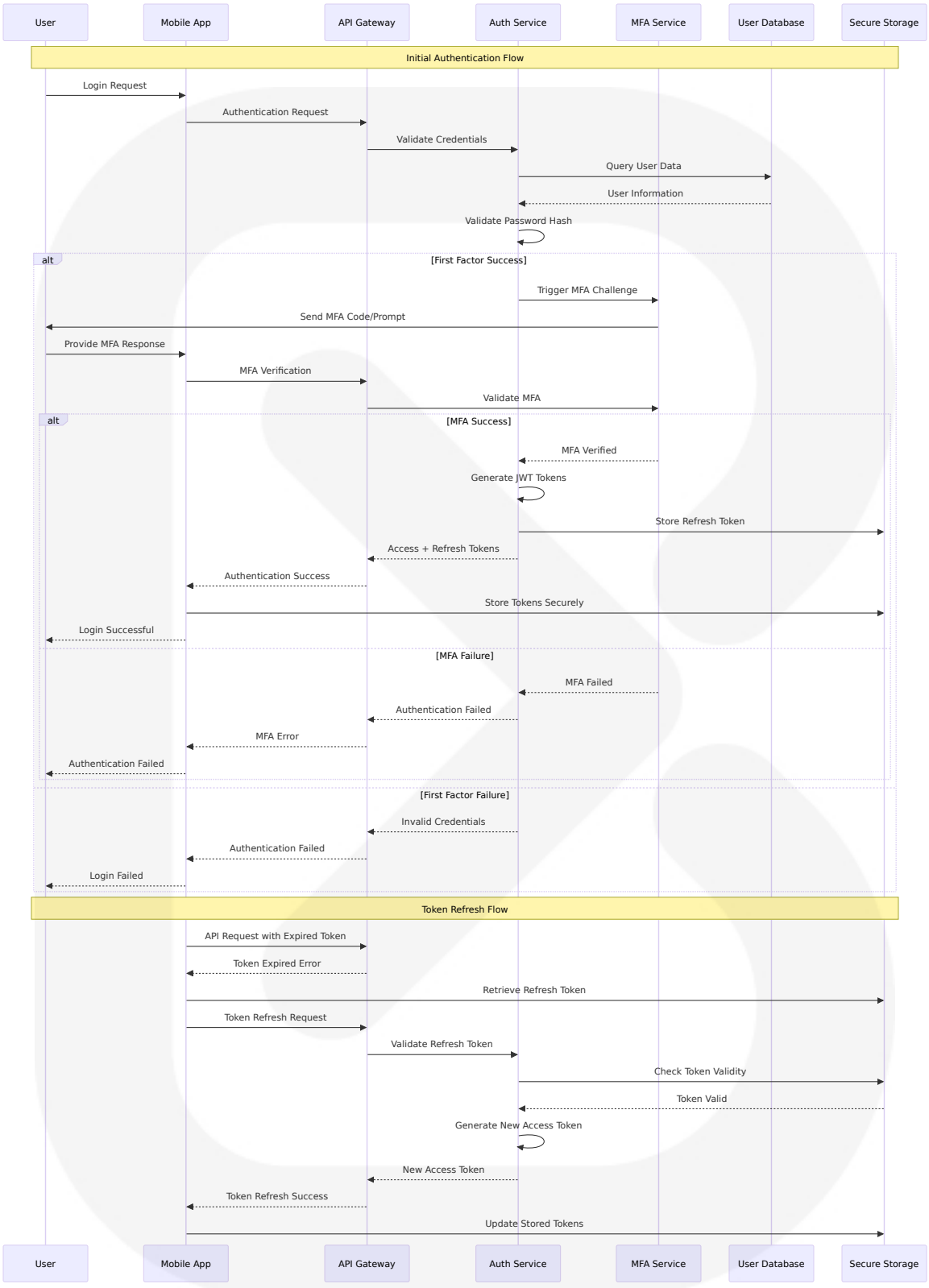
Automated Compliance Validation:

```
// Compliance Validation Service
class ComplianceValidator {
  constructor() {
    this.regulations = {
      GDPR: {
        dataRetention: this.validateDataRetention,
        consentManagement: this.validateConsent,
        dataPortability: this.validatePortability
      },
      MIFID_II: {
        transactionReporting: this.validateTransactionReporting,
        recordKeeping: this.validateRecordKeeping,
        clientCommunication: this.validateCommunication
      },
      PCI_DSS: {
        dataEncryption: this.validateEncryption,
        accessControl: this.validateAccess,
        networkSecurity: this.validateNetwork
      }
    };
  }

  async validateCompliance(operation, data, context) {
```

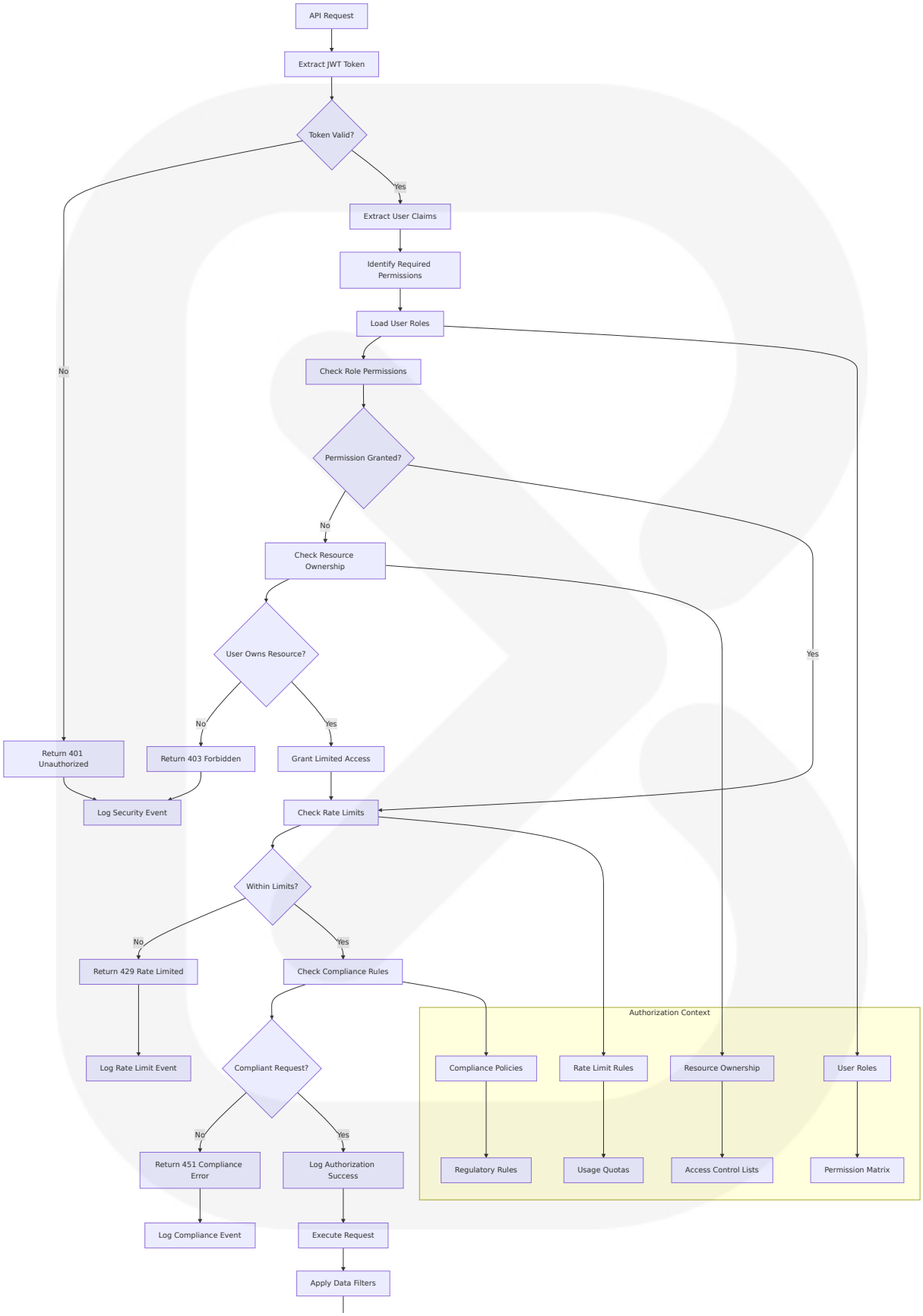
```
    const violations = [];

    for (const [regulation, validators] of Object.entries(this.regulatio
      for (const [control, validator] of Object.entries(validators)) {
        try {
          const result = await validator(operation, data, context);
          if (!result.compliant) {
            violations.push({
              regulation,
              control,
              violation: result.violation,
              severity: result.severity,
              remediation: result.remediation
            });
          }
        } catch (error) {
          violations.push({
            regulation,
            control,
            violation: `Validation error: ${error.message}`,
            severity: 'HIGH',
            remediation: 'Fix validation logic'
          });
        }
      }
    }

    return {
      compliant: violations.length === 0,
      violations,
      timestamp: new Date().toISOString()
    };
  }
}
```

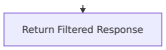
6.4.4 SECURITY ARCHITECTURE DIAGRAMS

6.4.4.1 Authentication Flow Diagram

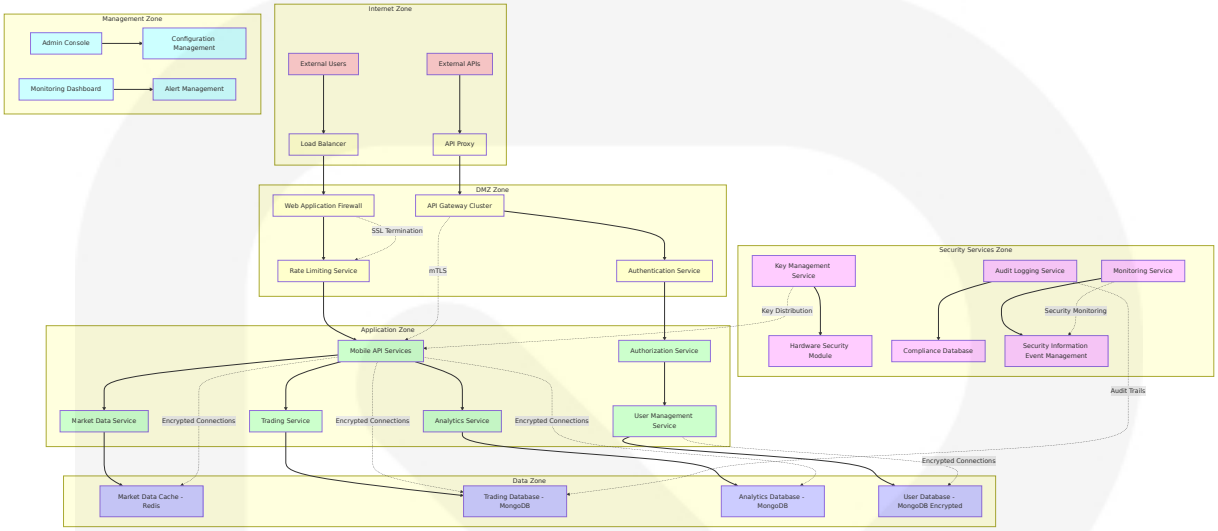


6.4.4.2 Authorization Flow Diagram


API Request
Extract JWT Token
Token Valid?
Yes
No
Extract User Claims
Identify Required Permissions
Load User Roles
Check Role Permissions
Permission Granted?
Yes
No
Check Resource Ownership
User Owns Resource?
Yes
No
Grant Limited Access
Return 403 Forbidden
Return 401 Unauthorized
Log Security Event
Check Rate Limits
Within Limits?
Yes
No
Check Compliance Rules
Return 429 Rate Limited
Log Rate Limit Event
Compliant Request?
Yes
No
Log Authorization Success
Return 451 Compliance Error
Log Compliance Event
Execute Request
Apply Data Filters
Authorization Context
Compliance Policies
Rate Limit Rules
Resource Ownership
User Roles
Regulatory Rules
Usage Quotas
Access Control Lists
Permission Matrix



6.4.4.3 Security Zone Diagram



6.4.5 SECURITY IMPLEMENTATION SUMMARY

The forex trading application implements a comprehensive security architecture that addresses the unique challenges of financial trading platforms while ensuring compliance with multiple regulatory frameworks. The system leverages modern security practices integrated from the get-go, with Expo providing a solid foundation while requiring developers to ensure applications are not just functional but also secure.

Key Security Achievements:

Security Domain	Implementation	Compliance Standard	Risk Mitigation
Authentication	Multi-factor with biometrics	Financial industry standards	Identity theft prevention
Authorization	Role-based with fine-grained permissions	MiFID II compliance	Unauthorized access prevention
Data Protection	Multi-layer encryption with HSM	GDPR + PCI DSS	Data breach prevention

Security Domain	Implementation	Compliance Standard	Risk Mitigation
Communication Security	TLS 1.3 with certificate pinning	Industry best practices	Man-in-the-middle prevention

The architecture ensures adherence to robust mobile app security best practices as a business-critical necessity, moving beyond generic advice to provide detailed security domains while maintaining the performance and user experience requirements of a real-time trading application.

Continuous Security Monitoring:
The system implements continuous security processes with regular review of code, dependencies, and Expo SDK updates to ensure the application remains secure against emerging threats, providing a robust foundation for secure forex trading operations.

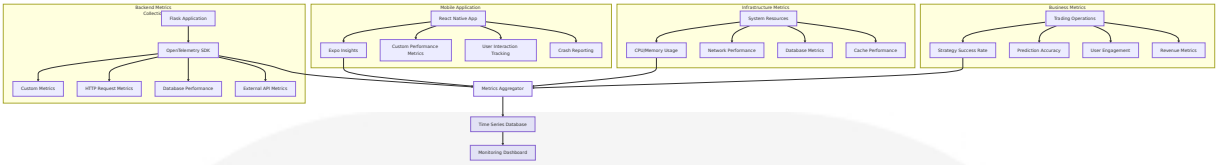
6.5 MONITORING AND OBSERVABILITY

6.5.1 MONITORING INFRASTRUCTURE

6.5.1.1 Metrics Collection

The forex trading application implements comprehensive metrics collection leveraging modern observability practices for React Native and Flask-based systems. Expo provides built-in monitoring and observability with Insights, allowing you to see how your app is doing with built-in monitoring and observability with Insights, while the backend utilizes OpenTelemetry and Elastic APM for full observability (logs, traces, and metrics) in a Python (Flask) application.

Metrics Collection Architecture:



Core Metrics Categories:

Metric Category	Collection Method	Frequency	Retention Period
Application Performance	OpenTelemetry instrumentation	Real-time	90 days
Business KPIs	Custom event tracking	Real-time	2 years
Infrastructure Health	System monitoring agents	30 seconds	1 year
User Experience	Mobile analytics SDK	Real-time	6 months

Mobile Application Metrics:

Expo Insights shows which platforms your users are using and which versions they're currently using, keeps track of your app's usage with API function usage and hosting requests, and shows error rates with over the air updates and crash stats with hosting.

Backend Metrics Implementation:

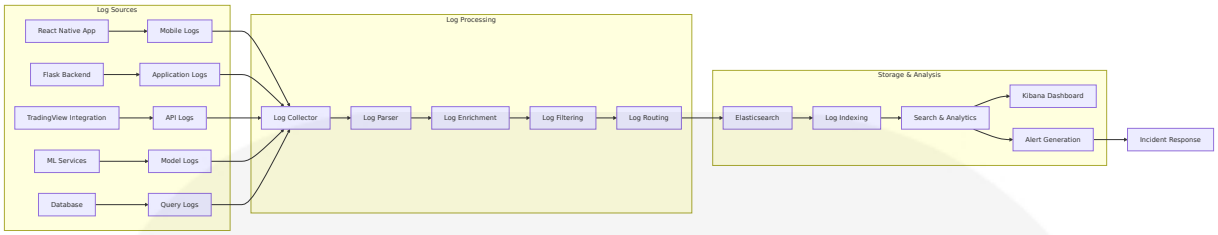
```
# OpenTelemetry Metrics Collection for Flask
from opentelemetry import metrics
from opentelemetry.sdk.metrics import MeterProvider
from opentelemetry.sdk.metrics.export import PeriodicExportingMetricReade
from opentelemetry.exporter.otlp.proto.grpc.metric_exporter import OTLPMe

#### Initialize metrics provider
metrics.set_meter_provider(MeterProvider(
    metric_readers=[
        PeriodicExportingMetricReader(
            OTLPMetricExporter(endpoint="http://localhost:4317"),
            export_interval_millis=30000
        )
    ]
```

```
))

meter = metrics.get_meter("forex-trading-app")

#### Custom business metrics
trading_strategy_counter = meter.create_counter(
    name="trading_strategies_generated",
    description="Number of trading strategies generated",
    unit="1"
)

prediction_accuracy_histogram = meter.create_histogram(
    name="ml_prediction_accuracy",
    description="ML model prediction accuracy distribution",
    unit="percentage"
)

api_response_time_histogram = meter.create_histogram(
    name="api_response_time",
    description="API response time distribution",
    unit="ms"
)

#### Trading-specific metrics
active_users_gauge = meter.create_up_down_counter(
    name="active_trading_sessions",
    description="Number of active trading sessions",
    unit="1"
)
```

6.5.1.2 Log Aggregation

The system implements centralized log aggregation following structured logs enriched with OTEL context (e.g., trace IDs) and exported via the LoggingOTLPHandler for comprehensive observability across all system components.

Log Aggregation Architecture:



Structured Logging Configuration:

Log Level	Use Case	Retention	Processing
ERROR	System failures, exceptions	1 year	Real-time alerts
WARN	Performance degradation	6 months	Daily analysis
INFO	Business events, user actions	3 months	Weekly reports
DEBUG	Development troubleshooting	30 days	On-demand

Flask Application Logging:

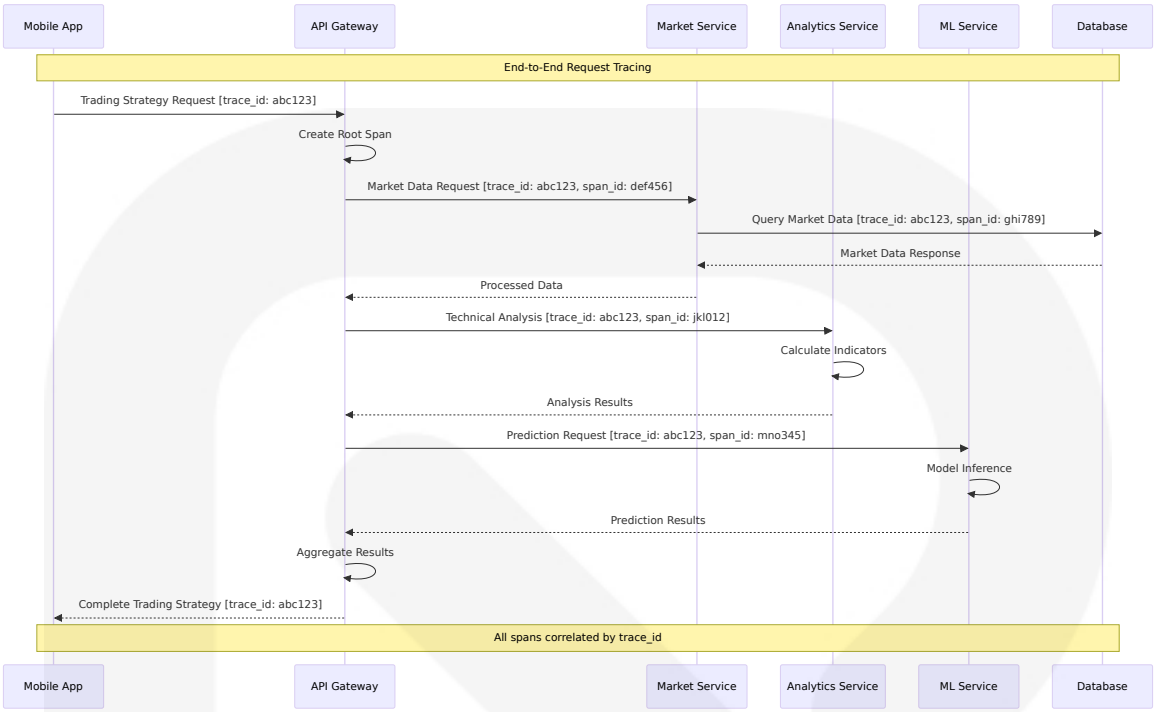
```
import logging
from opentelemetry.instrumentation.logging import LoggingInstrumentor
from opentelemetry.exporter.otlp.proto.grpc._log_exporter import OTLPLogE
from opentelemetry._logs import set_logger_provider
from opentelemetry.sdk._logs import LoggerProvider
from opentelemetry.sdk._logs.export import BatchLogRecordProcessor

#### Configure OpenTelemetry logging
set_logger_provider(LoggerProvider())
logger_provider = logging.getLoggerProvider()
logger_provider.add_log_record_processor(
    BatchLogRecordProcessor(
        OTLPLogExporter(endpoint="http://localhost:4317")
    )
)

#### Instrument logging to add trace context
LoggingInstrumentor().instrument(set_logging_format=True)

#### Structured logging format
```

```
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s - trace_
    handlers=[
        logging.StreamHandler(),
        logging.FileHandler('/var/log/forex-app/application.log')
    ]
)

logger = logging.getLogger(__name__)

#### Business event logging
def log_trading_event(event_type, user_id, symbol, details):
    logger.info(
        "Trading event occurred",
        extra={
            'event_type': event_type,
            'user_id': user_id,
            'symbol': symbol,
            'details': details,
            'component': 'trading_engine'
        }
    )
```

6.5.1.3 Distributed Tracing

The system implements traces manually instrumented using OpenTelemetry's Tracer to track requests across Flask routes and background tasks, providing end-to-end visibility across the distributed architecture.

