perplexity

MVP Implementation Plan: Quant Trading Assistant

Based on your comprehensive design document, here's a structured roadmap to build the MVP with detailed explanations to help you learn and become proficient in quantitative finance software development.[1]

Core MVP Scope

The MVP focuses on **sparse mean-reverting portfolio selection** with backtesting capabilities, risk management, and a simple dashboard. Phase 2 features (ML predictors, game theory, explainability) will be added after the core system is validated.[2] [1]

Technology Stack Recommendations

Backend Framework: FastAPI - Modern, high-performance Python framework ideal for financial APIs with automatic documentation and async support. FastAPI is production-ready and scales well for trading applications.[3] [4]

Database Layer:

- PostgreSQL for relational data (users, trades, portfolios) with JSONB support for flexible schema[1]
- TimescaleDB extension for efficient time-series OHLCV data storage[1]

Data Processing:

- Pandas and NumPy for matrix operations and VAR(1) computations[1]
- SciPy for eigenvalue decomposition and optimization[5] [1]

Frontend: React with Chart.js or Plotly for interactive performance visualization[2]

Implementation Phases

Sprint 0: Foundation (Week 1-2)

Objective: Establish project infrastructure and development environment.[1]

Tasks:

1. **Repository Setup**: Create a structured project with clear separation of concerns :[4]

```
quant-trading-assistant/
├── app/
│   ├── api/              # FastAPI routes
```

```
│   ├── models/           # Database models
│   ├── services/         # Business logic
│   ├── core/             # Configuration
│   └── utils/            # Helpers
├── tests/
├── data/                 # Market data cache
└── docker-compose.yml
```

2. **Database Schema Implementation**: Start with essential tables :[1]
   - `users`: Store risk profiles (risk_tolerance, capital, max_assets, drawdown_limit)
   - `market_data_meta`: Symbol metadata
   - `ohlcv_daily`: Time-series price data
   - `portfolios`: Generated portfolio configurations
   - `trades`: Execution records with slippage tracking

Learning Focus: Understanding database normalization, proper indexing for time-series queries, and connection pooling.[4] [1]

3. **CI/CD Pipeline**: Set up GitHub Actions for automated testing and Docker builds.[2]

Sprint 1: Core Data Pipeline (Week 3-5)

Objective: Build the foundational data ingestion and feature engineering system.[1]

Component 1: DataFetcher

Purpose: Retrieve historical market data from sources like Yahoo Finance, Alpha Vantage, or Polygon.io.[1]

Implementation Strategy:

```
class DataFetcher:
    def __init__(self, sources: List[str]):
        self.sources = sources
        self.cache = {}

    async def fetch(self, symbols: List[str],
                    start: datetime, end: datetime) -> pd.DataFrame:
        # Implement with rate limiting and retry logic
        # Cache results to minimize API calls
        # Return OHLCV DataFrame indexed by date
```

Key Learning:

- Handling API rate limits gracefully with exponential backoff[3]
- Data validation (checking for missing values, outliers, stock splits)
- Efficient caching strategies to reduce latency[1]

Component 2: FeatureEngineer

Purpose: Transform raw price data into standardized features for portfolio construction.[1]

Mathematical Foundation :[5] [1]

1. Compute log returns: $ r_t = \log(P_t / P_{t-1}) $
2. Z-score normalization: $ z_t = (r_t - \mu) / \sigma $
3. Build VAR(1) coefficient matrix from historical returns

Implementation:

```
class FeatureEngineer:
    def compute_features(self, df: pd.DataFrame) -> pd.DataFrame:
        # Calculate returns, volatility, correlation
        returns = np.log(df / df.shift(1))

        # Standardize features
        standardized = (returns - returns.mean()) / returns.std()

        return standardized

    def build_VAR1_matrix(self, returns: pd.DataFrame) -> np.ndarray:
        # Estimate VAR(1): R_t = R_{t-1} * A + Z_t
        # Use OLS or Maximum Likelihood Estimation
        X = returns.values[:-1]  # Lagged returns
        Y = returns.values[1:]   # Current returns

        A = np.linalg.lstsq(X, Y, rcond=None)[^1_0]
        return A
```

Learning Focus: Time-series analysis, stationarity testing, and understanding autocorrelation in financial data.[5] [1]

Sprint 2: Portfolio Constructor (Week 6-8)

Objective: Implement the **sparse mean-reverting portfolio selection algorithm**.[5] [1]

Theoretical Background :[6] [5] [1]

Mean reversion assumes asset prices oscillate around a long-term equilibrium. The algorithm identifies **cointegrated asset combinations** (portfolios whose linear combinations are stationary).[5]

Sparsity Constraint: Select only k assets (e.g., 10-20 from a universe of 200) to reduce transaction costs and improve interpretability.[7] [5] [1]

Algorithm Implementation :[5] [1]

Step 1: Covariance Selection (Graphical LASSO)

```
from sklearn.covariance import GraphicalLassoCV

def estimate_sparse_precision(returns: pd.DataFrame,
```

```
                               alpha: float = 0.1) -> np.ndarray:
    """
    Estimate sparse inverse covariance matrix.
    Zero entries indicate conditional independence.
    """
    model = GraphicalLassoCV(alphas=[alpha])
    model.fit(returns)
    return model.precision_  # Sparse inverse covariance
```

Step 2: Structured VAR(1) Estimation with L1 Penalty

```
from sklearn.linear_model import Lasso

def sparse_VAR_coefficients(returns: pd.DataFrame,
                            lambda_penalty: float = 0.05) -> np.ndarray:
    """
    Fit VAR(1) model with LASSO regularization for sparsity.
    """
    X = returns.values[:-1]
    Y = returns.values[1:]

    # Fit separate LASSO for each asset
    A_sparse = np.zeros((X.shape[^1_1], Y.shape[^1_1]))

    for i in range(Y.shape[^1_1]):
        lasso = Lasso(alpha=lambda_penalty, fit_intercept=False)
        lasso.fit(X, Y[:, i])
        A_sparse[:, i] = lasso.coef_

    return A_sparse
```

Step 3: Eigenvector Selection for Mean Reversion

```
def select_mean_reverting_portfolio(A: np.ndarray,
                                    Sigma: np.ndarray,
                                    k: int = 15) -> dict:
    """
    Solve generalized eigenproblem to find stationary directions.
    Select k assets with largest contributions.
    """
    # Generalized eigenvalue problem: A*x = λ*B*x
    # Where B relates to covariance structure
    eigenvalues, eigenvectors = scipy.linalg.eig(A, Sigma)

    # Select eigenvector with smallest eigenvalue (most mean-reverting)
    idx = np.argmin(np.abs(eigenvalues))
    weights = eigenvectors[:, idx].real

    # Enforce sparsity: keep top-k absolute weights
    top_k_indices = np.argsort(np.abs(weights))[-k:]
    sparse_weights = np.zeros_like(weights)
    sparse_weights[top_k_indices] = weights[top_k_indices]

    # Normalize to sum to 1
```

```
    sparse_weights /= sparse_weights.sum()

    return {
        'weights': sparse_weights,
        'assets': top_k_indices,
        'eigenvalue': eigenvalues[idx]
    }
```

Learning Focus:

- Understanding cointegration vs correlation[5]
- Eigenvalue decomposition for dimension reduction[1]
- L1 regularization for sparsity (LASSO)[6] [5]
- Trade-offs between sparsity and performance[7]

Sprint 3: Trading Logic & Risk Management (Week 9-11)

Component 1: TradeSignalEngine

Purpose: Convert portfolio value deviations into actionable buy/sell signals.[1]

Signal Generation Logic:

```
class TradeSignalEngine:
    def generate_signal(self, portfolio_value: float,
                        target_value: float,
                        threshold: float = 0.02) -> dict:
        """
        Mean-reversion signal: trade when portfolio deviates from target.
        """
        deviation = (portfolio_value - target_value) / target_value

        if deviation > threshold:
            # Portfolio overvalued → Sell
            signal = 'SELL'
            size = abs(deviation) * portfolio_value
        elif deviation < -threshold:
            # Portfolio undervalued → Buy
            signal = 'BUY'
            size = abs(deviation) * target_value
        else:
            signal = 'HOLD'
            size = 0

        return {'signal': signal, 'size': size, 'deviation': deviation}
```

Component 2: RiskManager

Purpose: Enforce user-defined constraints to prevent catastrophic losses.[1]

Risk Controls :[1]

1. **Position Sizing**: Kelly Criterion with conservative adjustment
2. **Max Exposure**: No single asset > 20% of capital
3. **Drawdown Stop**: Halt trading if losses exceed user threshold
4. **Volatility Scaling**: Reduce position sizes during high-volatility periods

```
class RiskManager:
    def __init__(self, profile: UserProfile):
        self.max_position = profile.capital * 0.2
        self.drawdown_limit = profile.drawdown_limit

    def check_risk(self, signal: dict,
                   current_drawdown: float) -> dict:
        """
        Validate signal against risk constraints.
        """
        # Hard stop: drawdown exceeded
        if current_drawdown > self.drawdown_limit:
            return {'approved': False, 'reason': 'Drawdown limit'}

        # Position sizing with Kelly fraction
        kelly_size = self.calculate_kelly(signal)
        adjusted_size = min(kelly_size, self.max_position)

        return {
            'approved': True,
            'adjusted_size': adjusted_size,
            'original_size': signal['size']
        }

    def calculate_kelly(self, signal: dict) -> float:
        """
        Kelly Criterion: f = (p*b - q) / b
        where p = win probability, q = 1-p, b = win/loss ratio
        """
        # Conservative: use half-Kelly to reduce variance
        win_prob = 0.55  # Estimated from backtest
        win_loss_ratio = 1.5
        kelly_fraction = (win_prob * win_loss_ratio - (1-win_prob)) / win_loss_ratio

        return signal['size'] * (kelly_fraction / 2)
```

Learning Focus: Understanding risk-adjusted returns, position sizing mathematics, and the importance of capital preservation.[8] [1]

Sprint 4: Backtester & Performance Evaluation (Week 12-14)

Component 1: Backtester

Purpose: Simulate historical strategy performance with realistic execution assumptions.[1]

Critical Considerations :[1]

- **Slippage Model**: Price impact = base_spread + α × (order_size / avg_daily_volume)^γ

