**Knowledge required for implementation of project**

A bank of financial, mathematical and computational information which could potentially relate to simulation of stock marketing trading.

**Mathematical foundations and performance metrics**

Building a trading platform requires deep understanding of financial mathematics, particularly performance measurement and risk quantification.

**Sharpe ratio** forms the cornerstone of risk-adjusted performance evaluation, calculated as (Rp - Rf) / σp where portfolio return minus risk-free rate is divided by standard deviation. This metric balances returns against volatility, but the **Sortino ratio** offers superior insight by focusing only on downside deviation: (Rp - Rf) / σd, where σd represents standard deviation of negative returns only.

**Maximum drawdown** reveals the worst-case portfolio decline from peak to trough, essential for understanding capital preservation:

MDD = (Trough Value - Peak Value) / Peak Value × 100%.

From OHLCV data, this requires tracking cumulative returns and running maximum values continuously.

Risk metrics center on **Value at Risk (VaR)**, quantifying potential losses at specific confidence levels. The parametric method uses

VaR = -[μ + (Z-score × σ)] × Portfolio Value,

while historical simulation sorts returns to find percentile losses. **Position sizing** via Kelly Criterion optimizes capital allocation:

f\* = W - (1-W)/R where W equals win rate and R represents average win/loss ratio, though conservative implementations use 25-50% of calculated Kelly percentage.

**Trading algorithm mathematical foundations**

Moving averages provide the mathematical foundation for trend analysis.

**Simple Moving Average**

SMA\_n = (P\_1 + P\_2 + ... + P\_n) / n offers equal weighting, while **Exponential Moving Average** applies exponentially decaying weights:

EMA\_t = α × P\_t + (1-α) × EMA\_(t-1) where smoothing factor α = 2/(n+1).

EMA mathematical expansion reveals EMA\_t = Σ[i=0 to ∞] α(1-α)^i × P\_(t-i), showing how recent prices receive greater influence.

**Mean reversion strategies** leverage Bollinger Bands mathematical structure: middle band equals 20-period SMA, while upper/lower bands add/subtract k × σ (typically k=2). The **%B indicator** normalizes price position: %B = (Price - Lower Band) / (Upper Band - Lower Band), providing overbought/oversold signals.

**RSI calculation** involves multi-step process:

separate gains/losses, calculate exponential averages

AG = EMA\_14(Gains),

AL = EMA\_14(Losses), then compute

RSI = 100 - (100 / (1 + AG/AL)).

Values above 70 indicate overbought conditions, below 30 suggest oversold.

**MACD momentum indicator** combines three elements: MACD line EMA\_12(Price) - EMA\_26(Price), signal line EMA\_9(MACD), and histogram MACD - Signal. Crossovers generate trading signals, while divergences reveal momentum shifts.

**Statistical arbitrage** employs sophisticated mathematical concepts including **cointegration testing** via Engle-Granger methodology and **Z-score calculations** for pairs trading:

Z\_t = (Spread\_t - μ\_spread) / σ\_spread.

Kalman filtering provides dynamic hedge ratio estimation through recursive state-space modeling, particularly valuable for non-stationary relationships.

**Risk management and market mechanics integration**

Transaction cost modeling requires comprehensive understanding of market microstructure.

**Total transaction costs** = Fixed\_Cost + (Variable\_Rate × Volume) + Spread\_Cost + Slippage\_Cost.

Bid-ask spreads follow (Ask - Bid) / Mid\_Price × 100% for relative measurement, while **slippage estimation** uses power law relationships: Slippage = α × (Order\_Size / Average\_Volume)^β.

**Execution algorithms** optimize trade implementation through mathematical models. **TWAP** divides orders equally across time intervals, while **VWAP** weights execution by historical volume patterns.

**Implementation Shortfall** balances market impact against timing risk through dynamic optimization.

**Portfolio risk management** integrates multiple constraint systems: concentration limits (5-10% maximum position size), leverage constraints (gross/net exposure limits), and correlation analysis through **diversification ratio**:

DR = σ\_portfolio / Σ(w\_i × σ\_i).

**Kelly Criterion position sizing** connects win rates and profit ratios to optimal capital allocation, though practical implementations require conservative scaling factors.