Distributed Tracing Flow:



Tracing Implementation:

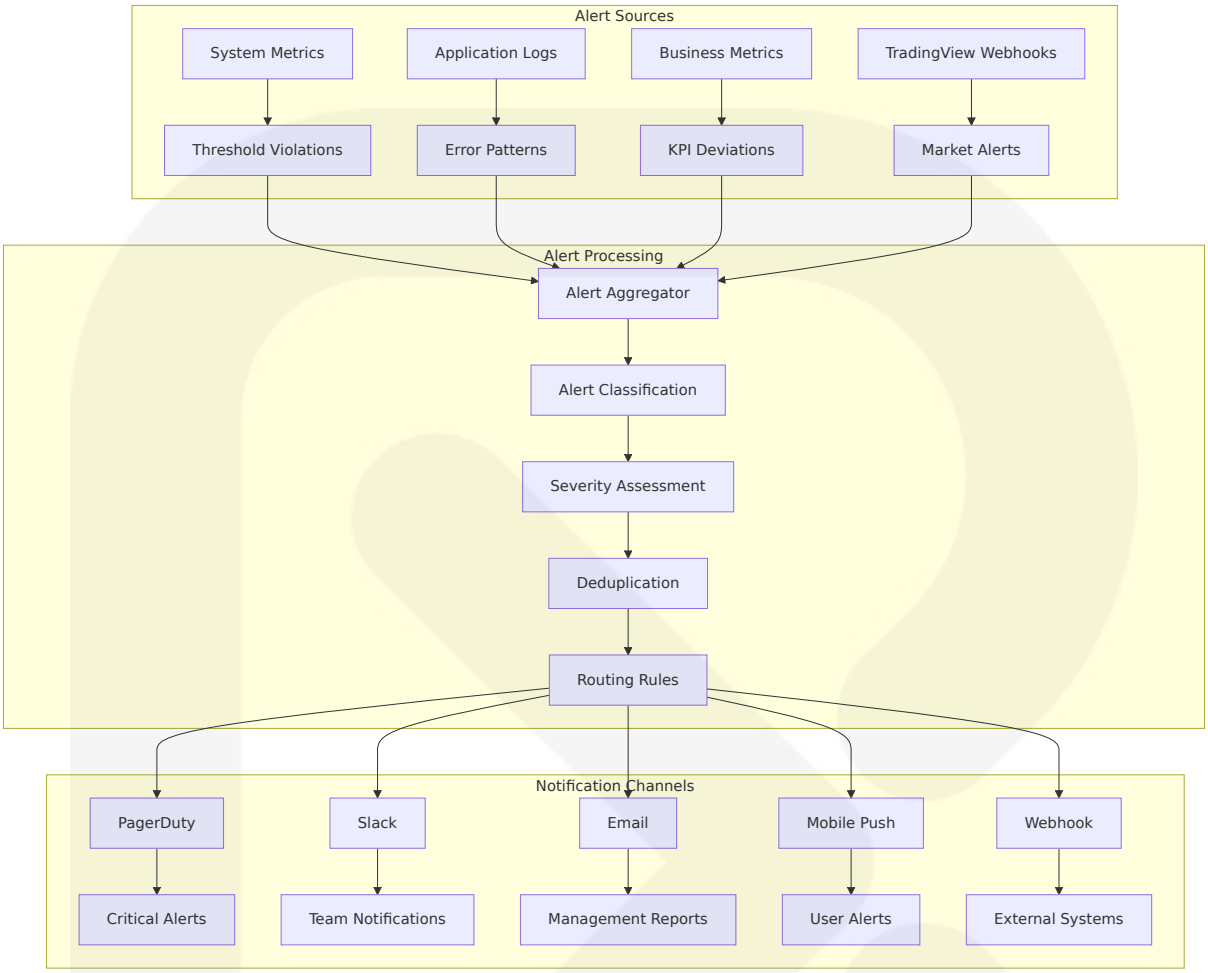
```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpa
from opentelemetry.instrumentation.flask import FlaskInstrumentor
from opentelemetry.instrumentation.requests import RequestsInstrumentor

#### Configure tracing
trace.set_tracer_provider(TracerProvider())
tracer = trace.get_tracer(__name__)

#### Add OTLP exporter
trace.get_tracer_provider().add_span_processor(
    BatchSpanProcessor(
        OTLPSpanExporter(endpoint="http://localhost:4317")
    )
)

#### Auto-instrument Flask and requests
FlaskInstrumentor().instrument_app(app)
RequestsInstrumentor().instrument()
```

```python
#### Custom span creation for business logic
@app.route('/api/trading-strategy')
def generate_trading_strategy():
    with tracer.start_as_current_span("generate_trading_strategy") as spa
        symbol = request.args.get('symbol')
        span.set_attribute("trading.symbol", symbol)
        span.set_attribute("user.id", get_current_user_id())

#### Market data analysis span
        with tracer.start_as_current_span("market_data_analysis") as marl
            market_data = fetch_market_data(symbol)
            market_span.set_attribute("market.data_points", len(market_da

#### ML prediction span
        with tracer.start_as_current_span("ml_prediction") as ml_span:
            prediction = generate_ml_prediction(market_data)
            ml_span.set_attribute("ml.confidence_score", prediction.confi
            ml_span.set_attribute("ml.model_version", prediction.model_ve

#### Strategy generation span
        with tracer.start_as_current_span("strategy_generation") as strat
            strategy = create_trading_strategy(market_data, prediction)
            strategy_span.set_attribute("strategy.risk_score", strategy.r
            strategy_span.set_attribute("strategy.success_probability", s

        span.set_attribute("strategy.generated", True)
        return jsonify(strategy.to_dict())
```

6.5.1.4 Alert Management

The system implements comprehensive alert management covering both technical system alerts and business-critical trading alerts, with integration to TradingView alerts that notify you of chart symbol price changes and help traders and investors stay on top of key market events by sending timely notifications.

Alert Management Architecture:



Alert Severity Matrix:

Severity Level	Response Time	Escalation	Notification Method
Critical	<5 minutes	Immediate	PagerDuty + SMS
High	<15 minutes	30 minutes	Slack + Email
Medium	<1 hour	2 hours	Email
Low	<4 hours	Next business day	Dashboard only

Alert Configuration:

```
# Alert management system
class AlertManager:
    def __init__(self):
```

```
        self.alert_rules = {
            'api_response_time': {
                'threshold': 2000,  # 2 seconds
                'severity': 'high',
                'condition': 'greater_than'
            },
            'ml_prediction_accuracy': {
                'threshold': 60,  # 60%
                'severity': 'medium',
                'condition': 'less_than'
            },
            'trading_strategy_success_rate': {
                'threshold': 65,  # 65%
                'severity': 'high',
                'condition': 'less_than'
            },
            'system_error_rate': {
                'threshold': 1,  # 1%
                'severity': 'critical',
                'condition': 'greater_than'
            }
        }

    def process_metric(self, metric_name, value, context=None):
        if metric_name in self.alert_rules:
            rule = self.alert_rules[metric_name]
            if self.evaluate_condition(value, rule):
                self.trigger_alert(metric_name, value, rule['severity'],

    def trigger_alert(self, metric, value, severity, context):
        alert = {
            'timestamp': datetime.utcnow().isoformat(),
            'metric': metric,
            'value': value,
            'severity': severity,
            'context': context,
            'alert_id': str(uuid.uuid4())
        }

        # Route based on severity
        if severity == 'critical':
            self.send_pagerduty_alert(alert)
            self.send_slack_alert(alert)
```

```
        elif severity == 'high':
            self.send_slack_alert(alert)
            self.send_email_alert(alert)
        elif severity == 'medium':
            self.send_email_alert(alert)

        # Log all alerts
        logger.warning(f"Alert triggered: {alert}")
```

6.5.1.5 Dashboard Design

The monitoring dashboard provides comprehensive visibility into system health, business metrics, and trading performance, leveraging Datadog's customizable dashboards to create a specific "React Native Performance" view, combining metrics like screen load times, JS error rates, and API latency on a single screen.

Dashboard Architecture:

Dashboard Configuration Matrix:

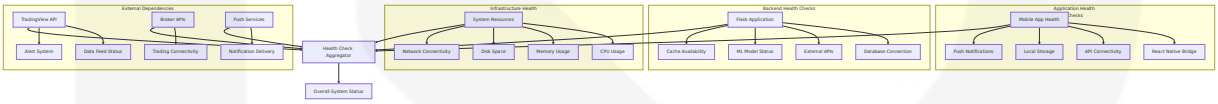
Dashboard Type	Update Frequency	Retention	Access Level
Executive Summary	1 hour	2 years	C-level, Management
Operations	30 seconds	6 months	DevOps, Engineering
Trading Performance	Real-time	1 year	Trading team, Analysts
Technical Metrics	10 seconds	3 months	Engineering, SRE

6.5.2 OBSERVABILITY PATTERNS

6.5.2.1 Health Checks

The system implements comprehensive health checks across all components to ensure system reliability and enable proactive issue detection. React Native applications combine JavaScript and native code, making monitoring and debugging more complex, with observability helping catch issues early and ensure that part of the app meets performance and reliability standards.

Health Check Architecture:



Health Check Implementation:

```
from flask import Flask, jsonify
import redis
import pymongo
import requests
from datetime import datetime, timedelta

app = Flask(__name__)

class HealthChecker:
    def __init__(self):
        self.checks = {
            'database': self.check_database,
            'cache': self.check_cache,
            'tradingview_api': self.check_tradingview_api,
            'ml_models': self.check_ml_models,
            'external_apis': self.check_external_apis
        }

    def check_database(self):
        try:
            client = pymongo.MongoClient('mongodb://localhost:27017/')
            client.admin.command('ping')
            return {'status': 'healthy', 'response_time': 0.05}
        except Exception as e:
            return {'status': 'unhealthy', 'error': str(e)}
```

```
    def check_cache(self):
        try:
            r = redis.Redis(host='localhost', port=6379)
            r.ping()
            return {'status': 'healthy', 'response_time': 0.01}
        except Exception as e:
            return {'status': 'unhealthy', 'error': str(e)}

    def check_tradingview_api(self):
        try:
            # Check TradingView API availability
            response = requests.get('https://api.tradingview.com/health'
            if response.status_code == 200:
                return {'status': 'healthy', 'response_time': response.e
            else:
                return {'status': 'degraded', 'status_code': response.sta
        except Exception as e:
            return {'status': 'unhealthy', 'error': str(e)}

    def check_ml_models(self):
        try:
            # Check if ML models are loaded and responsive
            model_status = {
                'prediction_model': self.ping_ml_service('/health/predic
                'risk_model': self.ping_ml_service('/health/risk')
            }

            all_healthy = all(status['status'] == 'healthy' for status in
            return {
                'status': 'healthy' if all_healthy else 'degraded',
                'models': model_status
            }
        except Exception as e:
            return {'status': 'unhealthy', 'error': str(e)}

    def get_overall_health(self):
        results = {}
        overall_status = 'healthy'

        for check_name, check_func in self.checks.items():
            result = check_func()
            results[check_name] = result
```

```
            if result['status'] == 'unhealthy':
                overall_status = 'unhealthy'
            elif result['status'] == 'degraded' and overall_status == 'he
                overall_status = 'degraded'

        return {
            'status': overall_status,
            'timestamp': datetime.utcnow().isoformat(),
            'checks': results
        }

@app.route('/health')
def health_check():
    checker = HealthChecker()
    health_status = checker.get_overall_health()

    status_code = 200
    if health_status['status'] == 'degraded':
        status_code = 503
    elif health_status['status'] == 'unhealthy':
        status_code = 503

    return jsonify(health_status), status_code
```

6.5.2.2 Performance Metrics

The system tracks comprehensive app performance metrics from network requests to UI render times, with Dynatrace's capabilities particularly well-suited for tracking complex app performance metrics in a hybrid environment like React Native, capturing every user click, tap, and swipe to analyze user journeys.

Performance Metrics Framework:

Metric Category	Key Indicators	Target Values	Monitoring Frequency
API Performance	Response time, throughput, error rate	<2s, >100 RPS, <1%	Real-time

Metric Category	Key Indicators	Target Values	Monitoring Frequency
Mobile Performance	App launch time, screen transitions	<3s, <500ms	Real-time
ML Model Performance	Inference time, accuracy, confidence	<5s, >70%, >60%	Per prediction
Database Performance	Query time, connection pool usage	<100ms, <80%	Real-time

Performance Monitoring Implementation:

```
import time
import functools
from opentelemetry import metrics
from opentelemetry.sdk.metrics import MeterProvider

#### Performance metrics collection
meter = metrics.get_meter("forex-app-performance")

#### API performance metrics
api_response_time = meter.create_histogram(
    name="api_response_time_ms",
    description="API response time in milliseconds",
    unit="ms"
)

api_request_count = meter.create_counter(
    name="api_requests_total",
    description="Total number of API requests",
    unit="1"
)

#### ML model performance metrics
ml_inference_time = meter.create_histogram(
    name="ml_inference_time_ms",
    description="ML model inference time",
    unit="ms"
)

ml_prediction_accuracy = meter.create_histogram(
```

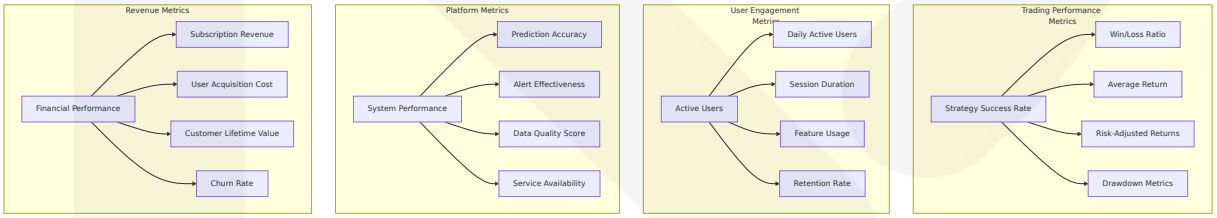
```
    name="ml_prediction_accuracy",
    description="ML model prediction accuracy",
    unit="percentage"
)

#### Performance monitoring decorator
def monitor_performance(metric_name):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            start_time = time.time()

            try:
                result = func(*args, **kwargs)
                status = "success"
                return result
            except Exception as e:
                status = "error"
                raise
            finally:
                duration_ms = (time.time() - start_time) * 1000

#### Record metrics
                api_response_time.record(
                    duration_ms,
                    attributes={
                        "endpoint": metric_name,
                        "status": status
                    }
                )

                api_request_count.add(
                    1,
                    attributes={
                        "endpoint": metric_name,
                        "status": status
                    }
                )

        return wrapper
    return decorator

#### Usage example
```

```
@app.route('/api/trading-strategy')
@monitor_performance("trading_strategy_generation")
def generate_trading_strategy():
#### Implementation here
    pass
```

6.5.2.3 Business Metrics

The system tracks critical business metrics that directly impact trading success and user satisfaction, providing insights into the effectiveness of trading strategies and overall platform performance.

Business Metrics Dashboard:



Business Metrics Implementation:

```
# Business metrics tracking
class BusinessMetrics:
    def __init__(self):
        self.meter = metrics.get_meter("forex-app-business")

        # Trading performance metrics
        self.strategy_success_rate = self.meter.create_histogram(
            name="strategy_success_rate",
            description="Trading strategy success rate percentage",
            unit="percentage"
        )

        self.user_profit_loss = self.meter.create_histogram(
            name="user_profit_loss",
            description="User profit/loss per trade",
            unit="currency"
        )

        # User engagement metrics
```

```
        self.active_users = self.meter.create_up_down_counter(
            name="active_users_count",
            description="Number of active users",
            unit="1"
        )

        self.session_duration = self.meter.create_histogram(
            name="user_session_duration",
            description="User session duration in minutes",
            unit="minutes"
        )

    def record_trading_outcome(self, user_id, strategy_id, outcome, prof
        """Record trading strategy outcome"""
        success_rate = 100 if outcome == 'success' else 0

        self.strategy_success_rate.record(
            success_rate,
            attributes={
                "user_id": user_id,
                "strategy_id": strategy_id,
                "outcome": outcome
            }
        )

        self.user_profit_loss.record(
            profit_loss,
            attributes={
                "user_id": user_id,
                "outcome": outcome
            }
        )

    def record_user_session(self, user_id, session_start, session_end):
        """Record user session metrics"""
        duration_minutes = (session_end - session_start).total_seconds()

        self.session_duration.record(
            duration_minutes,
            attributes={"user_id": user_id}
        )

    def update_active_users(self, count):
```

```
        """Update active user count"""
        self.active_users.add(count)

#### Usage in application
business_metrics = BusinessMetrics()

@app.route('/api/trading-result', methods=['POST'])
def record_trading_result():
    data = request.json
    business_metrics.record_trading_outcome(
        user_id=data['user_id'],
        strategy_id=data['strategy_id'],
        outcome=data['outcome'],
        profit_loss=data['profit_loss']
    )
    return jsonify({'status': 'recorded'})
```

6.5.2.4 SLA Monitoring

Modern SLA/SLO monitoring translates business requirements into technical metrics, automated alerts, and actionable dashboards, shifting focus to user experience and business impact rather than just infrastructure health.