- **Commission**: Fixed ($1/trade) + proportional (0.001%)
- **Fill Simulation**: Partial fills for large orders
- **Look-ahead Bias Prevention**: Ensure signals use only past data

```
class Backtester:
    def __init__(self, slippage_model: dict, commission: dict):
        self.slippage_params = slippage_model
        self.commission = commission

    def simulate(self, signals: List[dict],
                 price_data: pd.DataFrame) -> List[dict]:
        """
        Execute backtest with realistic market impact.
        """
        trades = []
        portfolio_value = self.initial_capital

        for signal in signals:
            # Calculate slippage
            adv = price_data['volume'].mean()
            impact = self.calculate_slippage(signal['size'], adv)

            # Simulate fill
            fill_price = signal['price'] * (1 + impact)
            fill_cost = self.commission['fixed'] + \
                        signal['size'] * self.commission['rate']

            # Update portfolio
            trade = {
                'timestamp': signal['timestamp'],
                'symbol': signal['symbol'],
                'qty': signal['size'],
                'fill_price': fill_price,
                'slippage': impact,
                'commission': fill_cost
            }
            trades.append(trade)

        return trades
```

Component 2: PerformanceEvaluator

Metrics to Track :[9] [1]

- **Sharpe Ratio**: Risk-adjusted returns = (μ - r_f) / σ
- **Sortino Ratio**: Downside risk-adjusted = (μ - r_f) / σ_downside
- **Maximum Drawdown**: Largest peak-to-trough decline
- **Win Rate**: Percentage of profitable trades
- **Profit Factor**: Gross profit / Gross loss

Learning Focus: Understanding the difference between absolute returns and risk-adjusted metrics, why Sharpe ratio matters to institutional investors.[8] [1]

Sprint 5: API & Dashboard (Week 15-16)

API Endpoints :[1]

```
# FastAPI routes
@app.post("/api/v1/backtest")
async def run_backtest(config: BacktestConfig):
    # Validate user inputs
    # Submit to background task queue (Celery)
    # Return job_id for status polling

@app.get("/api/v1/backtest/{job_id}")
async def get_backtest_results(job_id: str):
    # Fetch results from database
    # Return performance metrics + trade history

@app.post("/api/v1/portfolio/optimize")
async def optimize_portfolio(universe: List[str], params: dict):
    # Run PortfolioConstructor
    # Return sparse weights and expected metrics
```

Dashboard Features :[2] [1]

- User profile configuration (risk tolerance, capital)
- Universe selection interface
- Backtest results visualization (equity curve, drawdown chart)
- Trade-by-trade breakdown with explanations

Critical Recommendations

1. Start Simple, Iterate Fast[10] [2]
Build the minimal version first (manual universe selection, basic mean reversion, simple backtest), then validate with real historical data before adding complexity.[2]

2. Data Quality is Paramount[11]
Garbage in, garbage out. Spend time cleaning data: handle stock splits, adjust for dividends, filter out low-liquidity assets.[1]

3. Test Rigorously[1]
Write unit tests for mathematical functions (VAR estimation, eigenvalue computation) with known inputs/outputs. Use pytest with fixed random seeds for reproducibility.[1]

4. Document Your Assumptions[1]
Every model has limitations. Document assumptions about stationarity, transaction costs, and market impact so users understand when the strategy might fail.[1]

5. Version Control for Models[1]
Store hyperparameters, training data dates, and performance metrics for each model version. Use MLflow or simple JSON configs.[1]

Next Steps After MVP

Once the core system works, extend with :[1]

- **Phase 2**: ML predictors (LSTM for next-step returns, Seq2Seq for range predictions)
- **Phase 3**: Game theory modules (adversarial market model, multi-agent simulation)
- **Phase 4**: Explainability engine (SHAP values for model decisions)
- **Phase 5**: Production deployment with live trading paper accounts

Learning Resources

Master these concepts:

- Time-series econometrics (VAR models, cointegration testing)[5] [1]
- Convex optimization (LASSO, portfolio constraints)[6]
- Software design patterns (dependency injection, repository pattern)[4]
- Quantitative risk management (VaR, CVaR, stress testing)[1]

This MVP prioritizes **working software over perfect algorithms**. Build incrementally, test continuously, and let empirical results guide your decisions. Each component teaches critical skills in both quantitative finance and production-grade software engineering.[10] [8] [2] [1]

⁂

Phase 1 Summary: Core MVP Foundation

Phase 1 builds the **essential working system** for sparse mean-reverting portfolio trading with historical backtesting. Think of it as creating the engine before adding turbochargers (ML models) and safety sensors (game theory).[22]

What We're Building

The Five Core Components :[22]

1. **DataFetcher**: Downloads historical stock prices (OHLCV data) from APIs like Yahoo Finance. This is your data pipeline that feeds everything else.[22]
2. **FeatureEngineer**: Transforms raw prices into mathematical features the portfolio constructor needs. Calculates returns, standardizes data, and estimates the VAR(1) matrix that captures how stocks move together over time.[22]
3. **PortfolioConstructor**: The mathematical brain that selects **10-20 stocks from hundreds** using sparse mean-reversion algorithms. Solves eigenvalue problems to find asset combinations that oscillate around equilibrium (mean revert).[22]
4. **TradeSignalEngine**: Monitors portfolio value and generates buy/sell signals when prices deviate from expected mean-reverting levels. Uses threshold-based rules (e.g., trade when deviation exceeds 2%).[22]

5. **RiskManager**: The safety guard that enforces position size limits, maximum exposure per stock, and stops trading if losses exceed user-defined drawdown limits. Prevents catastrophic losses through Kelly criterion position sizing.[22]
6. **Backtester**: Simulates how the strategy would have performed historically with realistic transaction costs, slippage, and market impact. Tests the strategy on past data before risking real capital.[22]

What We're NOT Building Yet

Phase 1 deliberately **excludes advanced features** to validate the core concept first :[22]

- **No ML predictors** (LSTM/Seq2Seq models) - using simple rule-based signals instead[22]
- **No game theory modules** (adversarial markets, multi-agent simulations)[22]
- **No explainability engine** (SHAP analysis)[22]
- **No live trading** - only historical backtesting[22]

Why This Approach

Testing the Foundation: Mean reversion is the strategy's core assumption. If sparse portfolios don't show mean-reverting behavior in backtests, adding ML won't save it. Phase 1 validates whether the mathematical framework works before investing in complexity.[22]

Learning Path: As the agent building this, Phase 1 teaches you fundamental concepts—time series analysis, eigenvalue decomposition, portfolio optimization, backtesting methodology—that are prerequisites for understanding ML and game theory layers.[22]

Risk Reduction: Building modularly means each component can be tested independently. If something breaks, you know exactly where to look instead of debugging a tangled mess.[22]

Success Criteria

Phase 1 is complete when you can :[22]

- Input a list of stock symbols and date range[22]
- Output a sparse portfolio (10-20 stocks with weights)[22]
- Backtest the strategy and see performance metrics (Sharpe ratio, max drawdown)[22]
- Verify risk controls prevent excessive losses[22]
- Access results through a simple dashboard or API[22]

Once this works reliably with deterministic, reproducible results, you're ready for Phase 2's ML predictors and Phase 3's game theory enhancements.[22]

⁂

doubt : 1. is stock symbol is a stock name ;

A **stock symbol** is not the same as a **stock name**, although they are related.

- The **stock symbol** (also called **ticker symbol**) is a **unique short code** that identifies a company's stock on an exchange.
  For example:
  - AAPL → Apple Inc.
  - TSLA → Tesla Inc.
  - RELIANCE.NS → Reliance Industries (NSE, India)
- The **stock name** is the **full company name** or the **security's full title**.
  For example:
  - "Apple Inc."
  - "Tesla Inc."
  - "Reliance Industries Limited"

In your **Quant Trading Assistant**, when you refer to *stock symbols* (like `AAPL`, `TSLA`, `RELIANCE.NS`), you use them as **identifiers** to fetch market data, store time-series records, and link trades and portfolios across modules. The stock name is mainly for **display and user readability**, not for internal data handling or computation.

2. are we taking any input from the user regarding in which industry we are trying to trade or the decision is taken by the machine for best results

Based on your design document, the system offers **both options** depending on the phase and user preference.[23]

Phase 1 (MVP): User Defines the Universe

In the initial MVP, **the user provides the input** by specifying which stocks to include in the trading universe. This is called **universe selection**.[24] [25] [23]

How it works:

1. The user manually selects a list of stock symbols (e.g., "AAPL, TSLA, GOOGL, MSFT...")[23]
2. Optionally, they can filter by criteria like:
   - **Industry/Sector** (e.g., only technology stocks, only financial sector)[26] [27]
   - **Market cap** (e.g., large-cap stocks > $10B)[24]
   - **Liquidity** (e.g., only stocks with average daily volume > 1M shares)[28] [24]
3. The system then builds the **sparse portfolio** from this user-defined universe[23]

Why this approach first?:

- **Learning**: As a beginner, you learn how different sectors (technology, healthcare, energy) behave with mean-reversion strategies[29] [24]
- **Control**: Users can test specific hypotheses (e.g., "Does mean reversion work better in commodity stocks?")[24]
- **Simplicity**: Avoids the complexity of automated universe selection algorithms initially[30] [23]

Phase 2+: Machine Decides (Algorithmic Universe Selection)

After the MVP works, you can add **automated universe selection** where the machine filters thousands of stocks based on quantitative criteria.[28] [23] [24]

Example filters the machine could apply :[26] [28] [24]

1. **Liquidity Filter**: Select top 200 stocks by dollar volume (price × volume) to ensure tradability[28]
2. **Quality Screen**: Filter by financial metrics (Return on Capital > 15%, positive cash flow)[26]
3. **Volatility Filter**: Exclude stocks with volatility > 50% to reduce risk[28]
4. **Sector Diversification**: Automatically pick 5-7 stocks from each major sector[31] [27]

Implementation:

```
class UniverseSelector:
    def filter_universe(self, market_data: pd.DataFrame) -> List[str]:
        # Step 1: Liquidity filter
        market_data['dollar_volume'] = market_data['close'] * market_data['volume']
        liquid_stocks = market_data.nlargest(200, 'dollar_volume')

        # Step 2: Quality filter
        quality_stocks = liquid_stocks[
            (liquid_stocks['roce'] > 0.15) &
            (liquid_stocks['volatility'] < 0.5)
        ]

        # Step 3: Sector diversification
        selected = []
        for sector in ['Technology', 'Healthcare', 'Finance']:
            sector_stocks = quality_stocks[quality_stocks['sector'] == sector]
            selected.extend(sector_stocks.head(5)['symbol'].tolist())

        return selected
```

