Technical Documentation

System Architecture

Design Patterns

1. Strategy Pattern

The system uses the Strategy pattern for trading algorithms:

cpp class Strategy { public: virtual void on_data(const Candle& candle, const Candle* high_tf = nullptr) = 0; virtual bool should_buy() const = 0; virtual bool should_sell() const = 0; virtual void init(TradeEngine* engine) { this->engine = engine; } };

Benefits: - Easy to add new strategies without modifying existing code - Runtime strategy selection and switching - Clean separation of strategy logic from execution

2. Observer Pattern

Market data updates are propagated through the system:

cpp // Data flows: DataLoader -> Strategy -> TradeEngine -> Portfolio void on_data(const Candle& candle) { strategy->on_data(candle); if (strategy->should_buy()) { engine->queue_buy(candle.close, candle.timestamp_str, candle.timestamp, symbol); } engine->update(candle); }

3. Factory Pattern

Strategy creation is handled through factory functions:

cpp std::vector<std::pair<std::string, std::function<std::unique_ptr()>>> strategies = { {“SMA”, { return std::make_unique(); }}, {“EMARSI”, { return std::make_unique(); }} };

Core Components

1. Data Management Layer

DataLoader (`src/data_loader.cpp`) - Handles CSV file parsing with retry mechanisms - Supports multiple data sources and formats - Implements robust error handling for corrupted data - Memory-efficient streaming for large datasets

TimeframeAggregator (`src/timeframe_aggregator.cpp`) - Converts 1-minute data to higher timeframes (5m, 15m, 1h) - Maintains sliding windows for real-time aggregation - Optimized for low-latency live data processing

2. Strategy Execution Layer

TradeEngine (`src/trade_engine.cpp`) - Event-driven order processing with realistic latency simulation - Implements slippage modeling based on market conditions - Maintains position tracking and P&L calculation - Supports both backtesting and live shadow modes

Portfolio Manager (`src/portfolio_manager.cpp`) - Multi-asset position management - Real-time P&L calculation (realized and unrealized) - Integration with risk management system - Performance metrics calculation

3. Risk Management System

RiskManager (`src/risk_manager.cpp`) - Pre-trade risk checks (position size, correlation limits) - Real-time monitoring of drawdown and VaR - Dynamic risk limit adjustment based on market conditions - Alert system for risk threshold breaches

Performance Optimizations

1. Memory Management

cpp // Efficient circular buffers for time series data std::deque candles; if (candles.size() > lookback_period) { candles.pop_front(); // O(1) operation }

2. Data Structures

- **std::deque**: For sliding windows and time series data
- **std::unordered_map**: For fast symbol-based lookups
- **std::vector**: For bulk data operations and analytics

3. Threading Considerations

- Thread-safe counters using std::atomic
- Lock-free data structures for high-frequency updates
- Separate threads for data ingestion and strategy execution

Backtesting Methodology

Event-Driven Simulation

The backtesting engine uses an event-driven architecture that processes market data chronologically:

1. **Data Ingestion**: Historical candles are loaded and sorted by timestamp
2. **Strategy Processing**: Each candle triggers strategy logic evaluation
3. **Order Generation**: Strategies generate buy/sell signals
4. **Order Execution**: Orders are processed with realistic delays and slippage
5. **Portfolio Update**: Positions and P&L are updated

Realistic Market Conditions

Slippage Modeling

cpp double adjusted_price = price * (1.0 + slippage_rate); // Buy orders double adjusted_price = price * (1.0 - slippage_rate); // Sell orders

Latency Simulation

cpp // Orders are delayed by configurable latency (default: 5 seconds) pending_orders.push_back({ DelayedOrder::BUY, timestamp + std::chrono::seconds(latency_seconds), price, symbol });

Transaction Costs

- Configurable slippage rates (default: 2 basis points)
- Market impact modeling for large orders
- Realistic bid-ask spread simulation

Multi-Asset Backtesting

The system supports simultaneous backtesting across multiple assets:

cpp // Each asset maintains its own data stream and strategy instances for (auto& [symbol, candles] : all_candles) { for (auto& [name, factory] : selected_strategies)

{ // Run strategy on specific asset auto strategy = factory(); auto engine = std::make_unique(/.../); // Process all candles for this asset-strategy combination } }

Performance Benchmarking

Backtesting Speed

- **Target**: Process historical data faster than real-time
- **Achieved**: 100x-1000x real-time depending on strategy complexity
- **Measurement**: Candles processed per second

Live Data Processing

- **Target**: Strategy execution within 1ms of market data update
- **Latency Tracking**: Microsecond-precision timing measurements
- **Throughput**: 1000+ candles per second processing capability

Memory Usage

- **Efficient Data Structures**: Circular buffers for sliding windows
- **Memory Monitoring**: Real-time tracking of memory consumption
- **Optimization**: Automatic cleanup of old data beyond lookback periods

Error Handling and Reliability

Data Feed Management

cpp // Retry mechanism for data loading int max_retries = 3; for (int retry = 0; retry < max_retries; ++retry) { std::ifstream file(filename); if (!file.is_open()) { if (retry == max_retries - 1) { std::cerr << "Failed to open file after" << max_retries << " attempts"; } else { std::this_thread::sleep_for(std::chrono::milliseconds(100)); continue; } } // Process file… }

Strategy Execution Recovery

- Graceful handling of strategy exceptions
- Automatic strategy restart on failures
- Comprehensive logging for debugging

System Monitoring

- Real-time health checks and metrics collection
- Automatic alerting on system anomalies
- Performance degradation detection and reporting