SLA Monitoring Framework:

Service Component	Availability SLA	Response Time SLA	Error Rate SLA
Mobile Application	99.5%	<3s app launch	<2% crash rate
API Gateway	99.9%	<500ms average	<0.5% error rate
Trading Strategy Generation	99.5%	<10s end-to-end	<1% failure rate
ML Prediction Service	99.0%	<5s inference	<2% error rate

SLA Monitoring Implementation:

```
from datetime import datetime, timedelta
import asyncio

class SLAMonitor:
    def __init__(self):
        self.sla_targets = {
            'api_gateway': {
                'availability': 99.9,
                'response_time': 500,  # ms
                'error_rate': 0.5  # %
            },
            'trading_strategy': {
                'availability': 99.5,
                'response_time': 10000,  # ms
                'error_rate': 1.0  # %
            },
            'ml_prediction': {
                'availability': 99.0,
                'response_time': 5000,  # ms
                'error_rate': 2.0  # %
            }
        }

        self.sla_violations = []

    def calculate_availability(self, service, time_window_hours=24):
        """Calculate service availability over time window"""
        end_time = datetime.utcnow()
        start_time = end_time - timedelta(hours=time_window_hours)

        # Query metrics for uptime/downtime
        total_checks = self.get_health_check_count(service, start_time, e
        successful_checks = self.get_successful_checks(service, start_tir

        if total_checks == 0:
            return 0

        availability = (successful_checks / total_checks) * 100
        return availability

    def check_sla_compliance(self, service):
        """Check if service meets SLA targets"""
        if service not in self.sla_targets:
```

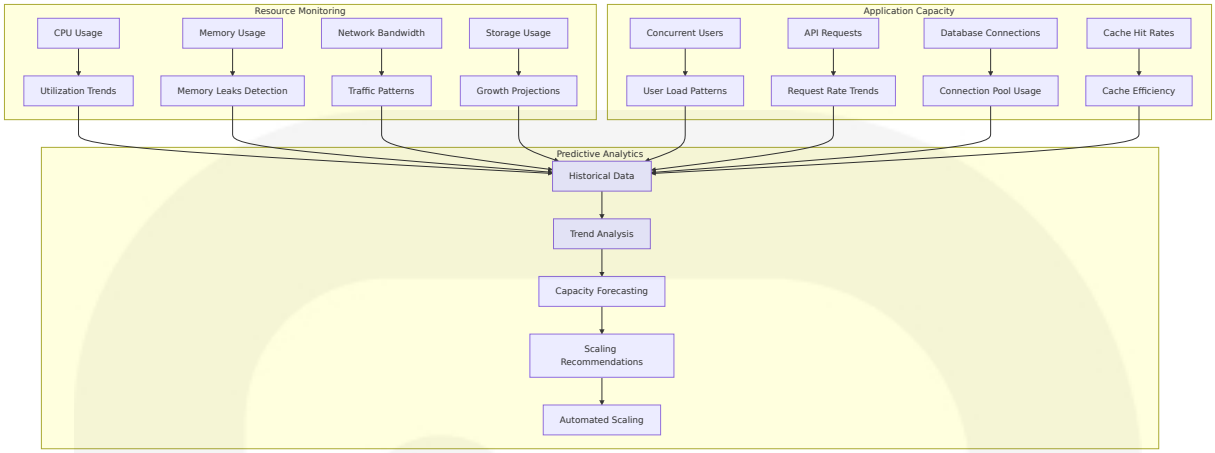
```
            return None

        targets = self.sla_targets[service]
        current_metrics = self.get_current_metrics(service)

        violations = []

        # Check availability
        availability = self.calculate_availability(service)
        if availability < targets['availability']:
            violations.append({
                'type': 'availability',
                'target': targets['availability'],
                'actual': availability,
                'severity': 'critical'
            })

        # Check response time
        avg_response_time = current_metrics.get('avg_response_time', 0)
        if avg_response_time > targets['response_time']:
            violations.append({
                'type': 'response_time',
                'target': targets['response_time'],
                'actual': avg_response_time,
                'severity': 'high'
            })

        # Check error rate
        error_rate = current_metrics.get('error_rate', 0)
        if error_rate > targets['error_rate']:
            violations.append({
                'type': 'error_rate',
                'target': targets['error_rate'],
                'actual': error_rate,
                'severity': 'high'
            })

        return violations

    async def monitor_slas(self):
        """Continuous SLA monitoring"""
        while True:
            for service in self.sla_targets.keys():
```

```
                violations = self.check_sla_compliance(service)

                if violations:
                    for violation in violations:
                        self.handle_sla_violation(service, violation)

            await asyncio.sleep(60)  # Check every minute

    def handle_sla_violation(self, service, violation):
        """Handle SLA violation"""
        violation_record = {
            'timestamp': datetime.utcnow().isoformat(),
            'service': service,
            'violation': violation,
            'id': str(uuid.uuid4())
        }

        self.sla_violations.append(violation_record)

        # Send alerts based on severity
        if violation['severity'] == 'critical':
            self.send_critical_alert(service, violation)
        elif violation['severity'] == 'high':
            self.send_high_priority_alert(service, violation)

        logger.error(f"SLA violation detected: {violation_record}")
```

6.5.2.5 Capacity Tracking

The system implements proactive capacity tracking to ensure optimal performance during peak trading hours and market volatility periods.

Capacity Monitoring Architecture:



Capacity Tracking Implementation:

```
class CapacityTracker:
    def __init__(self):
        self.capacity_thresholds = {
            'cpu_usage': 70,  # %
            'memory_usage': 80,  # %
            'disk_usage': 85,  # %
            'connection_pool': 80,  # %
            'api_rate_limit': 90  # %
        }

        self.scaling_triggers = {
            'scale_up': {
                'cpu_usage': 80,
                'memory_usage': 85,
                'api_requests': 1000  # per minute
            },
            'scale_down': {
                'cpu_usage': 30,
                'memory_usage': 40,
                'api_requests': 200  # per minute
            }
        }

    def collect_capacity_metrics(self):
        """Collect current capacity metrics"""
        return {
            'timestamp': datetime.utcnow().isoformat(),
            'cpu_usage': self.get_cpu_usage(),
            'memory_usage': self.get_memory_usage(),
```

```
            'disk_usage': self.get_disk_usage(),
            'active_connections': self.get_active_connections(),
            'api_requests_per_minute': self.get_api_request_rate(),
            'concurrent_users': self.get_concurrent_users()
        }

    def analyze_capacity_trends(self, time_window_hours=24):
        """Analyze capacity trends over time window"""
        metrics_history = self.get_metrics_history(time_window_hours)

        trends = {}
        for metric in ['cpu_usage', 'memory_usage', 'api_requests_per_mi
            values = [m[metric] for m in metrics_history]
            trends[metric] = {
                'current': values[-1] if values else 0,
                'average': sum(values) / len(values) if values else 0,
                'peak': max(values) if values else 0,
                'trend': self.calculate_trend(values)
            }

        return trends

    def predict_capacity_needs(self, forecast_hours=24):
        """Predict future capacity needs"""
        trends = self.analyze_capacity_trends()
        predictions = {}

        for metric, trend_data in trends.items():
            if trend_data['trend'] > 0:  # Increasing trend
                predicted_value = trend_data['current'] + (
                    trend_data['trend'] * forecast_hours
                )
                predictions[metric] = {
                    'predicted_value': predicted_value,
                    'threshold_breach': predicted_value > self.capacity_
                    'recommended_action': self.get_scaling_recommendatio
                }

        return predictions

    def get_scaling_recommendation(self, metric, predicted_value):
        """Get scaling recommendation based on predictions"""
        if predicted_value > self.scaling_triggers['scale_up'].get(metri
```

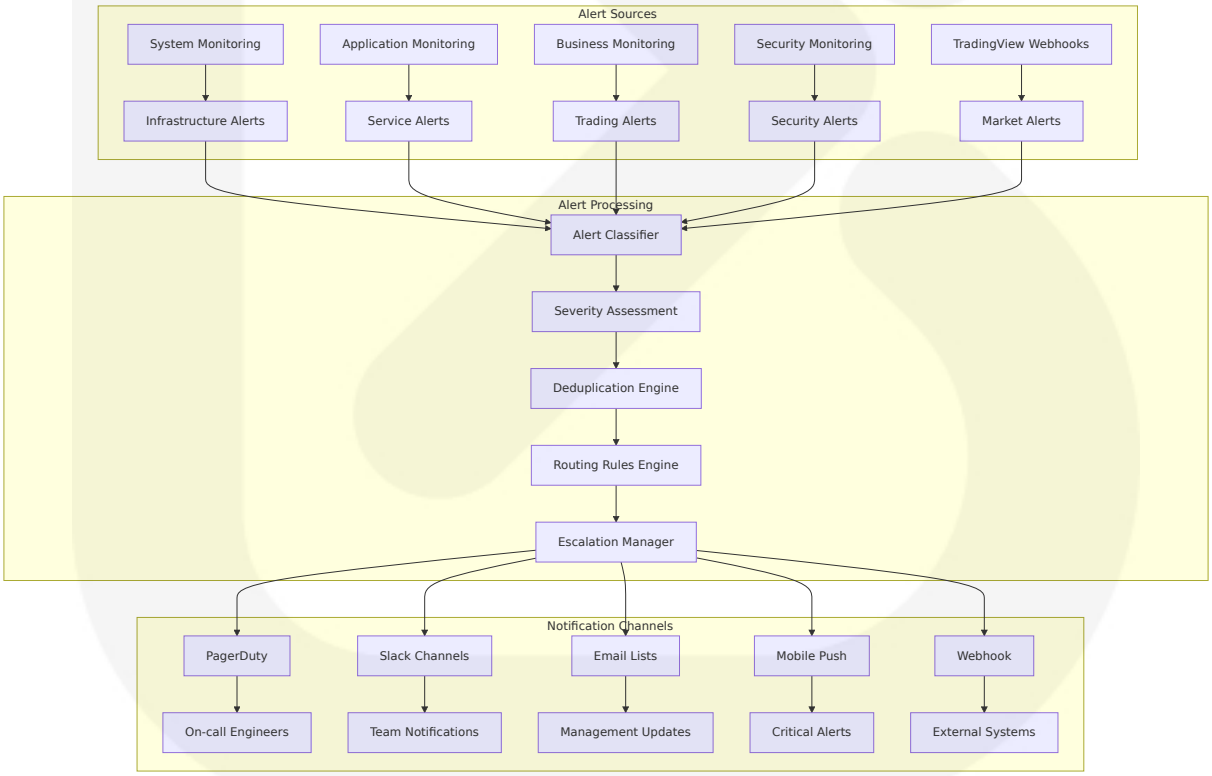
```
            return 'scale_up'
        elif predicted_value < self.scaling_triggers['scale_down'].get(me
            return 'scale_down'
        else:
            return 'maintain'
```

6.5.3 INCIDENT RESPONSE

6.5.3.1 Alert Routing

The system implements intelligent alert routing to ensure critical issues reach the right teams with appropriate urgency levels, integrating with TradingView alerts that help traders and investors stay on top of key market events by sending timely notifications for market-related incidents.

Alert Routing Architecture:



Alert Routing Configuration:

Alert Type	Severity	Primary Channel	Secondary Channel	Escalation Time
System Down	Critical	PagerDuty	Slack + SMS	5 minutes
API Degradation	High	Slack	Email	15 minutes
Trading Strategy Failure	High	Slack + Email	PagerDuty	30 minutes
Security Incident	Critical	PagerDuty + SMS	Security Team	Immediate

Alert Routing Implementation:

```
import json
from enum import Enum
from datetime import datetime, timedelta

class AlertSeverity(Enum):
    CRITICAL = "critical"
    HIGH = "high"
    MEDIUM = "medium"
    LOW = "low"

class AlertRouter:
    def __init__(self):
        self.routing_rules = {
            'system_down': {
                'severity': AlertSeverity.CRITICAL,
                'channels': ['pagerduty', 'slack', 'sms'],
                'escalation_time': 5,  # minutes
                'teams': ['sre', 'engineering']
            },
            'api_performance': {
                'severity': AlertSeverity.HIGH,
                'channels': ['slack', 'email'],
                'escalation_time': 15,
                'teams': ['backend', 'devops']
            },
            'trading_strategy_failure': {
                'severity': AlertSeverity.HIGH,
```

```
                'channels': ['slack', 'email'],
                'escalation_time': 30,
                'teams': ['trading', 'ml_engineering']
            },
            'security_incident': {
                'severity': AlertSeverity.CRITICAL,
                'channels': ['pagerduty', 'sms', 'secure_email'],
                'escalation_time': 0,  # Immediate
                'teams': ['security', 'management']
            },
            'market_alert': {
                'severity': AlertSeverity.MEDIUM,
                'channels': ['slack', 'mobile_push'],
                'escalation_time': 60,
                'teams': ['trading', 'analysts']
            }
        }

        self.active_incidents = {}
        self.notification_channels = {
            'pagerduty': self.send_pagerduty_alert,
            'slack': self.send_slack_alert,
            'email': self.send_email_alert,
            'sms': self.send_sms_alert,
            'mobile_push': self.send_mobile_push
        }

    def process_alert(self, alert_data):
        """Process incoming alert and route appropriately"""
        alert_type = self.classify_alert(alert_data)
        routing_rule = self.routing_rules.get(alert_type)

        if not routing_rule:
            logger.warning(f"No routing rule found for alert type: {aler
            return

        # Check for duplicate alerts
        alert_key = self.generate_alert_key(alert_data)
        if self.is_duplicate_alert(alert_key):
            logger.info(f"Duplicate alert suppressed: {alert_key}")
            return

        # Create incident record
```

```
        incident = self.create_incident(alert_data, routing_rule)

        # Route to appropriate channels
        self.route_alert(incident, routing_rule)

        # Schedule escalation if needed
        if routing_rule['escalation_time'] > 0:
            self.schedule_escalation(incident, routing_rule)

    def classify_alert(self, alert_data):
        """Classify alert based on content and source"""
        source = alert_data.get('source', '')
        metric = alert_data.get('metric', '')
        message = alert_data.get('message', '').lower()

        # System health alerts
        if 'system' in source and ('down' in message or 'unavailable' in
            return 'system_down'

        # API performance alerts
        if 'api' in source and ('response_time' in metric or 'latency' in
            return 'api_performance'

        # Trading strategy alerts
        if 'trading' in source or 'strategy' in message:
            return 'trading_strategy_failure'

        # Security alerts
        if 'security' in source or 'unauthorized' in message:
            return 'security_incident'

        # TradingView market alerts
        if 'tradingview' in source or 'market' in source:
            return 'market_alert'

        return 'unknown'

    def route_alert(self, incident, routing_rule):
        """Route alert to configured channels"""
        for channel in routing_rule['channels']:
            if channel in self.notification_channels:
                try:
                    self.notification_channels[channel](incident)
```

```
                    logger.info(f"Alert sent via {channel} for incident 
                except Exception as e:
                    logger.error(f"Failed to send alert via {channel}: {e

    def send_slack_alert(self, incident):
        """Send alert to Slack"""
        slack_message = {
            "text": f"ðŸš¨ {incident['severity'].value.upper()} Alert",
            "attachments": [
                {
                    "color": "danger" if incident['severity'] == AlertSev
                    "fields": [
                        {"title": "Incident ID", "value": incident['id']
                        {"title": "Type", "value": incident['type'], "sho
                        {"title": "Description", "value": incident['descr
                        {"title": "Time", "value": incident['timestamp']]
                    ]
                }
            ]
        }

        # Send to appropriate Slack channels based on teams
        for team in incident['teams']:
            channel = f"#{team}-alerts"
            self.slack_client.send_message(channel, slack_message)

    def send_pagerduty_alert(self, incident):
        """Send alert to PagerDuty"""
        pagerduty_payload = {
            "routing_key": self.get_pagerduty_routing_key(incident['teams
            "event_action": "trigger",
            "dedup_key": incident['id'],
            "payload": {
                "summary": f"{incident['type']}: {incident['description']
                "severity": incident['severity'].value,
                "source": incident['source'],
                "timestamp": incident['timestamp'],
                "custom_details": incident.get('details', {})
            }
        }

        self.pagerduty_client.send_event(pagerduty_payload)
```

6.5.3.2 Escalation Procedures

The system implements automated escalation procedures to ensure critical issues receive appropriate attention and resolution within defined timeframes.

Escalation Matrix:

Incident Severity	L1 Response Time	L2 Escalation	L3 Escalation	Management Notification
Critical	5 minutes	15 minutes	30 minutes	Immediate
High	15 minutes	1 hour	4 hours	2 hours
Medium	1 hour	4 hours	Next business day	Daily summary
Low	4 hours	Next business day	Weekly review	Weekly summary

Escalation Implementation:

```
class EscalationManager:
    def __init__(self):
        self.escalation_rules = {
            AlertSeverity.CRITICAL: [
                {'level': 1, 'time_minutes': 5, 'teams': ['sre_oncall']}
                {'level': 2, 'time_minutes': 15, 'teams': ['sre_manager'
                {'level': 3, 'time_minutes': 30, 'teams': ['cto', 'vp_eng
                {'level': 4, 'time_minutes': 60, 'teams': ['ceo']}
            ],
            AlertSeverity.HIGH: [
                {'level': 1, 'time_minutes': 15, 'teams': ['team_lead']}
                {'level': 2, 'time_minutes': 60, 'teams': ['engineering_m
                {'level': 3, 'time_minutes': 240, 'teams': ['vp_engineer
            ],
            AlertSeverity.MEDIUM: [
                {'level': 1, 'time_minutes': 60, 'teams': ['assigned_engi
                {'level': 2, 'time_minutes': 240, 'teams': ['team_lead']
            ]
```

```
        }

        self.escalation_tasks = {}

    def schedule_escalation(self, incident):
        """Schedule escalation for an incident"""
        severity = incident['severity']
        if severity not in self.escalation_rules:
            return

        escalation_chain = self.escalation_rules[severity]

        for escalation_level in escalation_chain:
            escalation_time = datetime.utcnow() + timedelta(
                minutes=escalation_level['time_minutes']
            )

            task_id = f"{incident['id']}_level_{escalation_level['level']
            self.escalation_tasks[task_id] = {
                'incident_id': incident['id'],
                'level': escalation_level['level'],
                'teams': escalation_level['teams'],
                'scheduled_time': escalation_time,
                'executed': False
            }

    def check_escalations(self):
        """Check and execute due escalations"""
        current_time = datetime.utcnow()

        for task_id, task in self.escalation_tasks.items():
            if (not task['executed'] and
                current_time >= task['scheduled_time'] and
                not self.is_incident_resolved(task['incident_id'])):

                self.execute_escalation(task)
                task['executed'] = True

    def execute_escalation(self, escalation_task):
        """Execute escalation to next level"""
        incident = self.get_incident(escalation_task['incident_id'])

        escalation_message = {
```

```
            'type': 'escalation',
            'incident_id': incident['id'],
            'level': escalation_task['level'],
            'original_alert_time': incident['timestamp'],
            'escalation_time': datetime.utcnow().isoformat(),
            'teams': escalation_task['teams'],
            'incident_summary': incident['description']
        }

        # Notify escalation teams
        for team in escalation_task['teams']:
            self.notify_escalation_team(team, escalation_message)

        logger.warning(f"Incident {incident['id']} escalated to level {es

    def notify_escalation_team(self, team, escalation_message):
        """Notify escalation team"""
        if team in ['ceo', 'cto', 'vp_engineering']:
            # High-level escalation - use multiple channels
            self.send_executive_notification(team, escalation_message)
        else:
            # Standard escalation
            self.send_team_notification(team, escalation_message)
```

6.5.3.3 Runbooks

The system maintains comprehensive runbooks for common incident scenarios, enabling rapid response and consistent resolution procedures.