Recommendation: Hybrid Approach

The **best practice** is to offer **both** :[24] [26]

- **Manual mode**: User provides industry/sector preferences (e.g., "I only want tech and healthcare stocks")[23]
- **Auto-assist mode**: Machine suggests a universe based on liquidity/quality, but user approves[24]

- **Fully automated**: Machine selects optimal universe dynamically each month[28]

Why this matters: Different sectors respond differently to mean-reversion strategies. Technology stocks might be more momentum-driven, while utilities/commodities show stronger mean reversion. Giving users control helps them discover which sectors work best with your algorithm.[32] [27] [29] [24]

Your Database Schema Supports This

Looking at your design, the `PortfolioConstructor` already has a `universe` field (list of symbols). You can populate this either from:[23]

- User input via API (`POST /api/v1/portfolio/optimize` with `universe` parameter)[23]
- Automated `UniverseSelector` component that runs before portfolio construction[24] [28]

Start with **user-defined universes in Phase 1** to validate the core algorithm, then add intelligent filters in Phase 2 once you understand which selection criteria improve performance.[30] [23] [24]

⁂

Step-by-Step Instructions: Completing Phase 1 of Quantitative Trading Assistant

These steps assume you are starting from basics, have Anaconda installed, and wish to build this on your local machine (MacBook or HP Victus). All instructions produce code that is simple, modular, and suited for absolute beginners eager to learn.

Step 1: Create a New Conda Environment

Open Terminal and run:

```
conda create --name quant-trade python=3.11
conda activate quant-trade
```

Reason: Isolates your work and avoids version conflicts.

Step 2: Install Essential Dependencies

Install the core packages for data science, backend, and trading logic:

```
conda install numpy pandas scikit-learn matplotlib ipython fastapi uvicorn psycopg2
pip install yfinance plotly
```

Reason: These are the standard tools for numerical analysis, data fetching, and API building. yfinance simplifies stock data access for starters.

Step 3: Set Up Directory Structure

Within your project folder, structure code for clarity and modularity:

```
quant-trading-assistant/
├── app/
│   ├── api/              # FastAPI REST endpoints
│   ├── models/           # Database models (Python classes)
│   ├── services/         # Core business logic (feature engineering, trading)
│   ├── core/             # Configuration files
│   └── utils/            # Helper functions
├── data/                 # Store raw and processed data files
├── tests/                # Test scripts and notebooks
├── requirements.txt
└── README.md
```

Reason: Clean folder design enables code reuse and upgrades.

Step 4: Design User Profile Management

Make a simple Python file to manage user preferences. Example:

```
# app/models/user.py
class UserProfile:
    def __init__(self, user_id, risk_tolerance, capital, max_assets, drawdown_limit):
        self.user_id = user_id
        self.risk_tolerance = risk_tolerance
        self.capital = capital
        self.max_assets = max_assets
        self.drawdown_limit = drawdown_limit
```

Reason: User profile is key for risk controls and customizing the assistant.

Step 5: Implement Data Fetcher

For Phase 1, use yfinance to get historical stock prices:

```
import yfinance as yf
import pandas as pd

def fetch_ohlcv(symbols, start, end):
    all_data = {}
    for sym in symbols:
        df = yf.download(sym, start=start, end=end)
        all_data[sym] = df
    return all_data
```

Try this for your chosen universe:

```
symbols = ['AAPL', 'MSFT', 'GOOGL']
data = fetch_ohlcv(symbols, '2020-01-01', '2022-01-01')
```

Reason: You learn the essentials of market data—the backbone of quant trading.

Step 6: Feature Engineering

Write code to calculate basic features:

```
def compute_log_returns(df):
    return np.log(df['Close'] / df['Close'].shift(1)).dropna()

# Do this for each symbol
returns = {sym: compute_log_returns(df) for sym, df in data.items()}
```

Reason: Log returns are the core input for mean-reversion models.

Step 7: Sparse Mean-Reverting Portfolio Construction

Implement the skeleton for your algorithm:

```
from sklearn.covariance import GraphicalLassoCV
import numpy as np

def sparse_covariance(returns_matrix):
    model = GraphicalLassoCV()
    model.fit(returns_matrix)
    return model.precision_

def select_sparse_portfolio(returns_dict, k):
    returns_matrix = np.array([returns_dict[sym].values for sym in returns_dict])
    returns_matrix = returns_matrix.T  # Time x Assets
    precision = sparse_covariance(returns_matrix)
    # Placeholder: select top-k assets based on variance or eigenvector magnitude
    variances = np.diag(precision)
    selected_ix = np.argsort(variances)[:k]
    selected_symbols = [list(returns_dict.keys())[ix] for ix in selected_ix]
    return selected_symbols
```

Reason: You begin learning sparse optimization and asset selection.

Step 8: Trade Signal Engine

Create a function to generate simple trade signals:

```
def trade_signal(portfolio_value, target_value, threshold=0.02):
    deviation = (portfolio_value - target_value) / target_value
```

```
    if deviation > threshold:
        return 'SELL', deviation
    elif deviation < -threshold:
        return 'BUY', -deviation
    else:
        return 'HOLD', 0
```

Reason: This teaches basic algorithmic trading logic.

Step 9: Risk Management

Apply basic checks:

```
def check_risk(signal, user_profile, position_size):
    if position_size > user_profile.capital * 0.2:
        return False, 'Exceeded max position size'
    # Add drawdown, volatility checks as needed
    return True, 'OK'
```

Reason: Fundamental for preventing losses and learning real-world trading discipline.

Step 10: Backtesting

Simulate performance using historical data:

```
def simple_backtest(trade_signals, returns):
    portfolio_value = 1.0
    values = []
    for signal, ret in zip(trade_signals, returns):
        if signal == 'BUY':
            portfolio_value *= (1 + ret)
        elif signal == 'SELL':
            portfolio_value *= (1 - ret)
        values.append(portfolio_value)
    return pd.Series(values)
```

Reason: Validates if your strategy would actually work.

Step 11: API and Dashboard (Optional for phase 1)

Once you have core functions, expose them via FastAPI:

```
from fastapi import FastAPI

app = FastAPI()

@app.post("/run_strategy/")
def run_strategy(universe: list, start: str, end: str):
```

```
    # Execute core logic above and return portfolio and stats
    return {"status": "ok"}
```

Reason: Lays the foundation for a user interface and further automation.

Step 12: Testing and Validation

Make a Jupyter notebook in your `/tests` folder to:

- Try different universes and parameters
- Print portfolio weights and backtest results
- Visualize performance (e.g., plot equity curve with matplotlib or plotly)
- Save results to `/data`

Reason: Iterative testing is the habit of all successful quantitative developers.

Step 13: Document Everything

Create a README.md with simple instructions, list of features completed, design choices, and usage examples.
Reason: Professional communication and reproducibility are vital.

Step 14: Version Control

Initialize git and commit code as you progress:

```
git init
git add .
git commit -m "Initial MVP code for quant trading assistant"
```

Reason: Tracks your learning, changes, and enables collaboration.

Expert Advice as Your Mentor

- **Focus first on working, readable code.** Don't optimize prematurely.
- **Start by processing only 2-3 stocks and 1 year of data**—expand once you're confident.
- **Print your intermediate results** at each step (data shape, statistics, selected assets).
- **If stuck, Google errors, inspect with print/debug, and iterate slowly.**
- **Ask questions at every step to clarify why**—not just how—you are building each component.
- **Maintain discipline with tests and documentation.**
- Once phase 1 works, plan to refactor, add more assets, and modularize your code for robustness.

Completion Criteria for Phase 1:

- Given a user-selected universe, your system can build a sparse portfolio algorithmically, simulate trades using historical data, apply basic risk checks, and display results—all using simple modular Python scripts.
- You understand every line you write and why it matters.

This stepwise process gives you a strong foundation and the real skills needed for advanced quantitative finance development.

⁂

Complete Roadmap & Clear Picture: Quantitative Trading Assistant

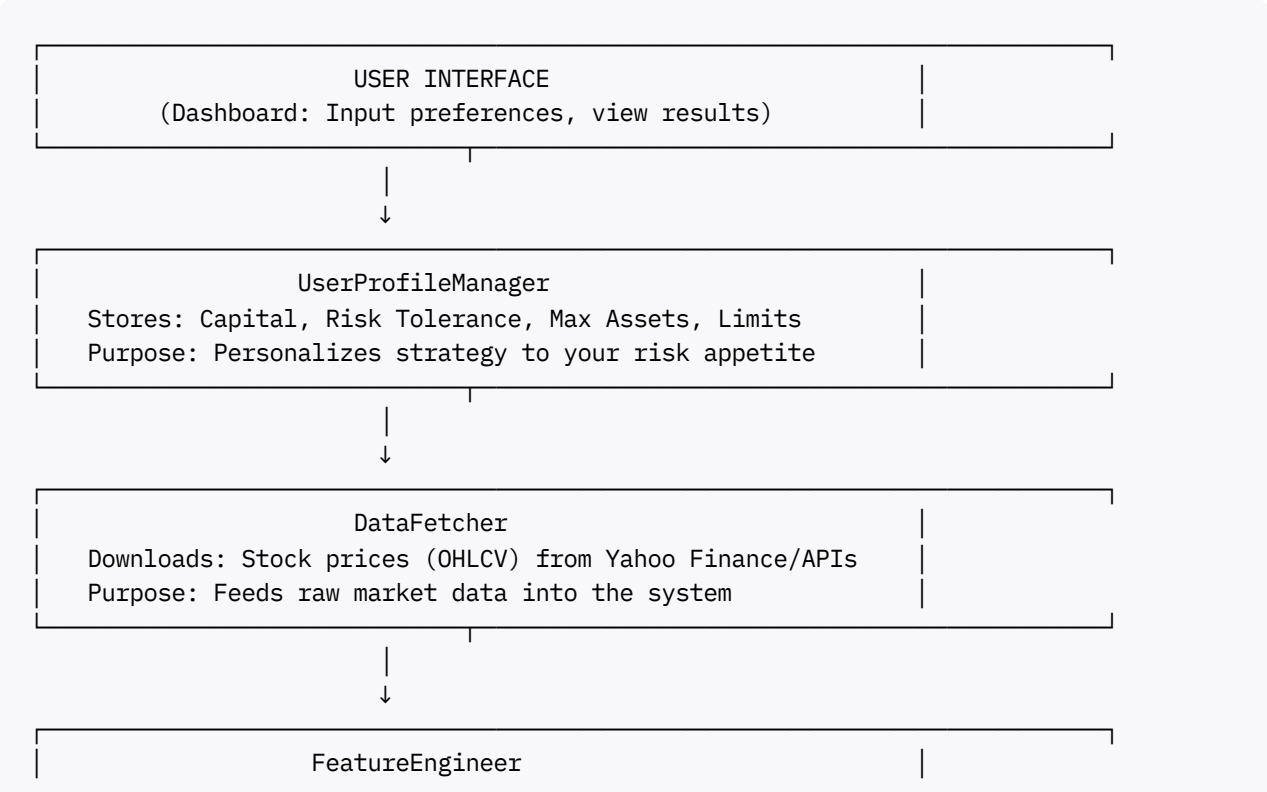
Here's a comprehensive visual roadmap showing what we're building, how components interact, and the step-by-step implementation plan.[46]