**Backtesting methodologies** demand rigorous bias prevention. **Walk-forward analysis** repeatedly optimizes parameters on training windows then tests on subsequent periods. **Point-in-time data** ensures historical accuracy, while **purged cross-validation** prevents data leakage in financial time series.

**Technology implementation architecture**

The C++ algorithmic core leverages **QuantLib** for comprehensive quantitative finance calculations, **Boost** libraries for threading and networking, and **Eigen** for high-performance linear algebra. **TA-Lib** provides 200+ technical indicators with optimized implementations. Low-latency requirements demand **lock-free data structures**, **static memory allocation**, and **SIMD optimizations**.

**Python integration** through FastAPI bridges C++ computations with web interfaces. **Pandas/NumPy** handle time-series manipulation, while **QuantLib-Python** provides derivative pricing capabilities. **Backtesting frameworks** like Backtesting.py and Zipline offer production-grade strategy evaluation with realistic market simulation.

**TimescaleDB optimization** transforms PostgreSQL into high-performance time-series database. **Hypercore columnar storage** achieves 90% compression ratios through delta encoding and dictionary compression. **Continuous aggregates** provide real-time OHLCV bar creation: time\_bucket('1 minute', timestamp) with automatic refresh policies.

**Data source integration** spans multiple providers: Polygon.io for sub-20ms real-time feeds, Alpaca API for commission-free paper trading, and WebSocket streaming for live market data. **Redis caching** implements multiple patterns: cache-aside for historical lookups, write-through for real-time updates, with sophisticated eviction policies.

**React/TypeScript frontend** utilizes specialized financial charting libraries. **React Financial Charts** provides candlestick visualization with technical indicator overlays, while **WebSocket integration** streams real-time updates with efficient data decimation for performance.

**System integration and simulation realism**

**Data flow architecture** follows clear pipeline:

Market Data APIs → FastAPI Bridge → TimescaleDB/Redis → WebSocket → React Frontend, with C++ Core Engine handling algorithmic processing.

**Event-driven architecture** ensures responsive real-time processing through **Complex Event Processing** and **CQRS patterns**.

**Risk management integration** operates through multiple checkpoints:

pre-trade risk validation, real-time position monitoring, and post-trade performance attribution.

**Position sizing algorithms** connect mathematical risk models with actual trade execution, while **correlation analysis** prevents excessive concentration.

**Performance evaluation systems** calculate metrics continuously: TWAP/VWAP execution quality, real-time Sharpe ratio monitoring, and comprehensive **Transaction Cost Analysis**. **Backtesting infrastructure** maintains point-in-time datasets with corporate action adjustments and realistic cost assumptions.

**Simulation realism** demands sophisticated **market microstructure modeling**: realistic order book dynamics, **market impact functions** following square-root price impact laws, and **latency simulation** including network delays and processing queues. **Agent-based modeling** creates diverse market participant behavior, while **volatility clustering** models realistic price dynamics.

**Implementation best practices and educational guidance**

**Development workflow** prioritizes mathematical accuracy verification before performance optimization. Start with **pure mathematical implementations** using exact formulas, then optimize for speed while maintaining numerical precision. **Unit testing** should verify calculations against known benchmark values across multiple market conditions.

**Architecture patterns** emphasize **separation of concerns**:

C++ handles computational intensity, Python manages data processing and API integration, while TypeScript provides interactive user interfaces.

**Microservices deployment** enables independent scaling of different system components.

**Risk management philosophy** should embed controls at every system layer: algorithm-level position sizing, pre-trade risk checks, real-time monitoring, and post-trade analysis. Never implement trading systems without comprehensive risk oversight and emergency stop mechanisms.

**Performance optimization** follows clear priorities: correctness first, then speed. Profile actual bottlenecks rather than premature optimization.

In financial systems, **millisecond improvements** in execution can significantly impact profitability, but accuracy errors cause catastrophic losses.

**Educational progression** should begin with simple moving average strategies, advance through mean reversion and momentum approaches, then tackle sophisticated statistical arbitrage. Each algorithm requires thorough mathematical understanding before implementation, extensive backtesting across market conditions, and paper trading validation before live deployment.

**Regulatory considerations** demand comprehensive audit trails, position reporting capabilities, and compliance with applicable financial regulations. Modern trading systems must balance performance optimization with regulatory requirements and operational risk management.