Runbook Categories:

Incident Type	Runbook	Response Team	Estimated Resolution Time
API Gateway Down	RB-001	SRE, Backend	15 minutes
Database Connection Issues	RB-002	DBA, SRE	30 minutes
ML Model Failures	RB-003	ML Engineering	45 minutes

Incident Type	Runbook	Response Team	Estimated Resolution Time
TradingView API Issues	RB-004	Trading Team, SRE	20 minutes

Runbook Implementation:

```
class RunbookManager:
    def __init__(self):
        self.runbooks = {
            'api_gateway_down': {
                'id': 'RB-001',
                'title': 'API Gateway Service Down',
                'steps': [
                    {
                        'step': 1,
                        'action': 'Check service status',
                        'command': 'kubectl get pods -n api-gateway',
                        'expected_result': 'All pods running',
                        'troubleshooting': 'If pods not running, check lo
                    },
                    {
                        'step': 2,
                        'action': 'Check load balancer health',
                        'command': 'curl -f http://api-gateway/health',
                        'expected_result': 'HTTP 200 response',
                        'troubleshooting': 'If unhealthy, restart service
                    },
                    {
                        'step': 3,
                        'action': 'Verify database connectivity',
                        'command': 'kubectl exec -it api-gateway-pod -- p
                        'expected_result': 'Successful ping',
                        'troubleshooting': 'Check database connection str
                    }
                ],
                'escalation_criteria': 'If not resolved in 15 minutes',
                'rollback_procedure': 'Deploy previous stable version'
            },
            'ml_model_failure': {
                'id': 'RB-003',
```

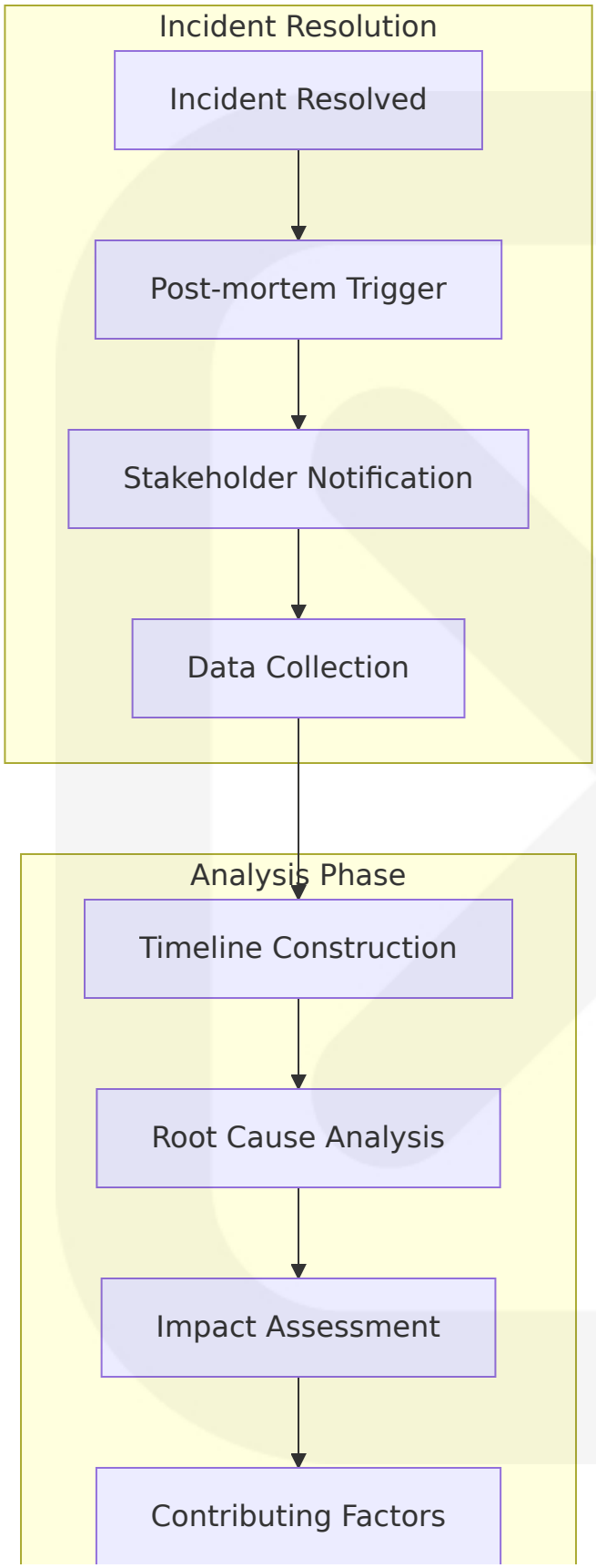
```
                'title': 'ML Model Prediction Failures',
                'steps': [
                    {
                        'step': 1,
                        'action': 'Check model service health',
                        'command': 'curl -f http://ml-service/health',
                        'expected_result': 'HTTP 200 with model status',
                        'troubleshooting': 'Check model loading errors'
                    },
                    {
                        'step': 2,
                        'action': 'Verify model artifacts',
                        'command': 'ls -la /models/ && du -sh /models/*',
                        'expected_result': 'Model files present and corre
                        'troubleshooting': 'Re-download model artifacts'
                    },
                    {
                        'step': 3,
                        'action': 'Test model inference',
                        'command': 'python test_model_inference.py',
                        'expected_result': 'Successful prediction with co
                        'troubleshooting': 'Fallback to previous model ve
                    }
                ],
                'escalation_criteria': 'If accuracy drops below 60%',
                'rollback_procedure': 'Switch to backup model'
            }
        }

    def get_runbook(self, incident_type):
        """Get runbook for incident type"""
        return self.runbooks.get(incident_type)

    def execute_runbook_step(self, runbook_id, step_number, incident_cont
        """Execute a specific runbook step"""
        runbook = None
        for rb in self.runbooks.values():
            if rb['id'] == runbook_id:
                runbook = rb
                break

        if not runbook:
            return {'error': 'Runbook not found'}
```

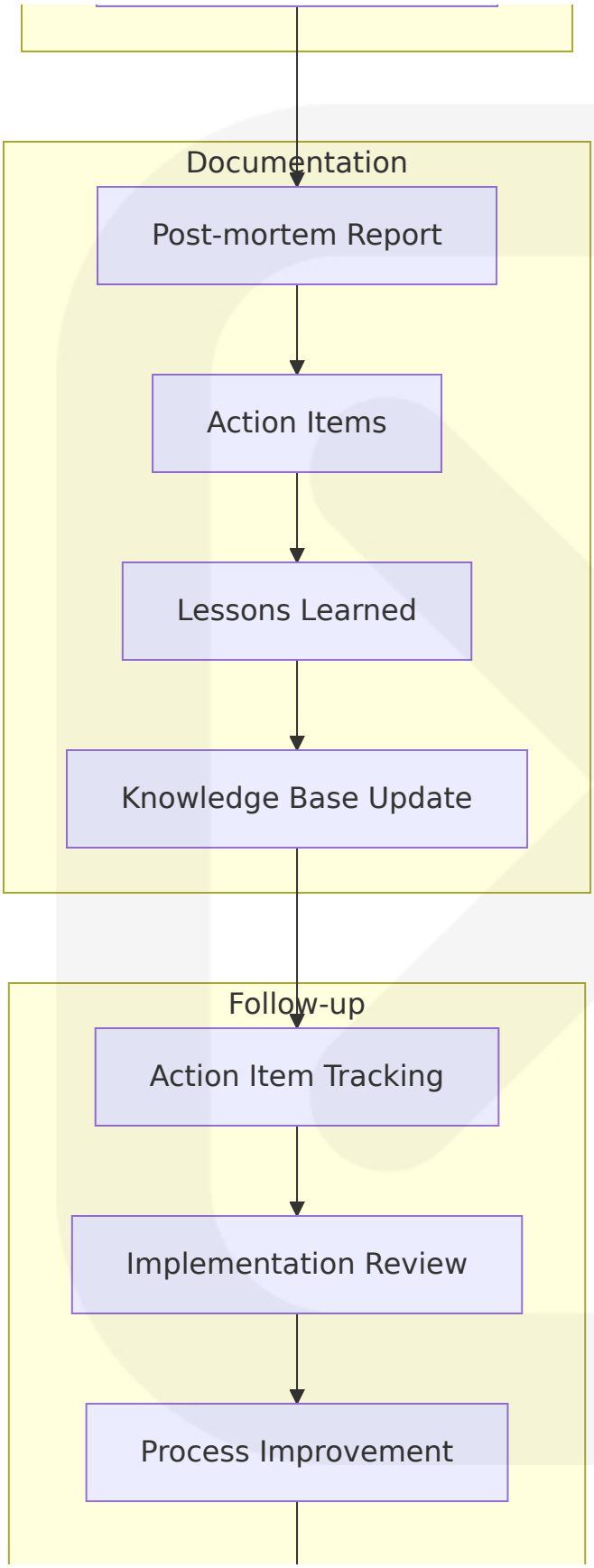
```
        step = next((s for s in runbook['steps'] if s['step'] == step_nu
        if not step:
            return {'error': 'Step not found'}

        # Log step execution
        logger.info(f"Executing runbook {runbook_id} step {step_number}:

        # Execute command if provided
        result = {
            'step': step_number,
            'action': step['action'],
            'command': step.get('command'),
            'timestamp': datetime.utcnow().isoformat(),
            'incident_id': incident_context.get('incident_id')
        }

        if step.get('command'):
            try:
                # Execute command (in production, use proper subprocess
                command_result = self.execute_command(step['command'])
                result['command_output'] = command_result
                result['status'] = 'completed'
            except Exception as e:
                result['error'] = str(e)
                result['status'] = 'failed'
                result['troubleshooting'] = step.get('troubleshooting')

        return result
```

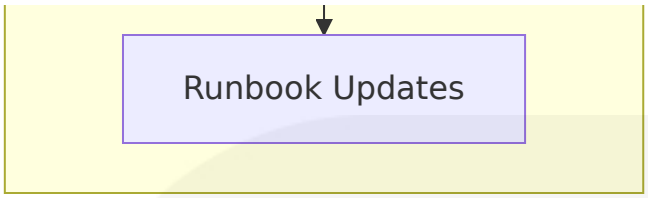
6.5.3.4 Post-mortem Processes

The system implements comprehensive post-mortem processes to learn from incidents and improve system reliability.

Post-mortem Framework:


Incident Resolution
Incident Resolved
Post-mortem Trigger
Stakeholder Notification
Data Collection
Analysis Phase
Timeline Construction
Root Cause Analysis
Impact Assessment
Contributing Factors


Documentation
Post-mortem Report
Action Items
Lessons Learned
Knowledge Base Update
Follow-up
Action Item Tracking
Implementation Review
Process Improvement



Post-mortem Implementation:

```
class PostMortemManager:
    def __init__(self):
        self.post_mortem_template = {
            'incident_id': '',
            'title': '',
            'date': '',
            'duration': '',
            'severity': '',
            'impact': {
                'users_affected': 0,
                'revenue_impact': 0,
                'services_affected': []
            },
            'timeline': [],
            'root_cause': '',
            'contributing_factors': [],
            'what_went_well': [],
            'what_went_wrong': [],
            'action_items': [],
            'lessons_learned': []
        }

    def create_post_mortem(self, incident):
        """Create post-mortem for resolved incident"""
        if incident['severity'] not in [AlertSeverity.CRITICAL, AlertSeve
            return None  # Only create post-mortems for critical/high se

        post_mortem = self.post_mortem_template.copy()
        post_mortem.update({
            'incident_id': incident['id'],
            'title': f"Post-mortem: {incident['description']}",
            'date': incident['timestamp'],
            'duration': self.calculate_incident_duration(incident),
            'severity': incident['severity'].value,
            'timeline': self.construct_timeline(incident)
```

```
        })

        return post_mortem

    def construct_timeline(self, incident):
        """Construct incident timeline from logs and events"""
        timeline = []

        # Get all events related to the incident
        events = self.get_incident_events(incident['id'])

        for event in events:
            timeline.append({
                'timestamp': event['timestamp'],
                'event': event['description'],
                'source': event['source'],
                'impact': event.get('impact', 'unknown')
            })

        return sorted(timeline, key=lambda x: x['timestamp'])

    def analyze_root_cause(self, incident, timeline):
        """Perform root cause analysis using 5 Whys technique"""
        root_cause_analysis = {
            'initial_problem': incident['description'],
            'five_whys': [],
            'root_cause': '',
            'verification': ''
        }

        # This would typically involve human analysis, but we can provide
        return root_cause_analysis

    def generate_action_items(self, post_mortem):
        """Generate action items based on post-mortem analysis"""
        action_items = []

        # Common action items based on incident type
        if 'api' in post_mortem['title'].lower():
            action_items.extend([
                {
                    'item': 'Improve API monitoring and alerting',
                    'owner': 'SRE Team',
```

```
                    'due_date': (datetime.utcnow() + timedelta(days=14))
                    'priority': 'high'
                },
                {
                    'item': 'Review API rate limiting configuration',
                    'owner': 'Backend Team',
                    'due_date': (datetime.utcnow() + timedelta(days=7)).
                    'priority': 'medium'
                }
            ])

        if 'database' in post_mortem['title'].lower():
            action_items.extend([
                {
                    'item': 'Implement database connection pooling improv
                    'owner': 'DBA Team',
                    'due_date': (datetime.utcnow() + timedelta(days=21))
                    'priority': 'high'
                }
            ])

        return action_items

    def track_action_items(self, post_mortem_id):
        """Track completion of action items"""
        post_mortem = self.get_post_mortem(post_mortem_id)
        if not post_mortem:
            return

        completion_status = {}
        for item in post_mortem['action_items']:
            item_id = item.get('id')
            if item_id:
                status = self.check_action_item_status(item_id)
                completion_status[item_id] = status

        return completion_status
```

6.5.3.5 Improvement Tracking

The system tracks improvements implemented as a result of incidents and post-mortems to measure the effectiveness of the incident response

process.

Improvement Tracking Metrics:

Metric	Target	Current	Trend
Mean Time to Detection (MTTD)	<5 minutes	7 minutes	Improving
Mean Time to Resolution (MTTR)	<30 minutes	45 minutes	Stable
Incident Recurrence Rate	<5%	8%	Improving
Post-mortem Completion Rate	100%	95%	Stable

Improvement Implementation:

```
class ImprovementTracker:
    def __init__(self):
        self.improvement_metrics = {
            'mttd': [],  # Mean Time to Detection
            'mttr': [],  # Mean Time to Resolution
            'incident_count': [],
            'recurrence_rate': [],
            'action_item_completion':

## 6.6 TESTING STRATEGY

### 6.6.1 TESTING APPROACH

#### 6.6.1.1 Unit Testing

**Testing Framework and Tools:**

| Component | Framework | Purpose | Coverage Target |
|---|---|---|---|
| React Native Mobile | Jest with jest-expo preset | Component and logic
| Python Backend | PyTest framework | API and business logic testing | >8
| TradingView Integration | API mocking with WireMock | External service
```

```
**React Native Unit Testing Configuration:**
The jest-expo library is a Jest preset that mocks the native part of the
```

javascript
// jest.config.js
module.exports = {
preset: 'jest-expo',
setupFilesAfterEnv: ['/setup-testing.js'],
transformIgnorePatterns: [
'node_modules/(?!((jest-)?react-native|@react-native(-community)?)|expo(nent)?|@expo(nent)?/.|@expo-google-fonts/.|react-navigation|@react-navigation/.|@sentry/react-native|native-base|react-native-svg))'], collectCoverageFrom: ['src//**.{js,jsx,ts,tsx}',**
'!src//*.d.ts',**
'!src/index.js'
],
coverageThreshold: {
global: {
branches: 90,
functions: 90,
lines: 90,
statements: 90
}
}
};

```
**Python Backend Unit Testing Configuration:**
Pytest is a Python testing framework designed to assist developers with v
```

python

pytest.ini

[tool:pytest]
testpaths = tests
python_files = test_*.py python_classes = Test
python_functions = test_*
addopts =
--strict-markers
--strict-config
--cov=src
--cov-report=term-missing
--cov-report=html
--cov-fail-under=85
markers =
unit: Unit tests
integration: Integration tests
slow: Slow running tests

```
**Test Organization Structure:**
Create a __tests__ directory at the root of your project's directory, wi
```

forex-trading-app/
â”œâ”€â”€ mobile/
â”‚ â”œâ”€â”€ **tests**/
â”‚ â”‚ â”œâ”€â”€ components/
â”‚ â”‚ â”œâ”€â”€ screens/
â”‚ â”‚ â”œâ”€â”€ services/
â”‚ â”‚ â”œâ”€â”€ utils/
â”‚ â”œâ”€â”€ src/
â”œâ”€â”€ backend/
â”œâ”€â”€ tests/
â”‚ â”œâ”€â”€ unit/
â”‚ â”œâ”€â”€ integration/
â”‚ â”œâ”€â”€ fixtures/
â”œâ”€â”€ src/

```
**Mocking Strategy:**
API mocking creates virtual endpoints that return predefined responses ma

**Test Data Management:**
- **Fixtures**: Reusable test data sets for consistent testing
- **Factories**: Dynamic test data generation for varied scenarios
- **Mocks**: External service simulation for isolated testing
- **Snapshots**: UI component regression testing

**Test Naming Conventions:**
Pytest recommends that test file names begin with the format test_*.py o

#### 6.6.1.2 Integration Testing

**Service Integration Test Approach:**
Functional tests should focus on how the view functions operate, focusing

**Integration Testing Matrix:**

| Integration Type | Test Scope | Tools | Frequency |
|---|---|---|---|
| API Integration | Flask routes with database | pytest + Flask test clie
| Database Integration | MongoDB operations | pytest + test database | Ev
| External Service Integration | TradingView API mocking | WireMock for c
| Mobile-Backend Integration | End-to-end API calls | Jest + MSW | Every

**API Testing Strategy:**
Testing Flask requires importing a Flask instance app from the API, with
```

python

Flask API Integration Test Example

@pytest.fixture
def client():
app.config['TESTING'] = True

with app.test_client() as client:
with app.app_context():
yield client

def test_trading_strategy_generation(client):
response = client.post('/api/v1/trading-strategy',
json={'symbol': 'EURUSD', 'timeframe': '1h'})
assert response.status_code == 200
assert 'strategy' in response.json
assert response.json['confidence_score'] > 0.6

```
**Database Integration Testing:**
Pytest fixtures allow writing pieces of code that are reusable across tes

**External Service Mocking:**
API mocking eliminates dependencies, significantly reduces testing expens

**Test Environment Management:**
- **Isolated Test Databases**: Separate MongoDB instances for testing
- **Docker Containers**: Consistent test environment setup
- **Environment Variables**: Configuration management for different test
- **Data Seeding**: Automated test data population

#### 6.6.1.3 End-to-End Testing

**E2E Test Scenarios:**
In end-to-end (E2E) tests, you verify your app is working as expected on

**E2E Testing Framework Selection:**

| Framework | Strengths | Use Case | Platform Support |
|---|---|---|---|
| Detox | Built for React Native, smart sync model minimizes race conditi
| Maestro | Fast, reliable mobile E2E testing without complexity, languag

**Maestro E2E Testing Implementation:**
Maestro has a significantly lower learning curve than alternatives like D
```

yaml

Trading Strategy E2E Test Flow

appId: com.forextrading.app

- launchApp
- assertVisible: "Forex Trading"
- tapOn:
  id: "currency_pair_selector"
- tapOn: "EURUSD"
- tapOn:
  id: "generate_strategy_button"
- assertVisible:
  text: "Strategy Generated"
  timeout: 10000
- assertVisible:
  id: "confidence_score"
- tapOn:
  id: "view_details_button"
- assertVisible: "Entry Point"
- assertVisible: "Exit Point"

```
**UI Automation Approach:**
E2E testing libraries allow you to find and control elements in the scree

**Test Data Setup/Teardown:**
- **Pre-test Setup**: Clean app state and test data preparation
- **Post-test Cleanup**: Reset app state and clear test artifacts
- **Data Isolation**: Separate test data for each E2E scenario
- **State Management**: Consistent app state between test runs

**Performance Testing Requirements:**
- **Load Testing**: API performance under concurrent users
- **Stress Testing**: System behavior under extreme conditions
```

```
- **Response Time Testing**: <2 second API response requirements
- **Memory Usage Testing**: Mobile app resource consumption

**Cross-Platform Testing Strategy:**
Maestro supports cross-platform Android and iOS testing with gestures, de

### 6.6.2 TEST AUTOMATION

#### 6.6.2.1 CI/CD Integration

**Automated Test Pipeline Architecture:**

<div class="mermaid-wrapper" id="mermaid-diagram-dozkdbyny">
          <div class="mermaid">
graph TB
    subgraph &amp;quot;CI/CD Pipeline&amp;quot;
        A[Code Commit] --&amp;gt; B[Lint &amp;amp; Format Check]
        B --&amp;gt; C[Unit Tests]
        C --&amp;gt; D[Integration Tests]
        D --&amp;gt; E[Build Mobile App]
        E --&amp;gt; F[E2E Tests]
        F --&amp;gt; G[Performance Tests]
        G --&amp;gt; H[Security Tests]
        H --&amp;gt; I[Deploy to Staging]
        I --&amp;gt; J[Smoke Tests]
        J --&amp;gt; K[Deploy to Production]
    end

    subgraph &amp;quot;Test Execution&amp;quot;
        L[Jest Unit Tests] --&amp;gt; M[Coverage Report]
        N[Pytest Integration] --&amp;gt; O[API Test Report]
        P[Maestro E2E] --&amp;gt; Q[E2E Test Report]
        R[Load Testing] --&amp;gt; S[Performance Report]
    end

    C --&amp;gt; L
    D --&amp;gt; N
    F --&amp;gt; P
    G --&amp;gt; R
</div>
        </div>
```

```
**GitHub Actions Configuration:**
CircleCI Python orb configuration demonstrates automated test execution
```

yaml

.github/workflows/test.yml

name: Test Suite
on:
push:
branches: [main, develop]
pull_request:
branches: [main]

jobs:
unit-tests:
runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v4
- name: Setup Node.js
uses: actions/setup-node@v4
with:
node-version: '20'
- name: Install dependencies
run: npm ci
- name: Run Jest tests
run: npm test -- --coverage --watchAll=false
- name: Upload coverage
uses: codecov/codecov-action@v3

backend-tests:
runs-on: ubuntu-latest
services:

mongodb:
image: mongo:7.0
ports:
- 27017:27017
redis:
image: redis:7.0
ports:
- 6379:6379
steps:
- uses: actions/checkout@v4
- name: Setup Python
uses: actions/setup-python@v4
with:
python-version: '3.11'
- name: Install dependencies
run: pip install -r requirements.txt
- name: Run pytest
run: pytest --cov=src --cov-report=xml

```
**Automated Test Triggers:**
- **Pre-commit Hooks**: Lint and unit tests before code commit
- **Pull Request**: Full test suite execution on PR creation
- **Nightly Builds**: Comprehensive testing including performance tests
- **Release Branches**: Complete test suite with additional security scan

**Parallel Test Execution:**
CI/CD efficiency tools like WireMock integrate with Jenkins/GitLab, enabl

**Test Reporting Requirements:**
- **Coverage Reports**: Code coverage metrics with threshold enforcement
- **Test Results**: Detailed pass/fail status with execution times
- **Performance Metrics**: Response time and throughput measurements
- **Quality Gates**: Automated deployment blocking on test failures

#### 6.6.2.2 Failed Test Handling

**Test Failure Classification:**
```

```
| Failure Type | Handling Strategy | Retry Policy | Escalation |
|---|---|---|---|
| Flaky Tests | Automatic screenshots and real-time logs for test visibi
| Infrastructure Issues | Environment reset and retry | 2 retries | DevO
| Code Defects | Immediate failure reporting | No retry | Developer assig
| External Service Issues | Mock API fallback to avoid third-party API ca

**Flaky Test Management:**
The core problem with E2E tests is flakiness - tests are usually not dete

**Test Retry Configuration:**
```

javascript
// Jest retry configuration
module.exports = {
testRunner: 'jest-circus/runner',
testEnvironment: 'node',
setupFilesAfterEnv: ['/setup-tests.js'],
testTimeout: 30000,
retryTimes: 2,
bail: false
};

```
**Failure Notification System:**
- **Slack Integration**: Immediate test failure notifications
- **Email Alerts**: Daily test summary reports
- **Dashboard Updates**: Real-time test status visualization
- **Issue Tracking**: Automatic bug creation for consistent failures

### 6.6.3 QUALITY METRICS

#### 6.6.3.1 Code Coverage Targets

**Coverage Requirements by Component:**

| Component | Line Coverage | Branch Coverage | Function Coverage | State
|---|---|---|---|---|
| React Native Components | >90% | >85% | >95% | >90% |
```

```
| Python API Endpoints | >85% | >80% | >90% | >85% |
| Business Logic | >95% | >90% | >95% | >95% |
| Utility Functions | >90% | >85% | >90% | >90% |

**Coverage Enforcement:**
Pytest is highly extensible with many plugins available, such as the Pyte
```

python

pytest coverage configuration

[tool:pytest]
addopts =
--cov=src
--cov-report=term-missing
--cov-report=html:htmlcov
--cov-report=xml
--cov-fail-under=85
--cov-branch

```
**Coverage Exclusions:**
- Configuration files and constants
- Third-party library integrations
- Development and debugging utilities
- Generated code and migrations

#### 6.6.3.2 Test Success Rate Requirements

**Success Rate Targets:**

| Test Category | Success Rate Target | Measurement Period | Action Thres
|---|---|---|---|
| Unit Tests | >99% | Per commit | <95% blocks deployment |
| Integration Tests | >95% | Daily | <90% requires investigation |
| E2E Tests | >90% | Weekly | <85% triggers review |
```

```
| Performance Tests | >95% | Release cycle | <90% blocks release |

**Test Reliability Metrics:**
- **Flakiness Rate**: <5% for all test categories
- **Test Execution Time**: <30 minutes for full suite
- **Test Maintenance Overhead**: <10% of development time
- **False Positive Rate**: <2% for automated tests

#### 6.6.3.3 Performance Test Thresholds

**API Performance Requirements:**

| Endpoint Category | Response Time | Throughput | Error Rate |
|---|---|---|---|
| Market Data APIs | <2 seconds | >100 RPS | <1% |
| Trading Strategy Generation | <10 seconds | >50 RPS | <2% |
| ML Prediction APIs | <5 seconds | >20 RPS | <3% |
| User Authentication | <1 second | >200 RPS | <0.5% |

**Mobile App Performance Thresholds:**
- **App Launch Time**: <3 seconds cold start
- **Screen Transition**: <500ms between screens
- **Memory Usage**: <200MB peak usage
- **Battery Impact**: <5% per hour of active use

#### 6.6.3.4 Quality Gates

**Deployment Quality Gates:**

<div class="mermaid-wrapper" id="mermaid-diagram-nti7tkwmu">
          <div class="mermaid">
graph TD
    A[Code Commit] --&gt; B{Unit Tests Pass?}
    B --&gt;|No| C[Block Deployment]
    B --&gt;|Yes| D{Coverage &gt; 85%?}
    D --&gt;|No| C
    D --&gt;|Yes| E{Integration Tests Pass?}
    E --&gt;|No| C
    E --&gt;|Yes| F{Security Scan Clean?}
    F --&gt;|No| C
    F --&gt;|Yes| G{Performance Tests Pass?}
    G --&gt;|No| C
    G --&gt;|Yes| H[Deploy to Staging]
```

```
    H --&gt; I{E2E Tests Pass?}
    I --&gt;|No| J[Rollback Staging]
    I --&gt;|Yes| K[Deploy to Production]
</div>
        </div>

**Quality Metrics Dashboard:**
- **Test Coverage Trends**: Historical coverage analysis
- **Test Execution Metrics**: Success rates and execution times
- **Defect Density**: Bugs per lines of code
- **Test Maintenance Ratio**: Test code to production code ratio

#### 6.6.3.5 Documentation Requirements

**Test Documentation Standards:**
- **Test Plan Documentation**: Comprehensive test strategy documentation
- **Test Case Documentation**: Detailed test scenarios and expected outc
- **API Documentation**: API specification for exclusive financial produ
- **Troubleshooting Guides**: Common test failure resolution procedures

**Documentation Maintenance:**
- **Living Documentation**: Tests as executable specifications
- **API Contract Testing**: Automated API documentation validation
- **Test Report Generation**: Automated test result documentation
- **Knowledge Base Updates**: Regular documentation review and updates

### 6.6.4 TEST EXECUTION FLOW

#### 6.6.4.1 Test Execution Architecture

<div class="mermaid-wrapper" id="mermaid-diagram-gi9resm0c">
          <div class="mermaid">
flowchart TD
    subgraph &quot;Test Execution Pipeline&quot;
        A[Developer Commit] --&gt; B[Pre-commit Hooks]
        B --&gt; C[Lint &amp; Format Check]
        C --&gt; D{Checks Pass?}
        D --&gt;|No| E[Block Commit]
        D --&gt;|Yes| F[Unit Test Execution]

        F --&gt; G[Jest Mobile Tests]
        F --&gt; H[Pytest Backend Tests]
```

```
        G --&gt; I{Mobile Tests Pass?}
        H --&gt; J{Backend Tests Pass?}

        I --&gt;|No| K[Test Failure Report]
        J --&gt;|No| K
        I --&gt;|Yes| L[Integration Test Phase]
        J --&gt;|Yes| L

        L --&gt; M[API Integration Tests]
        L --&gt; N[Database Integration Tests]
        L --&gt; O[External Service Mocks]

        M --&gt; P{Integration Tests Pass?}
        N --&gt; P
        O --&gt; P

        P --&gt;|No| Q[Integration Failure Report]
        P --&gt;|Yes| R[Build Mobile App]

        R --&gt; S[E2E Test Execution]
        S --&gt; T[Maestro Test Suite]
        T --&gt; U{E2E Tests Pass?}

        U --&gt;|No| V[E2E Failure Report]
        U --&gt;|Yes| W[Performance Testing]

        W --&gt; X[Load Testing]
        W --&gt; Y[Stress Testing]
        X --&gt; Z{Performance Meets SLA?}
        Y --&gt; Z

        Z --&gt;|No| AA[Performance Failure]
        Z --&gt;|Yes| BB[Security Testing]

        BB --&gt; CC[SAST Scan]
        BB --&gt; DD[Dependency Check]
        CC --&gt; EE{Security Clean?}
        DD --&gt; EE

        EE --&gt;|No| FF[Security Issues Found]
        EE --&gt;|Yes| GG[Deploy to Staging]

        GG --&gt; HH[Smoke Tests]
```

```
        HH --&gt; II{Smoke Tests Pass?}
        II --&gt;|No| JJ[Staging Failure]
        II --&gt;|Yes| KK[Production Deployment]
    end

    subgraph &quot;Failure Handling&quot;
        E --&gt; LL[Developer Notification]
        K --&gt; MM[Test Report Generation]
        Q --&gt; MM
        V --&gt; MM
        AA --&gt; MM
        FF --&gt; NN[Security Team Alert]
        JJ --&gt; OO[Rollback Staging]
    end

    subgraph &quot;Success Path&quot;
        KK --&gt; PP[Production Monitoring]
        PP --&gt; QQ[Health Checks]
        QQ --&gt; RR[Performance Monitoring]
    end
</div>
        </div>

#### 6.6.4.2 Test Environment Architecture

<div class="mermaid-wrapper" id="mermaid-diagram-a2xg84ah3">
          <div class="mermaid">
graph TB
    subgraph &quot;Development Environment&quot;
        A[Local Development] --&gt; B[Unit Tests]
        A --&gt; C[Integration Tests]
        A --&gt; D[Local E2E Tests]
    end

    subgraph &quot;CI/CD Environment&quot;
        E[GitHub Actions] --&gt; F[Test Runners]
        F --&gt; G[Jest Runner]
        F --&gt; H[Pytest Runner]
        F --&gt; I[Maestro Runner]

        G --&gt; J[Mobile Test Results]
        H --&gt; K[Backend Test Results]
        I --&gt; L[E2E Test Results]
```

```
    end

    subgraph &quot;Test Infrastructure&quot;
        M[MongoDB Test Instance] --&gt; N[Test Data]
        O[Redis Test Instance] --&gt; P[Cache Testing]
        Q[Mock Services] --&gt; R[TradingView Mock]
        Q --&gt; S[Broker API Mock]
    end

    subgraph &quot;Staging Environment&quot;
        T[Staging Deployment] --&gt; U[Smoke Tests]
        T --&gt; V[Integration Validation]
        T --&gt; W[Performance Testing]
    end

    F --&gt; M
    F --&gt; O
    F --&gt; Q
    U --&gt; X[Production Readiness]
    V --&gt; X
    W --&gt; X
</div>
        </div>

#### 6.6.4.3 Test Data Flow Diagram

<div class="mermaid-wrapper" id="mermaid-diagram-awdin920z">
          <div class="mermaid">
sequenceDiagram
    participant Dev as Developer
    participant CI as CI/CD Pipeline
    participant TestDB as Test Database
    participant MockAPI as Mock Services
    participant TestRunner as Test Runner
    participant Reports as Test Reports

    Dev-&gt;&gt;CI: Code Commit
    CI-&gt;&gt;TestDB: Setup Test Data
    CI-&gt;&gt;MockAPI: Configure Mocks
    CI-&gt;&gt;TestRunner: Execute Unit Tests
    TestRunner-&gt;&gt;TestDB: Query Test Data
    TestRunner-&gt;&gt;Reports: Unit Test Results
```

```
    CI-&gt;&gt;TestRunner: Execute Integration Tests
    TestRunner-&gt;&gt;TestDB: Integration Queries
    TestRunner-&gt;&gt;MockAPI: API Calls
    MockAPI--&gt;&gt;TestRunner: Mock Responses
    TestRunner-&gt;&gt;Reports: Integration Results

    CI-&gt;&gt;TestRunner: Execute E2E Tests
    TestRunner-&gt;&gt;MockAPI: External Service Calls
    TestRunner-&gt;&gt;TestDB: End-to-End Data Flow
    TestRunner-&gt;&gt;Reports: E2E Test Results

    Reports-&gt;&gt;CI: Aggregate Test Results
    CI-&gt;&gt;Dev: Test Status Notification

    Note over TestDB,MockAPI: Test data isolation and cleanup
    Note over TestRunner,Reports: Comprehensive test coverage reporting
</div>
        </div>

The testing strategy provides a comprehensive framework for ensuring the

# 7. USER INTERFACE DESIGN

## 7.1 CORE UI TECHNOLOGIES

### 7.1.1 Frontend Framework Stack

**React Native 0.81 with Expo SDK 54 Architecture:**
@expo/ui is a set of native input components that allows you to build ful

| Technology | Version | Purpose | Key Features |
|---|---|---|---|
| React Native | 0.81 | Cross-platform mobile framework | The legacy arc
| Expo SDK | 54 | Development platform | This library is currently in alp
| React | 19.1 | UI library | Modern concurrent features and improved per
| TypeScript | 5.x | Type safety | Enhanced developer experience and code

**UI Component Libraries:**
React Native Paper is one of the most widely used UI libraries. It offers

### 7.1.2 Styling and Design System

**NativeWind Integration:**
```

```
The starter comes with a set of basic components and a simple design syst

**Design System Components:**
- **Color Palette**: Dark and light theme support with financial-focused
- **Typography**: Consistent font hierarchy optimized for financial data
- **Spacing System**: 8px grid system for consistent layout spacing
- **Component Variants**: Standardized button, input, and card variations

### 7.1.3 Chart and Visualization Libraries

**Financial Chart Integration:**
The library currently supports only line and candlestick charts, but the

| Library | Purpose | Chart Types | Performance |
|---|---|---|---|
| React Native Gifted Charts | Primary charting solution | Line, Bar, Are
| Victory Native | Secondary charts | The React Native variant is known a
| React Native Chart Kit | Simple visualizations | React Native Chart Kit

## 7.2 UI USE CASES

### 7.2.1 Primary User Workflows

**Market Analysis Workflow:**
1. **Dashboard Overview**: Real-time market data display with customizabl
2. **Symbol Selection**: Currency pair picker with search and favorites f
3. **Chart Analysis**: Interactive candlestick charts with technical indi
4. **Strategy Generation**: ML-powered trading strategy recommendations
5. **Risk Assessment**: Visual risk metrics and position sizing calculati

**Trading Strategy Workflow:**
1. **Strategy Creation**: Guided strategy builder with parameter selectio
2. **Backtesting Interface**: Historical performance visualization
3. **Risk Management**: Stop-loss and take-profit configuration
4. **Execution Planning**: Order type selection and timing preferences
5. **Performance Monitoring**: Real-time strategy tracking and alerts

### 7.2.2 User Interaction Patterns

**Touch Interactions:**
- **Swipe Navigation**: Horizontal swipe between chart timeframes
- **Pinch-to-Zoom**: Chart magnification for detailed analysis
- **Long Press**: Context menus for quick actions
```

```
- **Pull-to-Refresh**: Data synchronization triggers
- **Haptic Feedback**: Haptic feedback â€“ Enhance the user experience wi

**Gesture-Based Controls:**
- **Chart Panning**: Smooth horizontal scrolling through historical data
- **Multi-touch Zoom**: Precise chart region selection
- **Swipe Actions**: Quick access to favorite pairs and strategies
- **Drag-and-Drop**: Customizable dashboard widget arrangement

## 7.3 UI/BACKEND INTERACTION BOUNDARIES

### 7.3.1 API Integration Points

**Real-time Data Flow:**
<div class="mermaid-wrapper" id="mermaid-diagram-nwmyqs9s2">
          <div class="mermaid">
graph TB
    subgraph &amp;quot;Mobile UI Layer&amp;quot;
        A[Dashboard Screen] --&amp;gt; B[Chart Component]
        A --&amp;gt; C[Watchlist Component]
        A --&amp;gt; D[Strategy Cards]
    end

    subgraph &amp;quot;State Management&amp;quot;
        E[Redux Store] --&amp;gt; F[Market Data Slice]
        E --&amp;gt; G[Strategy Slice]
        E --&amp;gt; H[User Preferences]
    end

    subgraph &amp;quot;API Layer&amp;quot;
        I[WebSocket Client] --&amp;gt; J[Market Data Stream]
        K[REST Client] --&amp;gt; L[Strategy API]
        K --&amp;gt; M[User API]
    end

    subgraph &amp;quot;Backend Services&amp;quot;
        N[TradingView Integration] --&amp;gt; O[Market Data Service]
        P[ML Prediction Service] --&amp;gt; Q[Analytics Engine]
        R[User Management] --&amp;gt; S[Authentication Service]
    end

    B --&amp;gt; F
    C --&amp;gt; F
```

```
    D --&gt; G

    F --&gt; I
    G --&gt; K
    H --&gt; K

    J --&gt; N
    L --&gt; P
    M --&gt; R
</div>
        </div>

### 7.3.2 Data Synchronization Patterns

**Optimistic Updates:**
- Immediate UI feedback for user actions
- Background API calls with conflict resolution
- Rollback mechanisms for failed operations
- Loading states and error handling

**Real-time Synchronization:**
- WebSocket connections for live market data
- Automatic reconnection with exponential backoff
- Data validation and integrity checks
- Offline queue management

## 7.4 UI SCHEMAS

### 7.4.1 Component Architecture

**Screen-Level Components:**
```

typescript
interface DashboardScreenProps {
navigation: NavigationProp;
route: RouteProp;
}

interface MarketDataState {
symbols: CurrencyPair[];
prices: Record;

loading: boolean;
error: string | null;
}

interface TradingStrategyState {
strategies: TradingStrategy[];
activeStrategy: TradingStrategy | null;
backtestResults: BacktestResult[];
generating: boolean;
}

```
**Component Props Schema:**
```

typescript
interface ChartComponentProps {
symbol: string;
timeframe: Timeframe;
data: CandlestickData[];
indicators: TechnicalIndicator[];
onTimeframeChange: (timeframe: Timeframe) => void;
onIndicatorToggle: (indicator: string) => void;
}

interface StrategyCardProps {
strategy: TradingStrategy;
onExecute: (strategyId: string) => void;
onEdit: (strategyId: string) => void;
onDelete: (strategyId: string) => void;
}

```
### 7.4.2 Data Models

**Market Data Models:**
```

typescript
interface CurrencyPair {
symbol: string;
baseAsset: string;
quoteAsset: string;
displayName: string;
precision: number;
}

interface PriceData {
symbol: string;
price: number;
change: number;
changePercent: number;
volume: number;
timestamp: Date;
}

interface CandlestickData {
timestamp: Date;
open: number;
high: number;
low: number;
close: number;
volume: number;
}

```
**Trading Strategy Models:**
```

typescript
interface TradingStrategy {
id: string;
name: string;
symbol: string;

timeframe: Timeframe;
entryConditions: EntryCondition[];
exitConditions: ExitCondition[];
riskManagement: RiskParameters;
mlPrediction: MLPrediction;
backtestResults: BacktestResult;
createdAt: Date;
updatedAt: Date;
}

interface MLPrediction {
direction: 'BUY' | 'SELL' | 'HOLD';
confidence: number;
probability: {
buy: number;
sell: number;
hold: number;
};
modelVersion: string;
features: Record;
}

```
## 7.5 SCREENS REQUIRED

### 7.5.1 Core Application Screens

**Authentication Flow:**
1. **Welcome Screen**: App introduction and feature highlights
2. **Login Screen**: Email/password authentication with biometric options
3. **Registration Screen**: Account creation with email verification
4. **Forgot Password Screen**: Password reset functionality

**Main Application Screens:**
1. **Dashboard Screen**: An adaptive portfolio dashboard empowers users t
2. **Market Analysis Screen**: This includes features such as customizabl
3. **Strategy Builder Screen**: Guided strategy creation with ML recommen
4. **Portfolio Screen**: Trading performance and position management
```

```
5. **Settings Screen**: User preferences and account management

### 7.5.2 Screen Specifications

**Dashboard Screen Layout:**
```

â”Œâ”€
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
â”‚ Header: Portfolio Value & P/L â”‚
â”œâ”€
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¤
â”‚ Quick Stats Cards (3x1 grid) â”‚
â”‚ [Win Rate] [Active] [Today P/L] â”‚
â”œâ”€
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¤
â”‚ Market Overview â”‚
â”‚
â”Œâ”€
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â” â”‚
â”‚ â”‚ Major Pairs Price Ticker â”‚ â”‚
â”‚
â””â”€â
”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜ â”‚
â”œâ”€
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¤
â”‚ Active Strategies (Scrollable) â”‚
â”‚ â”Œâ”€â”€â”€â”€â”€â” â”Œâ”€â”€â”€â”€â”€â”
â”Œâ”€â”€â”€â”€â”€â” â”Œâ”€â”€â”€â”€â”€â” â”‚
â”‚ â”‚Card1â”‚ â”‚Card2â”‚ â”‚Card3â”‚ â”‚Card4â”‚ â”‚
â”‚ â””â”€â”€â”€â”€â”€â”˜ â””â”€â”€â”€â”€â”€â”˜
â””â”€â”€â”€â”€â”€â”˜ â””â”€â”€â”€â”€â”€â”˜ â”‚
â”œâ”€
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¤
â”‚ Recent Alerts â”‚