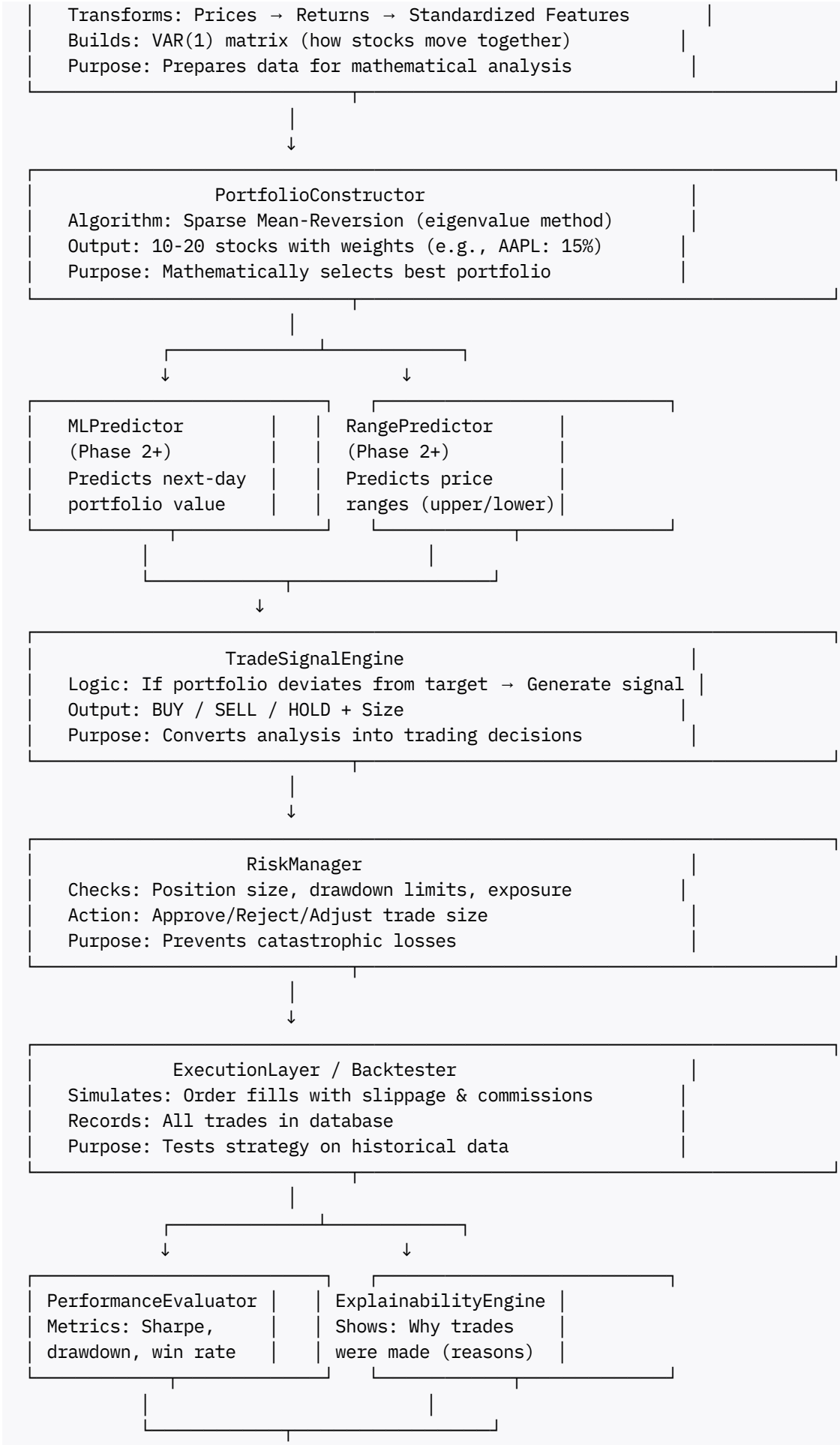
The Big Picture: What Are We Building?

A **smart trading assistant** that automatically selects profitable stock portfolios using mathematical algorithms, tests them on historical data, manages risk, and eventually uses AI to predict market movements. Think of it as your personal quantitative analyst that works 24/7.[46]

System Architecture: How Components Work Together

Component Map & Data Flow




Transforms: Prices → Returns → Standardized Features
Builds: VAR(1) matrix (how stocks move together)
Purpose: Prepares data for mathematical analysis
PortfolioConstructor
Algorithm: Sparse Mean-Reversion (eigenvalue method)
Output: 10-20 stocks with weights (e.g., AAPL: 15%)
Purpose: Mathematically selects best portfolio
MLPredictor
(Phase 2+)
Predicts next-day
portfolio value
RangePredictor
(Phase 2+)
Predicts price
ranges (upper/lower)
TradeSignalEngine
Logic: If portfolio deviates from target → Generate signal
Output: BUY / SELL / HOLD + Size
Purpose: Converts analysis into trading decisions
RiskManager
Checks: Position size, drawdown limits, exposure
Action: Approve/Reject/Adjust trade size
Purpose: Prevents catastrophic losses
ExecutionLayer / Backtester
Simulates: Order fills with slippage & commissions
Records: All trades in database
Purpose: Tests strategy on historical data
PerformanceEvaluator
Metrics: Sharpe,
drawdown, win rate
ExplainabilityEngine
Shows: Why trades
were made (reasons)

```
                    ↓
┌──────────────────────────────────────────────────────────────────────┐
│                        Dashboard/API                                 │
│   Displays: Equity curves, metrics, trade history                    │
│   Purpose: Visualize results and control system                      │
└──────────────────────────────────────────────────────────────────────┘
```

Phase-by-Phase Implementation Plan

Phase 0: Foundation (Week 1-2)

Goal: Set up development environment and infrastructure.[46]

Tasks:

1. Create project folder structure
2. Set up Git version control
3. Initialize Conda environment with dependencies
4. Create database schema (PostgreSQL)
5. Write basic configuration files

Deliverables: Empty but organized codebase ready for development.[46]

Phase 1: Core MVP (Week 3-8) — THIS IS PRIORITY

Goal: Build a working system that selects portfolios and backtests them.[46]

Sprint 1: Data Pipeline (Week 3-4)

Component 1: DataFetcher

- **Input**: List of stock symbols (e.g., ['AAPL', 'MSFT', 'GOOGL']), date range
- **Process**: Downloads historical OHLCV (Open, High, Low, Close, Volume) data from yfinance
- **Output**: Pandas DataFrame with daily prices indexed by date
- **Why**: Without data, nothing else can work[46]

Component 2: FeatureEngineer

- **Input**: Raw price DataFrame
- **Process**:
  - Calculate log returns: $ return_t = \log(price*t / price*{t-1}) $
  - Standardize (z-score): $ z = (return - \mu) / \sigma $
  - Build VAR(1) matrix: Shows how each stock's movement predicts others
- **Output**: Standardized feature matrix + VAR(1) coefficient matrix

- **Why**: Mathematical models need normalized inputs[46]

Sprint 2: Portfolio Construction (Week 5-6)

Component 3: PortfolioConstructor

- **Input**: VAR(1) matrix, universe of stocks, sparsity parameter k (e.g., select 15 stocks)
- **Process**:
  1. Solve eigenvalue problem to find mean-reverting combinations
  2. Select eigenvector with smallest eigenvalue (most stationary)
  3. Keep only top-k stocks by weight magnitude (sparsity)
  4. Normalize weights to sum to 1
- **Output**: Portfolio object with {symbols: weights} (e.g., {'AAPL': 0.12, 'MSFT': 0.18, ...})
- **Why**: This is the core intelligence—finding stocks that oscillate together[46]

Mathematical Concept: Mean reversion assumes prices return to equilibrium. Eigenvalue decomposition finds linear combinations of stocks that are **stationary** (don't drift away).[46]

Sprint 3: Trading Logic & Risk (Week 7)

Component 4: TradeSignalEngine

- **Input**: Current portfolio value, target value, deviation threshold
- **Process**:
  - If portfolio value > target + threshold → SELL (overvalued)
  - If portfolio value < target - threshold → BUY (undervalued)
  - Otherwise → HOLD
- **Output**: Signal object {action: 'BUY', size: 1000 shares, deviation: 2.3%}
- **Why**: Converts mathematical analysis into actionable trades[46]

Component 5: RiskManager

- **Input**: Trade signal, user profile (risk tolerance, capital, limits)
- **Process**:
  - Check: Is position size < 20% of capital? ✓
  - Check: Is drawdown < user limit? ✓
  - Check: Is volatility acceptable? ✓
  - Calculate: Kelly criterion for optimal size
- **Output**: Approved signal with adjusted size OR rejection with reason
- **Why**: Prevents blowing up your account[46]

Sprint 4: Backtesting (Week 8)

Component 6: Backtester

- **Input**: Historical data, portfolio, trading signals
- **Process**:
  - Simulate each trade day-by-day
  - Apply realistic costs: slippage (price impact), commissions
  - Track portfolio value over time
  - Record every trade with details
- **Output**:
  - Trade history DataFrame
  - Performance metrics (Sharpe ratio, max drawdown, total return)
- **Why**: Validates if the strategy would have actually worked[46]