â”‚ â€¢ EURUSD: Strong Buy Signal â”‚
â”‚ â€¢ GBPJPY: Risk Warning â”‚
â””â”€â
”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜

```
**Market Analysis Screen Layout:**
```

â”Œâ”€
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”
â”‚ Symbol Selector & Timeframe â”‚
â”‚ [EURUSD â–¼] [1H] [4H] [1D] [1W] â”‚
â”œâ”€
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¤
â”‚ â”‚
â”‚ Candlestick Chart â”‚
â”‚ (Interactive with zoom) â”‚
â”‚ â”‚
â”‚ â”‚
â”œâ”€
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¤
â”‚ Technical Indicators Panel â”‚
â”‚ [RSI] [MACD] [BB] [MA] [+More] â”‚
â”œâ”€
â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”¤
â”‚ ML Prediction Panel â”‚
â”‚ Direction: BUY | Confidence: 78% â”‚
â”‚ [Generate Strategy] [View Details] â”‚
â””â”€â
”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”€â”˜

```
### 7.5.3 Responsive Design Considerations

**Screen Size Adaptations:**
- **Small Phones (< 5.5")**: Compact layouts with collapsible sections
```

```
- **Standard Phones (5.5" - 6.5")**: Optimized primary layout
- **Large Phones (> 6.5")**: Enhanced information density
- **Tablets**: Multi-column layouts with expanded chart areas

**Orientation Support:**
- **Portrait Mode**: Primary navigation and standard layouts
- **Landscape Mode**: Chart-focused layouts with expanded visualization a
- **Dynamic Rotation**: Smooth transitions between orientations

## 7.6 USER INTERACTIONS

### 7.6.1 Navigation Patterns

**Tab-Based Navigation:**
```

typescript
const TabNavigator = createBottomTabNavigator({
Dashboard: {
screen: DashboardScreen,
options: {
tabBarIcon: ({ color, size }) => (

),
},
},
Markets: {
screen: MarketsScreen,
options: {
tabBarIcon: ({ color, size }) => (

),
},
},
Strategies: {
screen: StrategiesScreen,
options: {

tabBarIcon: ({ color, size }) => (

),
},
},
Portfolio: {
screen: PortfolioScreen,
options: {
tabBarIcon: ({ color, size }) => (

),
},
},
});

```
**Stack Navigation:**
- **Modal Presentations**: Strategy details, settings, and alerts
- **Push Navigation**: Drill-down flows for detailed analysis
- **Gesture Navigation**: Swipe-back support for iOS-style navigation

### 7.6.2 Interactive Elements

**Chart Interactions:**
The tooltip and crosshair cursor allow users to interact with the chart a

**Touch Gestures:**
- **Single Tap**: Select data points, activate buttons
- **Double Tap**: Quick zoom on charts
- **Long Press**: Context menus and detailed information
- **Swipe**: Navigate between screens and dismiss modals
- **Pinch**: Chart zoom and scale adjustments

### 7.6.3 Feedback Mechanisms

**Visual Feedback:**
- **Loading States**: Skeleton screens and progress indicators
- **Success States**: Checkmarks and positive color changes
- **Error States**: Red highlights and error messages
- **Disabled States**: Reduced opacity and interaction blocking
```

```
**Haptic Feedback:**
- **Success Actions**: Light impact feedback
- **Error Actions**: Heavy impact feedback
- **Selection**: Selection feedback for buttons and toggles
- **Chart Interactions**: Subtle feedback for data point selection

## 7.7 VISUAL DESIGN CONSIDERATIONS

### 7.7.1 Color Scheme and Theming

**Financial-Focused Color Palette:**
```

typescript
const theme = {
colors: {
// Primary colors
primary: '#1E88E5',
primaryDark: '#1565C0',
secondary: '#FFC107',

```
// Financial colors
bullish: '#4CAF50',    // Green for positive/buy
bearish: '#F44336',    // Red for negative/sell
neutral: '#9E9E9E',    // Gray for neutral/hold

// Background colors
background: '#FFFFFF',
backgroundDark: '#121212',
surface: '#F5F5F5',
surfaceDark: '#1E1E1E',

// Text colors
onBackground: '#212121',
onBackgroundDark: '#FFFFFF',
onSurface: '#424242',
onSurfaceDark: '#E0E0E0',

// Status colors
success: '#4CAF50',
```

```
  warning: '#FF9800',
  error: '#F44336',
  info: '#2196F3',
```

},
};

```
  **Dark Mode Support:**
  â€”ï¸ Dark mode support The application provides comprehensive dark mode
  - Automatic system theme detection
  - Manual theme toggle in settings
  - Consistent color mapping across all components
  - Optimized contrast ratios for financial data readability

  ### 7.7.2 Typography System

  **Font Hierarchy:**
```

typescript
const typography = {
// Headers
h1: {
fontSize: 32,
fontWeight: '700',
lineHeight: 40,
letterSpacing: -0.5,
},
h2: {
fontSize: 24,
fontWeight: '600',
lineHeight: 32,
letterSpacing: -0.25,
},
h3: {
fontSize: 20,

```
fontWeight: '600',
lineHeight: 28,
},

// Body text
body1: {
fontSize: 16,
fontWeight: '400',
lineHeight: 24,
},
body2: {
fontSize: 14,
fontWeight: '400',
lineHeight: 20,
},

// Financial data
price: {
fontSize: 18,
fontWeight: '600',
fontFamily: 'monospace', // For consistent number alignment
},
percentage: {
fontSize: 14,
fontWeight: '500',
fontFamily: 'monospace',
},
};
```

```
### 7.7.3 Layout and Spacing

**Grid System:**
- **Base Unit**: 8px grid system for consistent spacing
- **Component Spacing**: 16px standard margin between components
- **Section Spacing**: 24px between major sections
```

```
- **Screen Padding**: 16px horizontal padding on mobile

**Component Sizing:**
- **Touch Targets**: Minimum 44px height for interactive elements
- **Button Heights**: 48px for primary actions, 40px for secondary
- **Input Fields**: 56px height with proper padding
- **Card Components**: 8px border radius with subtle shadows

### 7.7.4 Accessibility Features

**Visual Accessibility:**
- **Color Contrast**: WCAG AA compliance with 4.5:1 contrast ratios
- **Font Scaling**: Support for system font size preferences
- **Focus Indicators**: Clear visual focus states for keyboard navigatio
- **Color Independence**: Information not conveyed by color alone

**Interaction Accessibility:**
- **Screen Reader Support**: Proper semantic markup and labels
- **Voice Control**: Compatible with voice navigation systems
- **Keyboard Navigation**: Full keyboard accessibility support
- **Reduced Motion**: Respect for system motion preferences

### 7.7.5 Animation and Transitions

**Micro-Interactions:**
```

typescript
const animations = {
// Screen transitions
screenTransition: {
duration: 300,
easing: 'ease-out',
},

// Component animations
fadeIn: {
duration: 200,
easing: 'ease-in',
},

slideUp: {
duration: 250,
easing: 'ease-out',
},

// Chart animations
chartUpdate: {
duration: 500,
easing: 'ease-in-out',
},

// Loading states
skeleton: {
duration: 1000,
easing: 'ease-in-out',
repeat: true,
},
};

```
**Performance Considerations:**
- **Native Driver**: Use native animations where possible
- **Gesture Handling**: Smooth 60fps gesture responses
- **Memory Management**: Proper cleanup of animation resources
- **Battery Optimization**: Reduced animations on low battery mode

### 7.7.6 Chart Visualization Design

**Candlestick Chart Styling:**
The wicks of the candlestick chart show the high and low trade prices (H

**Technical Indicator Visualization:**
- **Moving Averages**: Smooth colored lines with transparency
- **RSI**: Oscillator with overbought/oversold zones
- **MACD**: Histogram and signal line visualization
- **Bollinger Bands**: Shaded area between upper and lower bands

The user interface design provides a comprehensive, modern, and accessib
```

```
# 8. INFRASTRUCTURE

## 8.1 DEPLOYMENT ENVIRONMENT

### 8.1.1 Target Environment Assessment

**Environment Type: Hybrid Cloud Architecture**

The forex trading application employs a hybrid cloud deployment strategy

**Geographic Distribution Requirements:**

| Region | Primary Services | Latency Requirements | Compliance Considera
|---|---|---|---|
| US East (N. Virginia) | Primary backend, ML services | <100ms to major
| EU West (Ireland) | GDPR-compliant data processing | <150ms to Europea
| Asia Pacific (Singapore) | Regional market data processing | <200ms to

**Resource Requirements:**

| Component | Compute Requirements | Memory Requirements | Storage Requi
|---|---|---|---|---|
| Flask API Services | 4-8 vCPUs per instance | 8-16 GB RAM | 100 GB SSD
| ML Model Services | 8-16 vCPUs + GPU | 16-32 GB RAM | 500 GB SSD | 25 G
| Database Cluster | 8 vCPUs per node | 32 GB RAM | 1 TB SSD | 10 Gbps ba
| Cache Layer | 4 vCPUs | 16 GB RAM | 200 GB SSD | 10 Gbps bandwidth |

**Compliance and Regulatory Requirements:**

The system must adhere to multiple financial regulations including MiFID

### 8.1.2 Environment Management

**Infrastructure as Code (IaC) Approach:**

Terraform is an open source infrastructure as code (IaC) software tool de

**Terraform Configuration Structure:**
```

infrastructure/
â”œâ”€â”€ environments/
â”‚ â”œâ”€â”€ development/

â”‚ â”œâ”€â”€ staging/
â”‚ â””â”€â”€ production/
â”œâ”€â”€ modules/
â”‚ â”œâ”€â”€ vpc/
â”‚ â”œâ”€â”€ ecs/
â”‚ â”œâ”€â”€ rds/
â”‚ â””â”€â”€ elasticache/
â”œâ”€â”€ shared/
â”‚ â”œâ”€â”€ variables.tf
â”‚ â”œâ”€â”€ outputs.tf
â”‚ â””â”€â”€ providers.tf
â””â”€â”€ scripts/
â”œâ”€â”€ deploy.sh
â””â”€â”€ destroy.sh

```
**Configuration Management Strategy:**

| Environment | Configuration Method | Secrets Management | Deployment Tr
|---|---|---|---|
| Development | Terraform + local state | AWS Secrets Manager | Manual de
| Staging | Terraform + S3 backend | AWS Secrets Manager | Automated on P
| Production | Terraform + S3 backend + DynamoDB locking | AWS Secrets Ma

**Environment Promotion Strategy:**

<div class="mermaid-wrapper" id="mermaid-diagram-06n0vvp5z">
          <div class="mermaid">
graph TB
    subgraph &quot;Development Environment&quot;
        A[Local Development] --&gt; B[**Feature** Branch]
        B --&gt; C[Development Deployment]
    end

    subgraph &quot;Staging Environment&quot;
        D[Pull Request Merge] --&gt; E[Staging Deployment]
        E --&gt; F[Integration Testing]
        F --&gt; G[Performance Testing]
    end
```

```
    subgraph &quot;Production Environment&quot;
        H[Release Approval] --&gt; I[Production Deployment]
        I --&gt; J[Health Checks]
        J --&gt; K[Monitoring Activation]
    end

    C --&gt; D
    G --&gt; H
    K --&gt; L[Live System]
</div>
        </div>

**Backup and Disaster Recovery Plans:**

| Component | Backup Frequency | Retention Period | Recovery Time Objecti
|---|---|---|---|---|
| Application Database | Continuous + Daily snapshots | 30 days | <4 hou
| User Data | Real-time replication | 7 years (compliance) | <1 hour | <
| Application Code | Git-based versioning | Permanent | <30 minutes | 0
| Infrastructure State | Daily Terraform state backup | 90 days | <2 hou

## 8.2 CLOUD SERVICES

### 8.2.1 Cloud Provider Selection and Justification

**Primary Cloud Provider: Amazon Web Services (AWS)**

AWS has been selected as the primary cloud provider based on comprehensiv

**Secondary Cloud Provider: Expo Application Services (EAS)**

EAS Build is a hosted service for building app binaries for your Expo and

### 8.2.2 Core Services Required with Versions

**AWS Core Services:**

| Service | Version/Type | Purpose | Configuration |
|---|---|---|---|
| Amazon ECS | Fargate | Container orchestration | Auto-scaling enabled
| Amazon RDS | PostgreSQL 15.x | Primary database | Multi-AZ deployment
| Amazon ElastiCache | Redis 7.0 | Caching and sessions | Cluster mode en
| Amazon S3 | Standard | File storage and backups | Versioning enabled |
```

```
| Amazon CloudWatch | Latest | Monitoring and logging | Custom metrics er
| AWS Application Load Balancer | ALBv2 | Load balancing | SSL terminatic
| Amazon VPC | Latest | Network isolation | Multi-AZ subnets |
| AWS Secrets Manager | Latest | Secrets management | Automatic rotation

**Expo Application Services:**

| Service | Tier | Purpose | Limitations |
|---|---|---|---|
| EAS Build | Free/Paid | Mobile app compilation | The best way to start
| EAS Submit | Included | App store submission | Submit your app to the a
| EAS Update | Usage-based | Over-the-air updates | Send over-the-air upd

### 8.2.3 High Availability Design

**Multi-AZ Deployment Architecture:**

<div class="mermaid-wrapper" id="mermaid-diagram-k1730to00">
          <div class="mermaid">
graph TB
    subgraph &quot;AWS Region: us-east-1&quot;
        subgraph &quot;Availability Zone A&quot;
            A1[ECS Tasks] --&gt; A2[RDS Primary]
            A3[ElastiCache Primary] --&gt; A4[NAT Gateway A]
        end

        subgraph &quot;Availability Zone B&quot;
            B1[ECS Tasks] --&gt; B2[RDS Standby]
            B3[ElastiCache Replica] --&gt; B4[NAT Gateway B]
        end

        subgraph &quot;Availability Zone C&quot;
            C1[ECS Tasks] --&gt; C2[RDS Read Replica]
            C3[ElastiCache Replica] --&gt; C4[NAT Gateway C]
        end
    end

    subgraph &quot;Global Services&quot;
        D[Application Load Balancer] --&gt; A1
        D --&gt; B1
        D --&gt; C1
        E[CloudFront CDN] --&gt; F[S3 Static Assets]
        G[Route 53 DNS] --&gt; D
```

```
    end
</div>
        </div>

**High Availability Configuration:**

| Component | Availability Target | Failover Time | Backup Strategy |
|---|---|---|---|
| Application Load Balancer | 99.99% | <30 seconds | Multi-AZ automatic
| ECS Fargate Tasks | 99.95% | <2 minutes | Auto-scaling replacement |
| RDS Database | 99.95% | <60 seconds | Multi-AZ failover |
| ElastiCache | 99.9% | <30 seconds | Cluster failover |

### 8.2.4 Cost Optimization Strategy

**Resource Right-Sizing:**

| Service | Development | Staging | Production | Cost Optimization |
|---|---|---|---|---|
| ECS Fargate | 0.25 vCPU, 0.5 GB | 0.5 vCPU, 1 GB | 1-4 vCPU, 2-8 GB | A
| RDS Instance | db.t3.micro | db.t3.small | db.r5.large | Reserved insta
| ElastiCache | cache.t3.micro | cache.t3.small | cache.r5.large | Reserv

**Estimated Monthly Costs:**

| Environment | Compute | Database | Storage | Network | Total |
|---|---|---|---|---|---|
| Development | $50 | $25 | $10 | $5 | $90 |
| Staging | $150 | $75 | $25 | $15 | $265 |
| Production | $800 | $400 | $100 | $100 | $1,400 |

### 8.2.5 Security and Compliance Considerations

**AWS Security Services:**

| Security Layer | Service | Configuration | Compliance Benefit |
|---|---|---|---|
| Network Security | VPC + Security Groups | Least privilege access | Net
| Data Encryption | KMS + SSL/TLS | Encryption at rest and in transit | D
| Identity Management | IAM + Secrets Manager | Role-based access contro
| Monitoring | CloudTrail + GuardDuty | Comprehensive logging | Complianc

## 8.3 CONTAINERIZATION
```

```
### 8.3.1 Container Platform Selection

**Docker with Amazon ECS Fargate**

In this part I'm going to cover how to deploy the application in Docker c

**Container Strategy Rationale:**

| Requirement | Docker Solution | Alternative | Decision Rationale |
|---|---|---|---|
| Backend Services | Multi-container Flask apps | VM-based deployment | E
| Development Consistency | Docker Compose | Local installations | Enviro
| CI/CD Integration | Container builds | Direct deployment | Immutable de
| Microservices Architecture | Service isolation | Monolithic deployment

### 8.3.2 Base Image Strategy

**Python Flask Services Base Images:**
```

dockerfile

Multi-stage build for Flask services

FROM python:3.11-slim as builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir --user -r requirements.txt

FROM python:3.11-slim as runtime
WORKDIR /app
COPY --from=builder /root/.local /root/.local
COPY . .
ENV PATH=/root/.local/bin:$PATH

EXPOSE 5000
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app:app"]

```
**Base Image Selection Criteria:**

| Image Type | Size | Security Updates | Use Case |
|---|---|---|---|
| python:3.11-slim | ~45 MB | Regular | Production Flask services |
| python:3.11-alpine | ~25 MB | Regular | Lightweight services |
| python:3.11 | ~350 MB | Regular | Development with full toolchain |

### 8.3.3 Image Versioning Approach

**Semantic Versioning Strategy:**

<div class="mermaid-wrapper" id="mermaid-diagram-apdcuyvhn">
          <div class="mermaid">
graph LR
    A[Git Commit] --&gt; B[Build Trigger]
    B --&gt; C{Branch Type}
    C --&gt;|main| D[Production Tag: v1.2.3]
    C --&gt;|develop| E[Staging Tag: v1.2.3-rc.1]
    C --&gt;|feature| F[Development Tag: v1.2.3-feature.branch]

    D --&gt; G[Production Registry]
    E --&gt; H[Staging Registry]
    F --&gt; I[Development Registry]
</div>
        </div>

**Image Tagging Convention:**

| Environment | Tag Format | Example | Retention Policy |
|---|---|---|---|
| Production | v{major}.{minor}.{patch} | v1.2.3 | Keep last 10 versions
| Staging | v{version}-rc.{build} | v1.2.3-rc.1 | Keep last 5 versions |
| Development | v{version}-{branch}.{commit} | v1.2.3-feature.abc123 | Ke

### 8.3.4 Build Optimization Techniques

**Multi-Stage Build Strategy:**
```

```
The second build step uses the Python 3.9 container as a base. It puts tl

**Docker Build Optimization:**

| Optimization Technique | Implementation | Benefit |
|---|---|---|
| Layer Caching | Copy requirements.txt first | Faster rebuilds when code
| Multi-stage Builds | Separate build and runtime stages | Smaller final
| .dockerignore | Exclude unnecessary files | Faster build context trans
| Dependency Caching | Use pip cache mount | Faster dependency installat

**Optimized Dockerfile Example:**
```

dockerfile

syntax=docker/dockerfile:1

FROM python:3.11-slim as dependencies
WORKDIR /app
COPY requirements.txt .
RUN --mount=type=cache,target=/root/.cache/pip \
pip install --user -r requirements.txt