Component 7: PerformanceEvaluator

- **Input**: Backtest results
- **Process**: Calculate key metrics:
  - **Sharpe Ratio**: Risk-adjusted returns = (avg return - risk-free rate) / volatility
  - **Max Drawdown**: Worst peak-to-trough decline
  - **Win Rate**: % of profitable trades
  - **Sortino Ratio**: Penalizes only downside volatility
- **Output**: Metrics dictionary + visualizations (equity curve, drawdown chart)
- **Why**: Tells you if the strategy is worth deploying[46]

Phase 2: Machine Learning Enhancement (Week 9-12)

Goal: Add AI predictions to improve signal timing.[46]

Component 8: MLPredictor (StepAhead)

- **Input**: Recent 30 days of features
- **Model**: LSTM neural network
- **Output**: Prediction of tomorrow's portfolio value
- **Integration**: TradeSignalEngine uses prediction to adjust thresholds
- **Why**: ML can capture patterns eigenvalue methods miss[46]

Component 9: RangePredictor (Seq2Seq)

- **Input**: Recent 30 days of features
- **Model**: Sequence-to-sequence LSTM

- **Output**: Predicted price range (upper/lower bounds) for next 5 days
- **Integration**: RiskManager uses range to assess uncertainty
- **Why**: Knowing prediction confidence helps size positions[46]

Component 10: ExplainabilityEngine

- **Input**: Trade decision + model predictions
- **Process**: Use SHAP (SHapley Additive exPlanations) to show which features mattered most
- **Output**: "Trade triggered because: AAPL z-score = -2.1 (oversold), LSTM predicted 3% rebound"
- **Why**: Users need to trust and understand decisions[46]

Phase 3: Game Theory Integration (Week 13-16)

Goal: Model adversarial markets and multi-agent dynamics.[46]

Component 11: GameTheoryManager

Sub-component 11a: AdversarialMarketModel

- **Purpose**: Simulate worst-case price impact when you trade
- **Process**:
  - Generate 100 adversarial price paths (market moving against you)
  - Calculate PnL under each scenario
  - If worst 10% outcomes are catastrophic → reduce position size
- **Why**: Large orders move markets; plan for it[46]

Sub-component 11b: MultiAgentSimulator

- **Purpose**: Test strategy against other trading algorithms
- **Agents**: Mean-reverter (you), Momentum trader, Noise trader, Market maker
- **Process**: Simulate 1000 trading days with all agents competing
- **Output**: Which strategy dominates? Where does yours fail?
- **Why**: Real markets have competing strategies[46]

Sub-component 11c: EquilibriumAnalyzer

- **Purpose**: Find Nash equilibrium (stable strategy mix)
- **Process**:
  - Build payoff matrix (your return vs each opponent strategy)
  - Solve for Nash equilibrium using game theory math
- **Output**: "Your strategy is optimal 60% of the time, vulnerable to momentum traders"
- **Why**: Understand long-term viability[46]

Phase 4: Production Deployment (Week 17+)

Goal: Make system production-ready with live trading capability.[46]

Tasks:

1. **API Development**: REST endpoints for all operations
2. **Dashboard**: React frontend with real-time charts
3. **Database Optimization**: Index time-series tables, partition by date
4. **Background Jobs**: Celery workers for long-running backtests
5. **Monitoring**: Logging, error tracking, performance dashboards
6. **Paper Trading**: Connect to broker API (Alpaca, Interactive Brokers) for simulated trading
7. **Security**: Authentication, rate limiting, input validation

Technology Stack Summary

Layer	Technology	Purpose
Language	Python 3.11	Core development [46]
Data Science	NumPy, Pandas, SciPy, scikit-learn	Mathematical computations [46]
Data Source	yfinance (MVP), Alpha Vantage (production)	Stock price data [46]
ML Framework	PyTorch/TensorFlow	LSTM models [46]
Backend API	FastAPI	REST endpoints [46]
Database	PostgreSQL + TimescaleDB	Relational + time-series [46]
Task Queue	Celery	Background jobs [46]
Frontend	React + Chart.js	Dashboard [46]
Deployment	Docker + Kubernetes	Containerization [46]
Version Control	Git + GitHub	Code management [46]

Database Schema (Key Tables)

users

```
id (PK), name, email, risk_tolerance, capital, max_assets, drawdown_limit
```

Stores user preferences.[46]

ohlcv_daily

```
id (PK), symbol (FK), date, open, high, low, close, volume
```

Historical price data.[46]

portfolios

```
id (PK), created_at, assets (JSONB), weights (JSONB), method
```

Stored portfolio configurations.[46]

trades

```
id (PK), user_id, timestamp, symbol, qty, price, slippage, side, pnl
```

Every trade recorded for audit.[46]

Learning Path for Agent (You)

Week 1-2: Foundations

- Understand time-series data structures (Pandas)
- Learn basic statistics (mean, variance, correlation)
- Study mean reversion concept in finance

Week 3-4: Mathematical Core

- Linear algebra: eigenvalues, eigenvectors
- Time-series models: VAR, stationarity
- Cointegration testing

Week 5-6: Optimization

- Sparse optimization (LASSO, L1 regularization)
- Portfolio theory (Markowitz, Kelly criterion)

Week 7-8: Backtesting

- Market microstructure (bid-ask spreads, slippage)
- Performance metrics (Sharpe, Sortino, drawdown)

Week 9-12: Machine Learning

- LSTM architecture and training
- Sequence-to-sequence models
- Model explainability (SHAP)

Week 13-16: Game Theory

- Nash equilibrium concepts
- Multi-agent reinforcement learning
- Adversarial robustness

Success Metrics

Phase 1 Complete When:

- You can input symbols + date range → get sparse portfolio ✓
- Backtest runs without errors ✓
- Sharpe ratio > 1.0 on historical data ✓
- Risk controls prevent excessive drawdown ✓
- Results are reproducible (same inputs → same outputs) ✓

Phase 2 Complete When:

- ML models improve Sharpe ratio by 15%+ ✓
- Explainability shows logical feature importance ✓
- System handles 200 stocks efficiently ✓

Phase 3 Complete When:

- Adversarial adjustment reduces tail risk ✓
- Multi-agent simulations reveal strategy weaknesses ✓
- Nash analysis confirms strategy robustness ✓

This roadmap transforms you from beginner to expert by building increasingly sophisticated components, each teaching critical quantitative finance concepts. Every component has a clear purpose, input/output, and connects logically to the next.[46]

⁂

Phase 1: Complete Step-by-Step Implementation Guide

Here's a detailed, hands-on walkthrough to build Phase 1 from scratch. Follow each step carefully—every command, every line of code is explained so you understand what you're building and why.[^7_1]

WEEK 0: Environment Setup (Day 1-2)

Step 1: Create Project Directory

Open Terminal on your MacBook and run:

```
# Navigate to your preferred workspace
cd ~/Documents

# Create project folder
mkdir quant-trading-assistant
cd quant-trading-assistant

# Initialize Git
git init
```

Why: Organizing your work in a dedicated folder with version control from day one is professional practice.[^7_2]

Step 2: Set Up Conda Environment

```
# Create isolated Python environment
conda create --name quant-trade python=3.11 -y

# Activate it
conda activate quant-trade
```

Why: Isolates dependencies so they don't conflict with other projects.[^7_3]

Step 3: Install Core Dependencies

```
# Data science stack
conda install numpy pandas scipy scikit-learn matplotlib seaborn jupyter -y

# Financial data
pip install yfinance

# Backend framework
pip install fastapi uvicorn[standard]

# Database
```

```
pip install psycopg2-binary sqlalchemy

# Testing
pip install pytest pytest-cov
```

Why: These are industry-standard tools for quantitative finance.[7_4][7_3][^7_1]

Step 4: Install PostgreSQL Database

On macOS :[7_5][7_6]

```
# Install Homebrew (if not already installed)
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/instal

# Install PostgreSQL
brew install postgresql@15

# Start PostgreSQL service
brew services start postgresql@15

# Verify installation
psql --version
```

Create your database:

```
# Access PostgreSQL
psql postgres

# Inside psql prompt, run:
CREATE DATABASE quant_trading_db;
CREATE USER quant_user WITH PASSWORD 'secure_password';
GRANT ALL PRIVILEGES ON DATABASE quant_trading_db TO quant_user;
\q  # Exit
```

Why: PostgreSQL stores user profiles, portfolios, trades, and historical data reliably.[7_5][7_1]

Step 5: Create Project Structure

```
# Create folders
mkdir -p app/{api,models,services,core,utils}
mkdir -p data/{raw,processed}
mkdir -p tests
mkdir notebooks

# Create essential files
touch app/__init__.py
touch app/api/__init__.py
touch app/models/__init__.py
touch app/services/__init__.py
touch app/core/__init__.py
```

```
touch app/utils/__init__.py
touch requirements.txt
touch README.md
touch .gitignore
```

Create `.gitignore`:

```
echo "*.pyc
__pycache__/
.env
data/raw/*
*.db
.DS_Store
notebooks/.ipynb_checkpoints/" > .gitignore
```

Why: Professional structure makes collaboration and debugging easier.[7_3][7_4]

WEEK 1: Database Schema & Configuration (Day 3-5)