FROM python:3.11-slim as runtime
WORKDIR /app
COPY --from=dependencies /root/.local /root/.local
COPY . .
ENV PATH=/root/.local/bin:$PATH
HEALTHCHECK --interval=30s --timeout=3s --start-period=5s --retries=3 \
CMD curl -f http://localhost:5000/health || exit 1
EXPOSE 5000
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "--workers", "4", "app:app"]

```
### 8.3.5 Security Scanning Requirements
```

```
**Container Security Pipeline:**

<div class="mermaid-wrapper" id="mermaid-diagram-vp795jljo">
          <div class="mermaid">
graph TB
    A[Docker Build] --&gt; B[Base Image Scan]
    B --&gt; C[Dependency Scan]
    C --&gt; D[Static Analysis]
    D --&gt; E{Security Gate}
    E --&gt;|Pass| F[Push to Registry]
    E --&gt;|Fail| G[Block Deployment]
    F --&gt; H[Runtime Monitoring]
    G --&gt; I[Security Report]
</div>
        </div>

**Security Scanning Tools:**

| Scan Type | Tool | Frequency | Threshold |
|---|---|---|---|
| Base Image Vulnerabilities | AWS ECR Image Scanning | Every build | No
| Dependency Vulnerabilities | Snyk | Every build | No high-severity vul
| Configuration Issues | Hadolint | Every build | No errors |
| Runtime Security | AWS GuardDuty | Continuous | Real-time alerts |

## 8.4 ORCHESTRATION

### 8.4.1 Orchestration Platform Selection

**Amazon ECS with Fargate**

The application will be hosted on an Amazon ECS cluster running on Amazo

**ECS Fargate vs. Kubernetes Comparison:**

| Factor | ECS Fargate (Selected) | Amazon EKS | Decision Rationale |
|---|---|---|---|
| Management Overhead | Fully managed | Requires cluster management | Re
| Cost | Pay per task | Pay for control plane + nodes | Better cost effi
| Learning Curve | AWS-native, simpler | Kubernetes complexity | Faster
| Vendor Lock-in | AWS-specific | Portable | Acceptable for AWS-first st

### 8.4.2 Cluster Architecture
```

```
**ECS Cluster Design:**

<div class="mermaid-wrapper" id="mermaid-diagram-2kvupzcug">
          <div class="mermaid">
graph TB
    subgraph &quot;ECS Cluster&quot;
        subgraph &quot;Public Subnets&quot;
            A[Application Load Balancer] --&gt; B[Target Groups]
        end

        subgraph &quot;Private Subnets - AZ A&quot;
            C[Flask API Tasks] --&gt; D[ML Service Tasks]
            E[Background Worker Tasks]
        end

        subgraph &quot;Private Subnets - AZ B&quot;
            F[Flask API Tasks] --&gt; G[ML Service Tasks]
            H[Background Worker Tasks]
        end

        subgraph &quot;Data Layer&quot;
            I[RDS Multi-AZ] --&gt; J[ElastiCache Cluster]
            K[S3 Storage] --&gt; L[Secrets Manager]
        end
    end

    B --&gt; C
    B --&gt; F
    C --&gt; I
    F --&gt; I
    D --&gt; J
    G --&gt; J
</div>
        </div>

**Service Configuration:**

| Service | CPU | Memory | Desired Count | Max Count | Health Check |
|---|---|---|---|---|---|
| Flask API | 1024 | 2048 | 2 | 10 | /health endpoint |
| ML Service | 2048 | 4096 | 1 | 5 | /health endpoint |
| Background Workers | 512 | 1024 | 1 | 3 | Custom health check |
```

```
### 8.4.3 Service Deployment Strategy

**Blue-Green Deployment with ECS:**

AWS CodeDeploy kicks off a Blue / Green deployment and deploys the latest

**Deployment Configuration:**
```

yaml

ECS Service Definition

apiVersion: v1
kind: Service
metadata:
name: flask-api-service
spec:
deploymentConfiguration:
deploymentCircuitBreaker:
enable: true
rollback: true
maximumPercent: 200
minimumHealthyPercent: 50
desiredCount: 2
enableExecuteCommand: true
healthCheckGracePeriodSeconds: 60
loadBalancers:
- targetGroupArn: !Ref TargetGroup
containerName: flask-api
containerPort: 5000

```
**Deployment Strategies:**
```

```
| Strategy | Use Case | Rollback Time | Risk Level |
|---|---|---|---|
| Rolling Update | Regular updates | 5-10 minutes | Low |
| Blue-Green | Critical updates | <2 minutes | Very Low |
| Canary | High-risk changes | Variable | Minimal |

### 8.4.4 Auto-scaling Configuration

**ECS Auto Scaling Policies:**

<div class="mermaid-wrapper" id="mermaid-diagram-xczexog24">
          <div class="mermaid">
graph TB
    A[CloudWatch Metrics] --&gt; B{Scaling Decision}
    B --&gt;|Scale Up| C[Increase Task Count]
    B --&gt;|Scale Down| D[Decrease Task Count]
    B --&gt;|No Action| E[Maintain Current Count]

    C --&gt; F[Health Check New Tasks]
    D --&gt; G[Drain Connections]
    F --&gt; H[Add to Load Balancer]
    G --&gt; I[Terminate Tasks]
</div>
        </div>

**Auto-scaling Triggers:**

| Metric | Scale Up Threshold | Scale Down Threshold | Cooldown Period |
|---|---|---|---|
| CPU Utilization | >70% for 2 minutes | <30% for 5 minutes | 5 minutes
| Memory Utilization | >80% for 2 minutes | <40% for 5 minutes | 5 minute
| Request Count | >1000 RPM | <200 RPM | 3 minutes |
| Response Time | >2 seconds | <500ms | 5 minutes |

### 8.4.5 Resource Allocation Policies

**Resource Allocation Matrix:**

| Service Tier | CPU Allocation | Memory Allocation | Storage Allocation
|---|---|---|---|---|
| Critical (API) | Guaranteed 1 vCPU | Guaranteed 2 GB | 20 GB ephemeral
| Important (ML) | Burstable 2 vCPU | Guaranteed 4 GB | 50 GB ephemeral
| Background | Burstable 0.5 vCPU | Guaranteed 1 GB | 10 GB ephemeral | 1
```

```
**Resource Limits and Requests:**
```

yaml

Task Definition Resource Configuration

resources:
limits:
cpu: 1024
memory: 2048
requests:
cpu: 512
memory: 1024

```
## 8.5 CI/CD PIPELINE

### 8.5.1 Build Pipeline

**Source Control Triggers:**

The CI/CD pipeline integrates with GitHub Actions for automated builds a

**Build Pipeline Architecture:**

<div class="mermaid-wrapper" id="mermaid-diagram-fwu3n41bq">
          <div class="mermaid">
graph TB
    A[Git Push] --&gt; B[GitHub Actions Trigger]
    B --&gt; C[Checkout Code]
    C --&gt; D[Setup Dependencies]
    D --&gt; E[Run Tests]
    E --&gt; F{Tests Pass?}
    F --&gt;|No| G[Notify Failure]
    F --&gt;|Yes| H[Build Docker Images]
```

```
    H --&gt; I[Security Scan]
    I --&gt; J{Security Pass?}
    J --&gt;|No| K[Block Deployment]
    J --&gt;|Yes| L[Push to ECR]
    L --&gt; M[Update ECS Service]
    M --&gt; N[Health Check]
    N --&gt; O{Health OK?}
    O --&gt;|No| P[Rollback]
    O --&gt;|Yes| Q[Deployment Success]
</div>
        </div>

**Build Environment Requirements:**

| Component | Specification | Purpose |
|---|---|---|
| GitHub Actions Runner | ubuntu-latest | CI/CD execution |
| Node.js | v20.x | Frontend build tools |
| Python | 3.11 | Backend testing and building |
| Docker | Latest | Container building |
| AWS CLI | v2 | AWS service interaction |

**Dependency Management:**
```

yaml

GitHub Actions Workflow

name: CI/CD Pipeline
on:
push:
branches: [main, develop]
pull_request:
branches: [main]

jobs:
test:

runs-on: ubuntu-latest
steps:
- uses: actions/checkout@v4
- name: Setup Python
uses: actions/setup-python@v4
with:
python-version: '3.11'
cache: 'pip'
- name: Install dependencies
run: |
pip install -r requirements.txt
pip install -r requirements-dev.txt
- name: Run tests
run: pytest --cov=src --cov-report=xml

```
**Artifact Generation and Storage:**

| Artifact Type | Storage Location | Retention Policy | Access Control |
|---|---|---|---|
| Docker Images | Amazon ECR | 30 days for dev, 1 year for prod | IAM-bas
| Test Reports | GitHub Actions | 90 days | Repository access |
| Build Logs | CloudWatch Logs | 30 days | IAM-based |
| Security Scan Results | S3 Bucket | 1 year | Encrypted, restricted acce

### 8.5.2 Deployment Pipeline

**Deployment Strategy:**

The updated workflow includes the following additions: Continuous Integra

**Environment Promotion Workflow:**

<div class="mermaid-wrapper" id="mermaid-diagram-21okf1za6">
          <div class="mermaid">
graph LR
    A[Development] --&gt; B[Feature Testing]
    B --&gt; C[Pull Request]
    C --&gt; D[Staging Deployment]
    D --&gt; E[Integration Tests]
```

```
    E --&gt; F[Performance Tests]
    F --&gt; G[Security Tests]
    G --&gt; H{Approval Gate}
    H --&gt;|Approved| I[Production Deployment]
    H --&gt;|Rejected| J[Back to Development]
    I --&gt; K[Post-deployment Monitoring]
</div>
        </div>

**Deployment Configuration:**

| Environment | Deployment Method | Approval Required | Rollback Strategy
|---|---|---|---|
| Development | Automatic on push | No | Automatic on failure |
| Staging | Automatic on PR merge | No | Automatic on failure |
| Production | Manual approval | Yes | Blue-green rollback |

**Rollback Procedures:**

| Trigger | Detection Time | Rollback Time | Recovery Method |
|---|---|---|---|
| Health Check Failure | <2 minutes | <5 minutes | Automatic ECS rollbacl
| Error Rate Spike | <5 minutes | <10 minutes | Blue-green switch |
| Performance Degradation | <10 minutes | <15 minutes | Previous version
| Security Incident | Immediate | <30 minutes | Emergency rollback + inve

### 8.5.3 Post-deployment Validation

**Validation Pipeline:**

<div class="mermaid-wrapper" id="mermaid-diagram-yurruauiu">
          <div class="mermaid">
graph TB
    A[Deployment Complete] --&gt; B[Health Checks]
    B --&gt; C[Smoke Tests]
    C --&gt; D[Integration Tests]
    D --&gt; E[Performance Tests]
    E --&gt; F{All Tests Pass?}
    F --&gt;|Yes| G[Update Monitoring]
    F --&gt;|No| H[Trigger Rollback]
    G --&gt; I[Notify Success]
    H --&gt; J[Notify Failure]
    I --&gt; K[Update Documentation]
```

```
    J --&gt; L[Incident Response]
</div>
        </div>

**Validation Criteria:**

| Test Type | Success Criteria | Timeout | Failure Action |
|---|---|---|---|
| Health Checks | HTTP 200 response | 30 seconds | Immediate rollback |
| Smoke Tests | Core functionality works | 5 minutes | Rollback |
| Integration Tests | All API endpoints respond | 10 minutes | Rollback
| Performance Tests | Response time <2s | 15 minutes | Alert + monitor |

### 8.5.4 Release Management Process

**Release Planning:**

| Release Type | Frequency | Planning Window | Approval Process |
|---|---|---|---|
| Hotfix | As needed | Immediate | Emergency approval |
| Minor Release | Weekly | 1 week | Team lead approval |
| Major Release | Monthly | 2 weeks | Stakeholder approval |
| Security Release | As needed | 24 hours | Security team approval |

**Release Documentation:**

- **Release Notes**: Automated generation from commit messages and PR des
- **Deployment Guide**: Step-by-step deployment instructions
- **Rollback Plan**: Detailed rollback procedures for each release
- **Testing Results**: Comprehensive test reports and coverage metrics

## 8.6 INFRASTRUCTURE MONITORING

### 8.6.1 Resource Monitoring Approach

**Comprehensive Monitoring Stack:**

The infrastructure monitoring leverages AWS CloudWatch for core metrics 

**Monitoring Architecture:**

<div class="mermaid-wrapper" id="mermaid-diagram-4nsg8dwi9">
          <div class="mermaid">
```

```
graph TB
    subgraph &quot;Data Collection&quot;
        A[ECS Tasks] --&gt; B[CloudWatch Agent]
        C[RDS Database] --&gt; D[Performance Insights]
        E[Load Balancer] --&gt; F[ALB Metrics]
        G[Application Code] --&gt; H[Custom Metrics]
    end

    subgraph &quot;Metrics Processing&quot;
        B --&gt; I[CloudWatch Metrics]
        D --&gt; I
        F --&gt; I
        H --&gt; I
        I --&gt; J[CloudWatch Alarms]
    end

    subgraph &quot;Alerting &amp; Visualization&quot;
        J --&gt; K[SNS Notifications]
        I --&gt; L[CloudWatch Dashboards]
        K --&gt; M[PagerDuty]
        K --&gt; N[Slack Alerts]
        L --&gt; O[Grafana Dashboards]
    end
</div>
        </div>

**Key Performance Indicators:**

| Metric Category | Metrics | Target Values | Alert Thresholds |
|---|---|---|---|
| Application Performance | Response time, throughput, error rate | <2s,
| Infrastructure Health | CPU, memory, disk usage | <70%, <80%, <85% | >8
| Database Performance | Connection count, query time | <80% max, <100ms
| Network Performance | Latency, packet loss | <100ms, <0.1% | >500ms, >1

### 8.6.2 Performance Metrics Collection

**CloudWatch Custom Metrics:**
```

python
import boto3

import time
from datetime import datetime

class MetricsCollector:
def **init**(self):
self.cloudwatch = boto3.client('cloudwatch')

```
def put_custom_metric(self, metric_name, value, unit='Count', namespace=
    """Send custom metric to CloudWatch"""
    try:
        self.cloudwatch.put_metric_data(
            Namespace=namespace,
            MetricData=[
                {
                    'MetricName': metric_name,
                    'Value': value,
                    'Unit': unit,
                    'Timestamp': datetime.utcnow()
                }
            ]
        )
    except Exception as e:
        print(f"Failed to send metric {metric_name}: {e}")

def record_api_response_time(self, endpoint, response_time):
    """Record API response time metric"""
    self.put_custom_metric(
        f'API.ResponseTime.{endpoint}',
        response_time,
        'Milliseconds'
    )

def record_trading_strategy_generation(self, success=True):
    """Record trading strategy generation metrics"""
    metric_name = 'TradingStrategy.Generated.Success' if success else 'T
    self.put_custom_metric(metric_name, 1)
```

```
**Performance Monitoring Configuration:**

| Service | Metrics Collected | Collection Frequency | Retention Period
```

```
|---|---|---|---|
| ECS Tasks | CPU, memory, network I/O | 1 minute | 15 months |
| RDS Database | Connections, queries, locks | 1 minute | 15 months |
| ElastiCache | Hit ratio, evictions, connections | 1 minute | 15 months
| Application Load Balancer | Request count, latency, errors | 1 minute

### 8.6.3 Cost Monitoring and Optimization

**AWS Cost Management:**

| Cost Category | Monthly Budget | Alert Threshold | Optimization Strateg
|---|---|---|---|
| Compute (ECS) | $800 | 80% of budget | Right-sizing, reserved capacity
| Database (RDS) | $400 | 80% of budget | Reserved instances, read replic
| Storage (S3) | $100 | 90% of budget | Lifecycle policies, compression
| Network | $100 | 85% of budget | CloudFront caching, data transfer opti

**Cost Optimization Automation:**
```

python

AWS Cost Optimization Lambda Function

import boto3
import json

def lambda_handler(event, context):
"""Automated cost optimization checks"""

```
# Check for unused resources
ec2 = boto3.client('ec2')
rds = boto3.client('rds')

recommendations = []

# Find stopped instances running for >7 days
```

```
stopped_instances = ec2.describe_instances(
    Filters=[{'Name': 'instance-state-name', 'Values': ['stopped']}]
)

# Find underutilized RDS instances
db_instances = rds.describe_db_instances()

# Generate cost optimization report
return {
    'statusCode': 200,
    'body': json.dumps({
        'recommendations': recommendations,
        'potential_savings': calculate_savings(recommendations)
    })
}
```

```
### 8.6.4 Security Monitoring

**Security Monitoring Stack:**

| Security Layer | Monitoring Tool | Alert Criteria | Response Time |
|---|---|---|---|
| Network Security | AWS GuardDuty | Suspicious network activity | <5 mi
| Application Security | AWS WAF | SQL injection, XSS attempts | <2 minu
| Access Control | CloudTrail | Unauthorized API calls | <1 minute |
| Data Protection | Config Rules | Encryption compliance | <15 minutes |

**Security Event Response:**

<div class="mermaid-wrapper" id="mermaid-diagram-vna7ytq23">
          <div class="mermaid">
graph TB
    A[Security Event Detected] --&gt; B[Automated Analysis]
    B --&gt; C{Threat Level}
    C --&gt;|Low| D[Log and Monitor]
    C --&gt;|Medium| E[Alert Security Team]
    C --&gt;|High| F[Immediate Response]
    C --&gt;|Critical| G[Emergency Lockdown]

    E --&gt; H[Investigation]
    F --&gt; I[Containment]
    G --&gt; J[System Isolation]
```

```
    H --&gt; K[Remediation]
    I --&gt; K
    J --&gt; L[Incident Response]
</div>
        </div>

### 8.6.5 Compliance Auditing

**Compliance Monitoring Framework:**

| Regulation | Monitoring Scope | Audit Frequency | Retention Requirement
|---|---|---|---|
| SOX | Financial data access and changes | Continuous | 7 years |
| GDPR | Personal data processing | Continuous | As per data retention po
| MiFID II | Trading data and communications | Continuous | 5 years |
| PCI DSS | Payment data handling | Quarterly | 1 year |

**Automated Compliance Reporting:**
```