Step 6: Database Models

Create `app/models/database.py`:

```
from sqlalchemy import create_engine, Column, Integer, String, Float, DateTime, JSON, For
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, relationship
from datetime import datetime

Base = declarative_base()

# User Profile Model
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String(100))
    email = Column(String(100), unique=True)
    risk_tolerance = Column(Float)  # 0.0 to 1.0
    capital = Column(Float)
    max_assets = Column(Integer, default=20)
    drawdown_limit = Column(Float, default=0.25)  # 25% max drawdown
    created_at = Column(DateTime, default=datetime.utcnow)

    # Relationships
    portfolios = relationship("Portfolio", back_populates="user")
    trades = relationship("Trade", back_populates="user")

# Market Data Model
class MarketData(Base):
    __tablename__ = 'market_data'

    id = Column(Integer, primary_key=True)
```

```
    symbol = Column(String(10), index=True)
    date = Column(DateTime, index=True)
    open = Column(Float)
    high = Column(Float)
    low = Column(Float)
    close = Column(Float)
    volume = Column(Integer)

# Portfolio Model
class Portfolio(Base):
    __tablename__ = 'portfolios'

    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('users.id'))
    created_at = Column(DateTime, default=datetime.utcnow)
    assets = Column(JSON)  # {"AAPL": 0.15, "MSFT": 0.20, ...}
    method = Column(String(50))  # "sparse_mr", "ml_enhanced", etc.
    backtest_sharpe = Column(Float)
    backtest_max_drawdown = Column(Float)

    user = relationship("User", back_populates="portfolios")

# Trade Model
class Trade(Base):
    __tablename__ = 'trades'

    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey('users.id'))
    portfolio_id = Column(Integer, ForeignKey('portfolios.id'))
    timestamp = Column(DateTime, default=datetime.utcnow)
    symbol = Column(String(10))
    side = Column(String(4))  # 'BUY' or 'SELL'
    quantity = Column(Integer)
    price = Column(Float)
    slippage = Column(Float)
    commission = Column(Float)
    pnl = Column(Float)

    user = relationship("User", back_populates="trades")

# Database connection
DATABASE_URL = "postgresql://quant_user:secure_password@localhost/quant_trading_db"
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(bind=engine)

# Create all tables
def init_db():
    Base.metadata.create_all(bind=engine)
```

Why: SQLAlchemy ORM makes database interactions Pythonic and type-safe.[7_1][7_3]

Step 7: Configuration Management

Create `app/core/config.py`:

```
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    # Database
    DATABASE_URL: str = "postgresql://quant_user:secure_password@localhost/quant_trading_

    # API
    API_V1_PREFIX: str = "/api/v1"
    PROJECT_NAME: str = "Quant Trading Assistant"

    # Trading Parameters
    DEFAULT_SPARSITY: int = 15  # Number of assets in portfolio
    DEVIATION_THRESHOLD: float = 0.02  # 2% threshold for trade signals
    MAX_POSITION_SIZE: float = 0.20  # 20% max per asset

    # Data Sources
    DATA_START_DATE: str = "2020-01-01"
    DATA_END_DATE: str = "2024-10-01"

    class Config:
        env_file = ".env"

settings = Settings()
```

Why: Centralizes configuration for easy adjustment.[7_4][7_3]

Step 8: Initialize Database

Create `scripts/init_db.py`:

```
from app.models.database import init_db, SessionLocal, User

def create_test_user():
    db = SessionLocal()

    # Create a test user
    user = User(
        name="Test User",
        email="test@example.com",
        risk_tolerance=0.5,
        capital=100000.0,
        max_assets=15,
        drawdown_limit=0.25
    )

    db.add(user)
    db.commit()
    print(f"Created user: {user.name} (ID: {user.id})")
    db.close()
```

```
if __name__ == "__main__":
    print("Initializing database...")
    init_db()
    print("Database initialized!")

    print("Creating test user...")
    create_test_user()
    print("Done!")
```

Run it:

```
python scripts/init_db.py
```

Why: Sets up your database schema and creates a test user to work with.[^7_1]

WEEK 2: Data Fetcher & Feature Engineer (Day 6-10)

Step 9: Data Fetcher Service

Create `app/services/data_fetcher.py`:

```
import yfinance as yf
import pandas as pd
from datetime import datetime
from typing import List, Dict
from app.models.database import SessionLocal, MarketData

class DataFetcher:
    """Downloads and caches historical market data"""

    def __init__(self):
        self.cache = {}

    def fetch_ohlcv(self, symbols: List[str], start: str, end: str) -> Dict[str, pd.DataF
        """
        Fetch OHLCV data for given symbols.

        Args:
            symbols: List of ticker symbols (e.g., ['AAPL', 'MSFT'])
            start: Start date 'YYYY-MM-DD'
            end: End date 'YYYY-MM-DD'

        Returns:
            Dictionary mapping symbol to DataFrame with OHLCV data
        """
        data = {}

        for symbol in symbols:
            print(f"Fetching {symbol}...")

            try:
```

```
                # Download from yfinance
                df = yf.download(symbol, start=start, end=end, progress=False)

                if df.empty:
                    print(f"Warning: No data for {symbol}")
                    continue

                # Clean column names
                df.columns = [col.lower() for col in df.columns]

                # Store in cache
                data[symbol] = df

                # Optionally save to database
                self._save_to_db(symbol, df)

            except Exception as e:
                print(f"Error fetching {symbol}: {e}")
                continue

        return data

    def _save_to_db(self, symbol: str, df: pd.DataFrame):
        """Save market data to database"""
        db = SessionLocal()

        for date, row in df.iterrows():
            market_data = MarketData(
                symbol=symbol,
                date=date,
                open=row['open'],
                high=row['high'],
                low=row['low'],
                close=row['close'],
                volume=int(row['volume'])
            )
            db.add(market_data)

        db.commit()
        db.close()

    def load_from_db(self, symbols: List[str], start: str, end: str) -> Dict[str, pd.Data
        """Load cached data from database"""
        db = SessionLocal()
        data = {}

        for symbol in symbols:
            query = db.query(MarketData).filter(
                MarketData.symbol == symbol,
                MarketData.date >= start,
                MarketData.date <= end
            ).order_by(MarketData.date)

            records = query.all()

            if records:
```

```
                df = pd.DataFrame([{
                    'date': r.date,
                    'open': r.open,
                    'high': r.high,
                    'low': r.low,
                    'close': r.close,
                    'volume': r.volume
                } for r in records])

                df.set_index('date', inplace=True)
                data[symbol] = df

        db.close()
        return data
```

Test it in a Jupyter notebook (`notebooks/test_data_fetcher.ipynb`):

```
from app.services.data_fetcher import DataFetcher

# Initialize
fetcher = DataFetcher()

# Test with a few stocks
symbols = ['AAPL', 'MSFT', 'GOOGL', 'TSLA', 'JPM']
data = fetcher.fetch_ohlcv(symbols, '2022-01-01', '2024-01-01')

# Inspect
for symbol, df in data.items():
    print(f"\n{symbol}:")
    print(df.head())
    print(f"Shape: {df.shape}")
```

Why: Real data is the foundation—without it, nothing else works.[7_3][7_1]

Step 10: Feature Engineer Service

Create `app/services/feature_engineer.py`:

```
import numpy as np
import pandas as pd
from typing import Dict, Tuple
from sklearn.preprocessing import StandardScaler

class FeatureEngineer:
    """Transforms raw price data into features for portfolio construction"""

    def compute_returns(self, data: Dict[str, pd.DataFrame]) -> pd.DataFrame:
        """
        Calculate log returns for all assets.

        Args:
            data: Dictionary of symbol -> OHLCV DataFrame
```

```
        Returns:
            DataFrame with returns (rows=dates, columns=symbols)
        """
        returns_dict = {}

        for symbol, df in data.items():
            # Log returns: ln(P_t / P_{t-1})
            returns = np.log(df['close'] / df['close'].shift(1))
            returns_dict[symbol] = returns

        # Combine into single DataFrame
        returns_df = pd.DataFrame(returns_dict)
        returns_df = returns_df.dropna()  # Remove first row (NaN)

        return returns_df

    def standardize_features(self, returns: pd.DataFrame) -> pd.DataFrame:
        """
        Z-score normalization: (x - mean) / std

        Args:
            returns: DataFrame of returns

        Returns:
            Standardized returns
        """
        scaler = StandardScaler()
        standardized = scaler.fit_transform(returns)

        return pd.DataFrame(
            standardized,
            index=returns.index,
            columns=returns.columns
        )

    def estimate_VAR1_matrix(self, returns: pd.DataFrame) -> np.ndarray:
        """
        Estimate VAR(1) coefficient matrix using OLS.

        VAR(1) model: R_t = A * R_{t-1} + Z_t
        where A is the coefficient matrix we want to estimate.

        Args:
            returns: Standardized returns DataFrame

        Returns:
            A matrix (n_assets x n_assets)
        """
        # Prepare lagged data
        X = returns.values[:-1]  # R_{t-1}
        Y = returns.values[1:]   # R_t

        # OLS estimation: A = (X^T X)^{-1} X^T Y
        A = np.linalg.lstsq(X, Y, rcond=None)[^7_0]
```

```
        return A

    def compute_covariance(self, returns: pd.DataFrame) -> np.ndarray:
        """
        Calculate covariance matrix of returns.

        Args:
            returns: Returns DataFrame

        Returns:
            Covariance matrix
        """
        return np.cov(returns.T)

    def pipeline(self, data: Dict[str, pd.DataFrame]) -> Tuple[pd.DataFrame, np.ndarray,
        """
        Complete feature engineering pipeline.

        Returns:
            (standardized_returns, VAR1_matrix, covariance_matrix)
        """
        # Step 1: Compute returns
        returns = self.compute_returns(data)
        print(f"Computed returns for {len(returns.columns)} assets over {len(returns)} da

        # Step 2: Standardize
        standardized_returns = self.standardize_features(returns)

        # Step 3: Estimate VAR(1)
        var1_matrix = self.estimate_VAR1_matrix(standardized_returns)

        # Step 4: Compute covariance
        cov_matrix = self.compute_covariance(standardized_returns)

        return standardized_returns, var1_matrix, cov_matrix
```

Test it:

```
from app.services.feature_engineer import FeatureEngineer

# Use data from previous step
engineer = FeatureEngineer()
returns, var1, cov = engineer.pipeline(data)

print("Returns shape:", returns.shape)
print("\nVAR(1) matrix shape:", var1.shape)
print("\nFirst eigenvalue of VAR(1):", np.linalg.eigvals(var1)[^7_0])
```

Why: Raw prices are non-stationary; returns and normalization make them suitable for mathematical models.[7_7][7_1]

WEEK 3: Portfolio Constructor (Day 11-15)

Step 11: Sparse Mean-Reverting Portfolio Constructor

Create `app/services/portfolio_constructor.py`:

```
import numpy as np
import pandas as pd
from typing import Dict, List
from sklearn.covariance import GraphicalLassoCV
from scipy import linalg

class PortfolioConstructor:
    """Constructs sparse mean-reverting portfolios using eigenvalue decomposition"""

    def __init__(self, sparsity_k: int = 15):
        """
        Args:
            sparsity_k: Number of assets to select
        """
        self.sparsity_k = sparsity_k

    def sparse_covariance_estimation(self, returns: pd.DataFrame, alpha: float = 0.1) ->
        """
        Estimate sparse precision (inverse covariance) matrix using Graphical LASSO.

        Args:
            returns: Standardized returns
            alpha: Regularization parameter (higher = sparser)

        Returns:
            Sparse precision matrix
        """
        model = GraphicalLassoCV(alphas=[alpha])
        model.fit(returns)

        return model.precision_

    def box_tiao_decomposition(self, var1_matrix: np.ndarray,
                               cov_matrix: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
        """
        Box-Tiao canonical decomposition to find mean-reverting portfolios.

        Solves generalized eigenvalue problem:
        var1_matrix @ v = lambda @ cov_matrix @ v

        Args:
            var1_matrix: VAR(1) coefficient matrix
            cov_matrix: Covariance matrix

        Returns:
            (eigenvalues, eigenvectors)
        """
        # Solve generalized eigenvalue problem
        eigenvalues, eigenvectors = linalg.eig(var1_matrix, cov_matrix)
```

```
        # Convert to real (discard tiny imaginary parts from numerical errors)
        eigenvalues = np.real(eigenvalues)
        eigenvectors = np.real(eigenvectors)

        return eigenvalues, eigenvectors

    def select_sparse_portfolio(self, eigenvectors: np.ndarray,
                                eigenvalues: np.ndarray,
                                symbols: List[str]) -> Dict[str, float]:
        """
        Select the most mean-reverting portfolio with sparsity constraint.

        Strategy:
        1. Pick eigenvector with smallest |eigenvalue| (most mean-reverting)
        2. Keep only top-k assets by absolute weight
        3. Normalize to sum = 1

        Args:
            eigenvectors: Eigenvectors from Box-Tiao
            eigenvalues: Corresponding eigenvalues
            symbols: List of asset symbols

        Returns:
            Portfolio weights {symbol: weight}
        """
        # Find most mean-reverting direction (smallest |eigenvalue|)
        idx = np.argmin(np.abs(eigenvalues))
        weights = eigenvectors[:, idx]

        # Enforce sparsity: keep top-k by absolute value
        abs_weights = np.abs(weights)
        top_k_indices = np.argsort(abs_weights)[-self.sparsity_k:]

        # Create sparse weights
        sparse_weights = np.zeros_like(weights)
        sparse_weights[top_k_indices] = weights[top_k_indices]

        # Normalize to sum = 1 (fully invested)
        sparse_weights = sparse_weights / np.sum(sparse_weights)

        # Convert to dictionary
        portfolio = {
            symbols[i]: float(sparse_weights[i])
            for i in top_k_indices
        }

        return portfolio

    def construct_portfolio(self, returns: pd.DataFrame,
                            var1_matrix: np.ndarray,
                            cov_matrix: np.ndarray) -> Dict[str, float]:
        """
        Main pipeline: construct sparse mean-reverting portfolio.

        Returns:
```

```
            Portfolio weights {symbol: weight}
        """
        # Step 1: Box-Tiao decomposition
        eigenvalues, eigenvectors = self.box_tiao_decomposition(var1_matrix, cov_matrix)

        # Step 2: Select sparse portfolio
        portfolio = self.select_sparse_portfolio(
            eigenvectors,
            eigenvalues,
            returns.columns.tolist()
        )

        print(f"Selected {len(portfolio)} assets:")
        for symbol, weight in sorted(portfolio.items(), key=lambda x: x[^7_1], reverse=Tr
            print(f"  {symbol}: {weight:.4f}")

        return portfolio
```

Test it:

```
from app.services.portfolio_constructor import PortfolioConstructor

# Use features from previous step
constructor = PortfolioConstructor(sparsity_k=10)
portfolio = constructor.construct_portfolio(returns, var1, cov)

# Visualize
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 6))
plt.bar(portfolio.keys(), portfolio.values())
plt.title("Sparse Mean-Reverting Portfolio")
plt.xlabel("Asset")
plt.ylabel("Weight")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Why: This is the mathematical core—finding asset combinations that mean-revert.[7_8][7_7][^7_1]

WEEK 4: Trading Logic & Risk Management (Day 16-20)

Step 12: Trade Signal Engine

Create `app/services/trade_signal_engine.py`:

```
import pandas as pd
from typing import Dict, List
from enum import Enum

class Signal(Enum):
    BUY = "BUY"
    SELL = "SELL"
```

```
    HOLD = "HOLD"

class TradeSignalEngine:
    """Generates trading signals based on mean-reversion logic"""

    def __init__(self, deviation_threshold: float = 0.02):
        """
        Args:
            deviation_threshold: % deviation to trigger trade (e.g., 0.02 = 2%)
        """
        self.deviation_threshold = deviation_threshold

    def calculate_portfolio_value(self, portfolio: Dict[str, float],
                                  current_prices: Dict[str, float]) -> float:
        """
        Calculate current portfolio value.

        Args:
            portfolio: {symbol: weight}
            current_prices: {symbol: current_price}

        Returns:
            Total portfolio value
        """
        total_value = 0.0

        for symbol, weight in portfolio.items():
            if symbol in current_prices:
                total_value += weight * current_prices[symbol]

        return total_value

    def generate_signal(self, current_value: float,
                        target_value: float,
                        portfolio: Dict[str, float]) -> Dict:
        """
        Generate trading signal based on deviation from target.

        Mean-reversion logic:
        - If current > target + threshold → Portfolio overvalued → SELL
        - If current < target - threshold → Portfolio undervalued → BUY
        - Otherwise → HOLD

        Args:
            current_value: Current portfolio value
            target_value: Expected mean-reverting value
            portfolio: Portfolio weights

        Returns:
            Signal dictionary with action and details
        """
        deviation = (current_value - target_value) / target_value

        if deviation > self.deviation_threshold:
            signal = Signal.SELL
            message = f"Portfolio overvalued by {deviation*100:.2f}%"
```

```
        elif deviation < -self.deviation_threshold:
            signal = Signal.BUY
            message = f"Portfolio undervalued by {abs(deviation)*100:.2f}%"
        else:
            signal = Signal.HOLD
            message = f"Within threshold ({abs(deviation)*100:.2f}%)"

        return {
            'signal': signal,
            'deviation': deviation,
            'current_value': current_value,
            'target_value': target_value,
            'message': message,
            'portfolio': portfolio
        }
```

Step 13: Risk Manager

Create `app/services/risk_manager.py`:

```
from typing import Dict
from app.models.database import User

class RiskManager:
    """Enforces risk constraints on trading signals"""

    def __init__(self, user: User):
        """
        Args:
            user: User profile with risk parameters
        """
        self.user = user
        self.max_position_size = user.capital * 0.20  # 20% max per position
        self.drawdown_limit = user.drawdown_limit

    def check_position_size(self, signal: Dict) -> bool:
        """Verify position size doesn't exceed limits"""
        for symbol, weight in signal['portfolio'].items():
            position_value = weight * self.user.capital

            if position_value > self.max_position_size:
                return False, f"Position {symbol} exceeds 20% limit"

        return True, "Position sizes OK"

    def check_drawdown(self, current_equity: float, peak_equity: float) -> bool:
        """Check if current drawdown exceeds user limit"""
        drawdown = (peak_equity - current_equity) / peak_equity

        if drawdown > self.drawdown_limit:
            return False, f"Drawdown {drawdown*100:.1f}% exceeds limit {self.drawdown_lim

        return True, "Drawdown within limits"
```

```
    def kelly_criterion(self, win_rate: float, avg_win: float, avg_loss: float) -> float:
        """
        Calculate Kelly fraction for position sizing.

        Kelly = (p*b - q) / b
        where:
            p = win probability
            q = 1 - p
            b = avg_win / avg_loss

        Use half-Kelly for safety.
        """
        if avg_loss == 0:
            return 0.0

        b = avg_win / abs(avg_loss)
        kelly = (win_rate * b - (1 - win_rate)) / b

        # Use half-Kelly to reduce variance
        return max(0, kelly / 2)

    def validate_signal(self, signal: Dict, current_equity: float,
                        peak_equity: float) -> Dict:
        """
        Comprehensive risk check.

        Returns:
            {approved: bool, reason: str, adjusted_signal: Dict}
        """
        # Check position sizes
        size_ok, size_msg = self.check_position_size(signal)
        if not size_ok:
            return {'approved': False, 'reason': size_msg}

        # Check drawdown
        dd_ok, dd_msg = self.check_drawdown(current_equity, peak_equity)
        if not dd_ok:
            return {'approved': False, 'reason': dd_msg}

        return {
            'approved': True,
            'reason': 'All risk checks passed',
            'adjusted_signal': signal
        }
```

Test it:

```
from app.services.risk_manager import RiskManager
from app.models.database import SessionLocal, User

# Load test user
db = SessionLocal()
user = db.query(User).first()

# Initialize risk manager
```

```
risk_mgr = RiskManager(user)

# Create mock signal
test_signal = {
    'signal': 'BUY',
    'portfolio': {'AAPL': 0.15, 'MSFT': 0.20}
}

# Validate
result = risk_mgr.validate_signal(test_signal, current_equity=95000, peak_equity=100000)
print(result)
```

Why: Risk management prevents catastrophic losses—the #1 rule in trading.[7_3][7_1]

WEEK 5-6: Backtester & Performance Evaluation (Day 21-30)

Step 14: Backtester

Create `app/services/backtester.py`:

```
import pandas as pd
import numpy as np
from typing import Dict, List
from app.services.trade_signal_engine import TradeSignalEngine

class Backtester:
    """Simulates historical strategy performance"""

    def __init__(self, initial_capital: float = 100000):
        self.initial_capital = initial_capital
        self.commission_rate = 0.001  # 0.1%
        self.slippage_rate = 0.0005  # 0.05%

    def calculate_slippage(self, price: float, quantity: int,
                           avg_volume: float) -> float:
        """
        Estimate price impact (slippage).

        Model: slippage = base_spread + alpha * (order_size / volume)^gamma
        """
        base_spread = self.slippage_rate
        order_size = abs(quantity * price)
        impact = base_spread * (1 + 0.1 * (order_size / avg_volume) ** 0.5)

        return impact

    def simulate(self, portfolio: Dict[str, float],
                 price_data: Dict[str, pd.DataFrame],
                 signal_engine: TradeSignalEngine) -> pd.DataFrame:
        """
        Run backtest simulation.