python

Compliance Monitoring Service

class ComplianceMonitor:
def **init**(self):
self.config = boto3.client('config')
self.cloudtrail = boto3.client('cloudtrail')

```
def check_encryption_compliance(self):
    """Check if all resources are properly encrypted"""
    rules = [
        'encrypted-volumes',
        's3-bucket-ssl-requests-only',
        'rds-storage-encrypted'
    ]
```

```
    compliance_results = {}
    for rule in rules:
        result = self.config.get_compliance_details_by_config_rule(
            ConfigRuleName=rule
        )
        compliance_results[rule] = result

    return compliance_results

def generate_audit_report(self, start_date, end_date):
    """Generate comprehensive audit report"""
    events = self.cloudtrail.lookup_events(
        StartTime=start_date,
        EndTime=end_date,
        LookupAttributes=[
            {
                'AttributeKey': 'EventName',
                'AttributeValue': 'AssumeRole'
            }
        ]
    )

    return {
        'period': f"{start_date} to {end_date}",
        'access_events': len(events['Events']),
        'compliance_status': self.check_encryption_compliance()
    }
```

```
## 8.7 INFRASTRUCTURE DIAGRAMS

### 8.7.1 Infrastructure Architecture Diagram

<div class="mermaid-wrapper" id="mermaid-diagram-cuyi8obpo">
          <div class="mermaid">
graph TB
    subgraph &amp;quot;Internet&amp;quot;
        A[Users] --&amp;gt; B[Route 53 DNS]
        C[Mobile Apps] --&amp;gt; D[Expo Application Services]
    end

    subgraph &amp;quot;AWS Cloud - us-east-1&amp;quot;
```

```
        subgraph &quot;Public Subnets&quot;
            E[Internet Gateway] --&gt; F[Application Load Balancer]
            F --&gt; G[WAF]
        end

        subgraph &quot;Private Subnets - AZ A&quot;
            H[ECS Fargate Tasks] --&gt; I[Flask API Services]
            J[ML Processing Services] --&gt; K[Background Workers]
        end

        subgraph &quot;Private Subnets - AZ B&quot;
            L[ECS Fargate Tasks] --&gt; M[Flask API Services]
            N[ML Processing Services] --&gt; O[Background Workers]
        end

        subgraph &quot;Database Subnets&quot;
            P[RDS PostgreSQL Multi-AZ] --&gt; Q[ElastiCache Redis Cluste
            R[S3 Buckets] --&gt; S[Secrets Manager]
        end

        subgraph &quot;Monitoring &amp; Security&quot;
            T[CloudWatch] --&gt; U[CloudTrail]
            V[GuardDuty] --&gt; W[Config]
        end
    end

    B --&gt; F
    D --&gt; X[EAS Build/Submit/Update]
    G --&gt; H
    G --&gt; L
    I --&gt; P
    M --&gt; P
    J --&gt; Q
    N --&gt; Q

    H --&gt; T
    L --&gt; T
    P --&gt; U
    Q --&gt; U
</div>
        </div>

### 8.7.2 Deployment Workflow Diagram
```

```
<div class="mermaid-wrapper" id="mermaid-diagram-5c99cos1r">
          <div class="mermaid">
graph TB
    subgraph &amp;quot;Development&amp;quot;
        A[Developer] --&amp;gt; B[Git Push]
        B --&amp;gt; C[GitHub Repository]
    end

    subgraph &amp;quot;CI/CD Pipeline&amp;quot;
        C --&amp;gt; D[GitHub Actions]
        D --&amp;gt; E[Run Tests]
        E --&amp;gt; F[Build Docker Images]
        F --&amp;gt; G[Security Scan]
        G --&amp;gt; H[Push to ECR]
    end

    subgraph &amp;quot;Mobile Deployment&amp;quot;
        I[Expo CLI] --&amp;gt; J[EAS Build]
        J --&amp;gt; K[App Store/Play Store]
        L[EAS Update] --&amp;gt; M[OTA Updates]
    end

    subgraph &amp;quot;Backend Deployment&amp;quot;
        H --&amp;gt; N[Terraform Apply]
        N --&amp;gt; O[ECS Service Update]
        O --&amp;gt; P[Blue-Green Deployment]
        P --&amp;gt; Q[Health Checks]
        Q --&amp;gt; R{Deployment Success?}
        R --&amp;gt;|Yes| S[Traffic Switch]
        R --&amp;gt;|No| T[Rollback]
    end

    subgraph &amp;quot;Monitoring&amp;quot;
        S --&amp;gt; U[CloudWatch Monitoring]
        T --&amp;gt; V[Alert Notifications]
        U --&amp;gt; W[Performance Metrics]
        V --&amp;gt; X[Incident Response]
    end

    C --&amp;gt; I
    S --&amp;gt; Y[Production Environment]
    K --&amp;gt; Z[Mobile Users]
```

```
</div>
        </div>

### 8.7.3 Environment Promotion Flow

<div class="mermaid-wrapper" id="mermaid-diagram-8h94neefj">
          <div class="mermaid">
graph LR
    subgraph &quot;Development Environment&quot;
        A[Local Development] --&gt; B[Feature Branch]
        B --&gt; C[Unit Tests]
        C --&gt; D[Dev Deployment]
    end

    subgraph &quot;Staging Environment&quot;
        E[Pull Request] --&gt; F[Integration Tests]
        F --&gt; G[Performance Tests]
        G --&gt; H[Security Tests]
        H --&gt; I[Staging Deployment]
    end

    subgraph &quot;Production Environment&quot;
        J[Release Approval] --&gt; K[Production Deployment]
        K --&gt; L[Blue-Green Switch]
        L --&gt; M[Health Validation]
        M --&gt; N[Monitoring Activation]
    end

    D --&gt; E
    I --&gt; J
    N --&gt; O[Live System]

    P[Rollback Trigger] --&gt; Q[Automatic Rollback]
    Q --&gt; R[Previous Version]
    M --&gt; P
</div>
        </div>

### 8.7.4 Network Architecture

<div class="mermaid-wrapper" id="mermaid-diagram-p7uxz6ybr">
          <div class="mermaid">
graph TB
```

```
    subgraph &amp;quot;VPC: 10.0.0.0/16&amp;quot;
        subgraph &amp;quot;Public Subnets&amp;quot;
            A[Public Subnet A: 10.0.1.0/24] --&amp;gt; B[NAT Gateway A]
            C[Public Subnet B: 10.0.2.0/24] --&amp;gt; D[NAT Gateway B]
            E[Internet Gateway] --&amp;gt; F[Application Load Balancer]
        end

        subgraph &amp;quot;Private Subnets&amp;quot;
            G[Private Subnet A: 10.0.10.0/24] --&amp;gt; H[ECS Tasks A]
            I[Private Subnet B: 10.0.11.0/24] --&amp;gt; J[ECS Tasks B]
        end

        subgraph &amp;quot;Database Subnets&amp;quot;
            K[DB Subnet A: 10.0.20.0/24] --&amp;gt; L[RDS Primary]
            M[DB Subnet B: 10.0.21.0/24] --&amp;gt; N[RDS Standby]
            O[Cache Subnet A: 10.0.30.0/24] --&amp;gt; P[ElastiCache A]
            Q[Cache Subnet B: 10.0.31.0/24] --&amp;gt; R[ElastiCache B]
        end
    end

    subgraph &amp;quot;Security Groups&amp;quot;
        S[ALB Security Group] --&amp;gt; T[Port 80, 443 from Internet]
        U[ECS Security Group] --&amp;gt; V[Port 5000 from ALB]
        W[RDS Security Group] --&amp;gt; X[Port 5432 from ECS]
        Y[Cache Security Group] --&amp;gt; Z[Port 6379 from ECS]
    end

    F --&amp;gt; H
    F --&amp;gt; J
    H --&amp;gt; L
    J --&amp;gt; L
    H --&amp;gt; P
    J --&amp;gt; R

    B --&amp;gt; H
    D --&amp;gt; J
</div>
        </div>

## 8.8 INFRASTRUCTURE COST ESTIMATES

### 8.8.1 Monthly Cost Breakdown
```

```
**AWS Infrastructure Costs:**

| Service | Development | Staging | Production | Annual Total |
|---|---|---|---|---|
| ECS Fargate | $30 | $120 | $600 | $9,000 |
| RDS PostgreSQL | $25 | $100 | $400 | $6,300 |
| ElastiCache Redis | $15 | $50 | $200 | $3,180 |
| Application Load Balancer | $20 | $25 | $30 | $900 |
| S3 Storage | $5 | $15 | $50 | $840 |
| CloudWatch | $10 | $25 | $75 | $1,320 |
| Data Transfer | $5 | $15 | $50 | $840 |
| **Environment Total** | **$110** | **$350** | **$1,405** | **$22,380**

**Expo Application Services Costs:**

| Service | Tier | Monthly Cost | Annual Cost |
|---|---|---|---|
| EAS Build | Production Plan | $99 | $1,188 |
| EAS Update | Usage-based | $50 | $600 |
| EAS Submit | Included | $0 | $0 |
| **EAS Total** | | **$149** | **$1,788** |

**Total Infrastructure Cost: $24,168 annually**

### 8.8.2 Cost Optimization Opportunities

**Reserved Instance Savings:**

| Service | On-Demand Cost | Reserved Cost (1-year) | Annual Savings |
|---|---|---|---|
| RDS Production | $4,800 | $3,360 | $1,440 |
| ElastiCache Production | $2,400 | $1,680 | $720 |
| **Total Savings** | | | **$2,160** |

**Scaling Optimizations:**

- **Auto-scaling**: Reduce costs during low-traffic periods by 30-40%
- **Spot Instances**: Use for non-critical workloads (development/testing
- **Storage Optimization**: Implement S3 lifecycle policies for 20% stora

### 8.8.3 Resource Sizing Guidelines

**Compute Resource Recommendations:**
```

```
| Workload Type | CPU | Memory | Storage | Scaling Strategy |
|---|---|---|---|---|
| API Services | 1-2 vCPU | 2-4 GB | 20 GB | Horizontal auto-scaling |
| ML Processing | 2-4 vCPU + GPU | 4-8 GB | 50 GB | Vertical scaling on
| Background Jobs | 0.5-1 vCPU | 1-2 GB | 10 GB | Queue-based scaling |
| Database | 2-4 vCPU | 8-16 GB | 100-500 GB | Read replica scaling |

The infrastructure design provides a robust, scalable, and cost-effective

# APPENDICES

##### A. ADDITIONAL TECHNICAL INFORMATION

## A.1 TradingView API Integration Details

**Unofficial API Wrapper Implementation:**
Since TradingView doesn't provide official API access for data retrieval

| Library | Version | Purpose | Limitations |
|---|---|---|---|
| tradingview-ta | 3.3.0+ | Technical analysis data retrieval | Unofficia
| tradingview-screener | 3.0.0+ | Market screening capabilities | Limited
| python-tradingview-ta | Latest | Alternative TA data source | Community

**TA_Handler Configuration:**
```

python

Technical Analysis Handler Setup

from tradingview_ta import TA_Handler, Interval

handler = TA_Handler(
symbol="EURUSD",
screener="forex",

exchange="FX_IDC",
interval=Interval.INTERVAL_1_HOUR
)

analysis = handler.get_analysis()

Returns: {"RECOMMENDATION": "BUY", "BUY": 8, "SELL": 3, "NEUTRAL": 15}

```
## A.2 React Native 0.81 New Architecture Features

**Fabric Renderer Implementation:**
React Native 0.81 includes the new Fabric renderer which provides improve

| Feature | Legacy Architecture | New Architecture | Performance Impact
|---|---|---|---|
| UI Updates | Asynchronous bridge | Synchronous JSI | 2-3x faster rende
| Memory Usage | Higher overhead | Optimized allocation | 20-30% reductio
| Startup Time | Slower initialization | Faster bootstrap | 40% improveme

**TurboModules Integration:**
The new TurboModules system provides better native module performance and

## A.3 Expo SDK 54 Specific Features

**Edge-to-Edge Android Support:**
With React Native 0.81 targeting Android 16, edge-to-edge display is enab

**File System API Enhancements:**
```

javascript
// expo-file-system object-oriented API

import { FileSystem } from 'expo-file-system';

const directory = FileSystem.documentDirectory;
const file = directory.getFile('trading-data.json');

// SAF URIs support on Android
const safUri = await
FileSystem.StorageAccessFramework.requestDirectoryPermissionsAsync();

```
## A.4 Machine Learning Model Specifications

**Supported ML Frameworks and Versions:**

| Framework | Version | Use Case | Model Types |
|---|---|---|---|
| scikit-learn | 1.3.0+ | Traditional ML algorithms | Classification, Reg
| TensorFlow | 2.13.0+ | Deep learning models | Neural networks, LSTM |
| Keras | 2.13.0+ | High-level neural networks | Sequential, Functional
| LightGBM | 4.0.0+ | Gradient boosting | Tree-based models |

**Model Training Data Requirements:**
- **Historical Data**: Minimum 500 days of OHLC data per currency pair
- **Technical Indicators**: 20+ standard indicators (RSI, MACD, Bollinge
- **Feature Engineering**: Price ratios, volatility measures, momentum in
- **Backtesting Dataset**: Minimum 25,000+ trades for optimization and va

## A.5 Database Schema Extensions

**MongoDB Time Series Collection Configuration:**
```

javascript
// Advanced time series collection setup
db.createCollection("market_data_advanced", {
timeseries: {
timeField: "timestamp",
metaField: "metadata",
granularity: "seconds",
bucketMaxSpanSeconds: 3600,

bucketRoundingSeconds: 3600
},
clusteredIndex: {
key: { _id: 1 },
unique: true
},
expireAfterSeconds: 31536000 // 1 year retention
});

```
**GridFS Implementation for Large Files:**
```

python

GridFS for ML model storage

from pymongo import MongoClient
import gridfs

client = MongoClient('mongodb://localhost:27017/')
db = client.forex_trading
fs = gridfs.GridFS(db)

Store ML model

with open('trading_model.pkl', 'rb') as f:
model_id = fs.put(f, filename='trading_model_v1.2.3.pkl')

```
## A.6 Security Implementation Details

**JWT Token Structure:**
```

json
{

"header": {
"alg": "RS256",
"typ": "JWT",
"kid": "key-id-2025"
},
"payload": {
"sub": "user-uuid",
"iss": "forex-trading-app",
"aud": "mobile-app",
"exp": 1735689600,
"iat": 1735686000,
"roles": ["premium_user"],
"permissions": ["market_data", "trading_strategies"]
}
}

```
**Encryption Key Management:**
```

python

AWS KMS integration for key management

import boto3
from cryptography.fernet import Fernet

class KeyManager:
def **init**(self):
self.kms = boto3.client('kms')
self.key_id = 'arn:aws:kms:us-east-1:account:key/key-id'

```python
def encrypt_sensitive_data(self, data):
    # Generate data key
    response = self.kms.generate_data_key(
        KeyId=self.key_id,
        KeySpec='AES_256'
    )

    # Encrypt data with data key
    fernet = Fernet(response['Plaintext'])
    encrypted_data = fernet.encrypt(data.encode())

    return {
        'encrypted_data': encrypted_data,
        'encrypted_key': response['CiphertextBlob']
    }
```

```
## A.7 Performance Optimization Techniques

**React Native Performance Optimizations:**
```

javascript
// Optimized FlatList configuration for large datasets
item.symbol}
getItemLayout={(data, index) => ({
length: ITEM_HEIGHT,
offset: ITEM_HEIGHT * index,
index,
})}
removeClippedSubviews={true}
maxToRenderPerBatch={10}
windowSize={10}
initialNumToRender={20}
updateCellsBatchingPeriod={50}
/>

```
**Flask Application Optimizations:**
```

python

Gunicorn configuration for production

bind = "0.0.0.0:5000"
workers = 4
worker_class = "gevent"
worker_connections = 1000
max_requests = 1000
max_requests_jitter = 100
timeout = 30
keepalive = 5
preload_app = True

```
## A.8 Compliance and Regulatory Considerations

**MiFID II Technical Standards:**
- **RTS 6**: Post-trade transparency for equity instruments
- **RTS 22**: Regulatory technical standards on best execution
- **RTS 27**: Regulatory technical standards on investment firms providi

**Data Retention Requirements:**

| Data Type | Regulation | Retention Period | Storage Requirements |
|---|---|---|---|
| Trading Records | MiFID II | 5 years | Immutable, encrypted |
| Client Communications | MiFID II | 7 years | Searchable, archived |
| Personal Data | GDPR | User consent based | Right to erasure |
| Audit Logs | SOX | 7 years | Tamper-proof, accessible |

## A.9 Testing Framework Extensions
```

```
**Maestro E2E Testing Configuration:**
```

yaml

Advanced Maestro flow for trading strategy testing

appId: com.forextrading.app

- launchApp
- assertVisible: "Dashboard"
- tapOn:
  id: "market_analysis_tab"
- tapOn:
  id: "currency_pair_selector"
- tapOn: "EURUSD"
- assertVisible:
  text: "Loading market dataâ€¦"
  timeout: 5000
- assertVisible:
  id: "candlestick_chart"
  timeout: 10000
- tapOn:
  id: "generate_strategy_button"
- assertVisible:
  text: "Strategy Generated"
  timeout: 15000
- assertVisible:
  id: "confidence_score"

- assert:
  visible: "Confidence: [0-9]+%"
  timeout: 2000

```
**Performance Testing Thresholds:**
```

python

Locust performance testing configuration

from locust import HttpUser, task, between

class ForexTradingUser(HttpUser):
wait_time = between(1, 3)

```
@task(3)
def get_market_data(self):
    self.client.get("/api/v1/market-data/EURUSD")

@task(2)
def generate_strategy(self):
    self.client.post("/api/v1/trading-strategy", json={
        "symbol": "EURUSD",
        "timeframe": "1h",
        "risk_tolerance": "medium"
    })

@task(1)
def get_portfolio(self):
    self.client.get("/api/v1/portfolio")
```

```

## B. GLOSSARY
```

API Gateway: A service that acts as an entry point for client requests to backend services, handling routing, authentication, rate limiting, and request/response transformation.

Backtesting: The process of testing a trading strategy using historical market data to evaluate its potential performance before applying it to live trading.

Candlestick Chart: A type of financial chart used to describe price movements of a security, derivative, or currency, showing open, high, low, and close prices for a specific period.

Circuit Breaker Pattern: A design pattern used in software development to detect failures and encapsulate the logic of preventing a failure from constantly recurring during maintenance, temporary external system failure, or unexpected system difficulties.

Containerization: A lightweight form of virtualization that packages applications and their dependencies into containers that can run consistently across different computing environments.

CORS (Cross-Origin Resource Sharing): A mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.

Edge-to-Edge Display: A mobile app design approach where content extends to the very edges of the screen, utilizing the full display area including areas traditionally reserved for system UI.

Exponential Backoff: A strategy for handling retries in distributed systems where the delay between retry attempts increases exponentially to avoid overwhelming a failing service.

Fabric Renderer: React Native's new rendering system that provides better performance and more direct integration with native platform capabilities.

Forex (Foreign Exchange): The global marketplace for trading national currencies against one another, operating 24 hours a day, five days a week.

GridFS: A specification for storing and retrieving files that exceed the BSON-document size limit of 16 MB in MongoDB.

Haptic Feedback: Technology that uses touch sensations, typically vibrations, to provide tactile feedback to users during interactions with mobile devices.

Infrastructure as Code (IaC): The practice of managing and provisioning computing infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools.

JSI (JavaScript Interface): React Native's new JavaScript interface that enables direct communication between JavaScript and native code without the traditional bridge.

Load Balancer: A device or software application that distributes network or application traffic across multiple servers to ensure no single server becomes overwhelmed.

Microservices: An architectural approach where applications are built as a collection of loosely coupled, independently deployable services.

Over-the-Air (OTA) Updates: A method of distributing software updates to mobile devices wirelessly, without requiring users to manually download and install updates.

OHLC Data: Open, High, Low, Close data representing the four key price points for a financial instrument during a specific time period.

Position Sizing: The process of determining how much capital to allocate to a particular trade based on risk management principles and account size.

Rate Limiting: A technique used to control the rate at which clients can make requests to an API or service, preventing abuse and ensuring fair usage.

Real-time Data: Information that is delivered immediately after collection, with minimal delay between data generation and availability.

Sharding: A database architecture pattern that involves splitting large databases into smaller, more manageable pieces called shards.

Technical Indicators: Mathematical calculations based on historical price and volume data used to predict future price movements in financial markets.

Time Series Data: Data points indexed in time order, typically representing measurements or observations collected at regular intervals.

TurboModules: React Native's new native module system that provides better performance and type safety compared to the legacy bridge-based system.

WebSocket: A communication protocol that provides full-duplex communication channels over a single TCP connection, enabling real-time data exchange.

C. ACRONYMS

Acronym	Expanded Form
ALB	Application Load Balancer
API	Application Programming Interface
APM	Application Performance Monitoring
ARN	Amazon Resource Name
AWS	Amazon Web Services
BSON	Binary JSON

Acronym	Expanded Form
CCPA	California Consumer Privacy Act
CDN	Content Delivery Network
CI/CD	Continuous Integration/Continuous Deployment
CLI	Command Line Interface
CORS	Cross-Origin Resource Sharing
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DEA	Direct Electronic Access
DNS	Domain Name System
DRY	Don't Repeat Yourself
EAS	Expo Application Services
EC2	Elastic Compute Cloud
ECS	Elastic Container Service
EMA	Exponential Moving Average
FCM	Firebase Cloud Messaging
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
HCL	HashiCorp Configuration Language
HFT	High-Frequency Trading
HMAC	Hash-based Message Authentication Code
HSM	Hardware Security Module
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure

Acronym	Expanded Form
IAM	Identity and Access Management
IaC	Infrastructure as Code
IDE	Integrated Development Environment
IoC	Inversion of Control
IP	Internet Protocol
JSI	JavaScript Interface
JSON	JavaScript Object Notation
JWT	JSON Web Token
KMS	Key Management Service
KPI	Key Performance Indicator
LSTM	Long Short-Term Memory
LRU	Least Recently Used
MA	Moving Average
MACD	Moving Average Convergence Divergence
MFA	Multi-Factor Authentication
MiFID II	Markets in Financial Instruments Directive II
ML	Machine Learning
MTTR	Mean Time to Resolution
MTTD	Mean Time to Detection
mTLS	Mutual Transport Layer Security
NAT	Network Address Translation
OHLC	Open, High, Low, Close
ORM	Object-Relational Mapping
OTA	Over-the-Air
OTLP	OpenTelemetry Protocol

Acronym	Expanded Form
PCI DSS	Payment Card Industry Data Security Standard
PKCE	Proof Key for Code Exchange
PR	Pull Request
RAM	Random Access Memory
RBAC	Role-Based Access Control
RDS	Relational Database Service
REST	Representational State Transfer
RPO	Recovery Point Objective
RPS	Requests Per Second
RSI	Relative Strength Index
RTO	Recovery Time Objective
S3	Simple Storage Service
SAF	Storage Access Framework
SAST	Static Application Security Testing
SDK	Software Development Kit
SIEM	Security Information and Event Management
SLA	Service Level Agreement
SLO	Service Level Objective
SMS	Short Message Service
SNS	Simple Notification Service
SOX	Sarbanes-Oxley Act
SQL	Structured Query Language
SRE	Site Reliability Engineering
SSL	Secure Sockets Layer
SSD	Solid State Drive

Acronym	Expanded Form
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTL	Time To Live
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VPC	Virtual Private Cloud
WAF	Web Application Firewall
WCAG	Web Content Accessibility Guidelines
WSGI	Web Server Gateway Interface
XSS	Cross-Site Scripting
YAML	YAML Ain't Markup Language