        Args:
```

```
            portfolio: Portfolio weights
            price_data: Historical OHLCV data
            signal_engine: Trade signal generator

        Returns:
            DataFrame with daily portfolio values and trades
        """
        # Initialize
        capital = self.initial_capital
        equity_curve = []
        trades_log = []

        # Get all dates
        dates = price_data[list(portfolio.keys())[^7_0]].index

        # Track positions
        positions = {symbol: 0 for symbol in portfolio.keys()}

        for date in dates:
            # Get current prices
            current_prices = {}
            for symbol in portfolio.keys():
                if date in price_data[symbol].index:
                    current_prices[symbol] = price_data[symbol].loc[date, 'close']

            # Calculate portfolio value
            portfolio_value = sum(
                positions[symbol] * current_prices.get(symbol, 0)
                for symbol in portfolio.keys()
            )

            total_value = capital + portfolio_value

            # Generate signal (simplified: rebalance monthly)
            if date.day == 1:  # First day of month
                # Rebalance to target weights
                for symbol, target_weight in portfolio.items():
                    if symbol not in current_prices:
                        continue

                    target_value = total_value * target_weight
                    current_value = positions[symbol] * current_prices[symbol]

                    if abs(target_value - current_value) > 100:  # Only trade if meaningf
                        # Calculate trade
                        price = current_prices[symbol]
                        quantity = int((target_value - current_value) / price)

                        if quantity != 0:
                            # Apply slippage
                            slippage = self.calculate_slippage(price, quantity, 1e6)
                            fill_price = price * (1 + slippage if quantity > 0 else 1 - s

                            # Apply commission
                            commission = abs(quantity * fill_price) * self.commission_rat
```

```
                            # Execute trade
                            positions[symbol] += quantity
                            capital -= quantity * fill_price + commission

                            # Log trade
                            trades_log.append({
                                'date': date,
                                'symbol': symbol,
                                'side': 'BUY' if quantity > 0 else 'SELL',
                                'quantity': abs(quantity),
                                'price': fill_price,
                                'slippage': slippage,
                                'commission': commission
                            })

                # Record equity
                equity_curve.append({
                    'date': date,
                    'equity': total_value,
                    'cash': capital,
                    'positions_value': portfolio_value
                })

        # Convert to DataFrame
        equity_df = pd.DataFrame(equity_curve).set_index('date')
        trades_df = pd.DataFrame(trades_log)

        return equity_df, trades_df
```

Step 15: Performance Evaluator

Create `app/services/performance_evaluator.py`:

```
import pandas as pd
import numpy as np
from typing import Dict

class PerformanceEvaluator:
    """Calculates strategy performance metrics"""

    def calculate_returns(self, equity_curve: pd.DataFrame) -> pd.Series:
        """Calculate daily returns"""
        return equity_curve['equity'].pct_change().dropna()

    def sharpe_ratio(self, returns: pd.Series, risk_free_rate: float = 0.02) -> float:
        """
        Calculate annualized Sharpe ratio.

        Sharpe = (mean_return - risk_free_rate) / std_return * sqrt(252)
        """
        excess_returns = returns - risk_free_rate / 252
        return np.sqrt(252) * excess_returns.mean() / returns.std()

    def sortino_ratio(self, returns: pd.Series, risk_free_rate: float = 0.02) -> float:
```

```
        """
        Calculate Sortino ratio (penalizes only downside volatility).
        """
        excess_returns = returns - risk_free_rate / 252
        downside_returns = returns[returns < 0]
        downside_std = downside_returns.std()

        if downside_std == 0:
            return np.inf

        return np.sqrt(252) * excess_returns.mean() / downside_std

    def max_drawdown(self, equity_curve: pd.DataFrame) -> float:
        """
        Calculate maximum drawdown.

        Max DD = max((peak - trough) / peak)
        """
        equity = equity_curve['equity']
<span style="display:none">[^7_10][^7_11][^7_12][^7_13][^7_14][^7_15][^7_16][^7_17][^7_18

<div align="center">⁂</div>

[^7_1]: Quant-trading-assistant-Detailed-Design-Uml-Dataflow-with-Game-Theory.pdf
[^7_2]: https://www.youtube.com/watch?v=VirndPTeRaw
[^7_3]: https://realpython.com/get-started-with-fastapi/
[^7_4]: https://www.geeksforgeeks.org/python/fastapi-tutorial/
[^7_5]: https://dev.to/techprane/setting-up-postgresql-for-macos-users-step-by-step-instr
[^7_6]: https://www.devart.com/dbforge/postgresql/how-to-install-postgresql-on-macos/
[^7_7]: https://hudsonthames.org/sparse-mean-reverting-portfolio-selection/
[^7_8]: https://pmc.ncbi.nlm.nih.gov/articles/PMC2718382/
[^7_9]: https://www.youtube.com/watch?v=rvFsGRvj9jo
[^7_10]: https://pyimagesearch.com/2025/03/17/getting-started-with-python-and-fastapi-a-c
[^7_11]: https://code-b.dev/blog/build-apis-with-python
[^7_12]: https://www.youtube.com/watch?v=nWWPlEO0he8
[^7_13]: https://www.enterprisedb.com/postgres-tutorials/installation-postgresql-mac-os
[^7_14]: https://github.com/jaydu1/SparsePortfolio
[^7_15]: https://kinsta.com/blog/fastapi/
[^7_16]: https://www.youtube.com/watch?v=PShGF_udSpk
[^7_17]: https://docs.mosek.com/latest/pythonfusion/case-studies-portfolio.html
[^7_18]: https://pypi.org/project/fastapi/
[^7_19]: https://www.postgresql.org/download/macosx/
[^7_20]: https://github.com/dppalomar/sparseIndexTracking
[^7_21]: https://fastapi.tiangolo.com/tutorial/
```


1. Quant-trading-assistant-Detailed-Design-Uml-Dataflow-with-Game-Theory.pdf
2. https://www.linkedin.com/pulse/ultimate-guide-building-mvp-2025-f22-labs-vvbec
3. https://github.com/42Mo/fastapi-trading-app-example
4. https://alpaca.markets/learn/build-your-own-brokerage-with-fastapi-part-1
5. https://github.com/sam179/Sparse-Mean-Reverting-Portfolio-Selection
6. https://www.pnas.org/doi/10.1073/pnas.0904287106
7. https://resonanzcapital.com/insights/sparsity-in-porfolio-construction-doing-more-with-less

8. https://www.mnclgroup.com/a-practical-guide-to-advanced-trading-strategies
9. https://www.editiongroup.com/insights/minimum-viable-product-guide-for-founders
10. https://www.linkedin.com/pulse/how-build-minimum-viable-product-tips-tools-best-practices-yx6kc
11. https://wiserbrand.com/ai-mvp-development-how-to-build-smarter-minimum-viable-products-with-ai/
12. https://insights.daffodilsw.com/blog/how-to-build-an-mvp-that-attracts-funding
13. https://www.composer.trade/learn/quant-trading-strategies
14. https://blog.jetbrains.com/pycharm/2024/09/how-to-use-fastapi-for-machine-learning/
15. https://uxcam.com/blog/mvp-testing/
16. https://www.sciencedirect.com/science/article/abs/pii/S0378426619302614
17. https://www.youtube.com/watch?v=SVyuxZqbOrE
18. https://www.stanga.net/trends/top-40-mvp-development-companies-in-2025/
19. https://www.ecb.europa.eu/pub/pdf/scpwps/ecbwp936.pdf
20. https://www.c-sharpcorner.com/article/building-a-stock-market-api-with-fastapi-and-python/
21. https://www.f22labs.com/blogs/ultimate-guide-how-to-build-a-successful-mvp/
22. Quant-trading-assistant-Detailed-Design-Uml-Dataflow-with-Game-Theory.pdf
23. Quant-trading-assistant-Detailed-Design-Uml-Dataflow-with-Game-Theory.pdf
24. https://www.quantrocket.com/codeload/quant-finance-lectures/quant_finance_lectures/Lecture29-Universe-Selection.ipynb.html
25. https://www.quantconnect.com/docs/v2/writing-algorithms/algorithm-framework/universe-selection/key-concepts
26. https://www.luxalgo.com/blog/high-quality-investment-strategies-three-filters/
27. https://www.sciencedirect.com/science/article/abs/pii/S0957417422024538
28. https://www.theseus.fi/bitstream/10024/883847/2/Zhang_Kun_Mei_Yu_Hu_Yiming.pdf
29. https://stoxbox.in/mentorbox/marketopedia/technical-analysis/how-to-select-stocks
30. https://www.linkedin.com/pulse/ultimate-guide-building-mvp-2025-f22-labs-vvbec
31. https://strategyquant.com/doc/quantanalyzer/using-sectors-in-portfolio-master/
32. https://cssanalytics.wordpress.com/2014/06/04/momentum-strategies-and-universe-selection/
33. https://www.quantconnect.com/forum/discussion/9576/realtime-intraday-universe-selection/
34. https://www.quantrocket.com/codeload/sell-gap/sell_gap/Part2-Universe-Selection.ipynb.html
35. https://www.youtube.com/watch?v=rf7t5K1Bczc
36. https://www.ssga.com/library-content/pdfs/etf/us/sector-investing-a-powerful-portfolio-investing-tool.pdf
37. https://innoquantivity.com/2019/11/quantconnect-algorithm-framework-universe-selection/
38. https://www.fe.training/free-resources/portfolio-management/portfolio-construction/
39. https://fidelfolio.com/stock-selection-challenges-solved-by-quant-investing/
40. https://dl.acm.org/doi/10.1016/j.eswa.2022.119434
41. https://www.unpri.org/manager-selection/portfolio-construction-and-investment-decision-making-in-manager-selection/2733.article
42. https://www.reddit.com/r/stocks/comments/vily90/whats_your_portfolio_construction_of_stockssector/

43. https://www.youtube.com/watch?v=PGQVmE13ZwM
44. https://institutional.fidelity.com/advisors/insights/topics/portfolio-construction/the-case-for-using-a-sector-based-framework-in-equity-portfolio-construction
45. Quant-trading-assistant-Detailed-Design-Uml-Dataflow-with-Game-Theory.pdf
46. Quant-trading-assistant-Detailed-Design-Uml-Dataflow-with-Game-Theory.pdf