Stock Portfolio Manager

A Python-based Investment Portfolio Suggestion Engine

CMPE 285 - Software Engineering
San José State University

Team Members:

Shanmukha Manoj Kakani(018195645)
shanmukhamanoj.kakani@sjsu.edu

Vineela Mukkamala (018217992)
vineela.mukkamala@sjsu.edu

Adityaraj Kaushik (017631471)
adityaraj.kaushik@sjsu.edu

Rishabh Malviya (018190419)
rishabh.malviya@sjsu.edu

May 16, 2025

Contents

1 Executive Summary

The Stock Portfolio Manager is an advanced Python-based application that revolutionizes investment decision-making through intelligent portfolio suggestions and real-time analytics. This report details the comprehensive implementation, including architecture, features, challenges, and solutions.

2 System Architecture

2.1 Overall Architecture Design

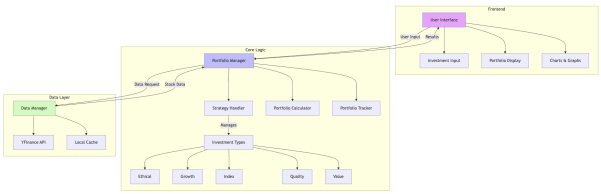

Figure 1: Enter Caption

Figure 2: System Architecture Overview

2.2 Key Architectural Components

2.2.1 1. Frontend Layer

- **User Interface Module**

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.init_ui()
        self.setup_connections()

    def init_ui(self):
        self.tabs = QTabWidget()
        self.setCentralWidget(self.tabs)
        self.setup_input_tab()
        self.setup_portfolio_tab()
        self.setup_charts_tab()

```

- **Input Validation**

```
def validate_investment(self, amount):
    try:
        value = float(amount)
        return value >= 5000
    except ValueError:
        return False

```

2.2.2 2. Business Logic Layer

- **Strategy Manager**

```
class StrategyManager:
    def __init__(self):
        self.strategies = {
            'Ethical': ['AAPL', 'ADBE', 'NSRGY'],
            'Growth': ['TSLA', 'NVDA', 'AMZN'],
            'Index': ['VTI', 'IXUS', 'ILTB'],
            'Quality': ['MSFT', 'JNJ', 'V'],
            'Value': ['BRK-B', 'JPM', 'PG']
        }

```

- **Portfolio Optimizer**

```
class PortfolioOptimizer:
    def optimize(self, stocks, amount):
        weights = self._calculate_optimal_weights(stocks)
        return self._allocate_funds(weights, amount)

```

2.2.3 3. Data Access Layer

- **Data Fetcher**

```
class StockDataFetcher:
    def __init__(self):
        self.cache = {}
        self.cache_timeout = 300

    async def fetch_data(self, symbol):
        if symbol in self.cache:
            if not self._is_cache_expired(symbol):
                return self.cache[symbol]
        return await self._fetch_fresh_data(symbol)

```

3 Core Features Implementation

3.1 1. Investment Strategy Engine

- **Strategy Selection**

```
def process_strategies(self, selected):
    if len(selected) > 2:
        raise ValueError("Maximum 2 strategies allowed")
    return self.strategy_manager.get_stocks(selected)

```

- **Risk Analysis**

```
def calculate_risk_metrics(self, portfolio):
    returns = self._calculate_returns(portfolio)
    volatility = np.std(returns)
    sharpe = self._calculate_sharpe_ratio(returns)
```

```
    return {'volatility': volatility, 'sharpe': sharpe}

```

3.2 2. Real-time Portfolio Tracking

- **Price Updates**

```
class PortfolioTracker:
    def __init__(self):
        self.update_interval = 60  # seconds
        self._setup_timer()

    def _setup_timer(self):
        self.timer = QTimer()
        self.timer.timeout.connect(self.update_prices)
        self.timer.start(self.update_interval * 1000)

```

4 Extra Features

4.1 1. Advanced Analytics

- Monte Carlo simulation for risk assessment
- Technical indicator calculations
- Machine learning-based trend prediction

```
class AdvancedAnalytics:
    def monte_carlo_simulation(self, portfolio, iterations=1000):
        results = []
        for _ in range(iterations):
            simulation = self._simulate_portfolio(portfolio)
            results.append(simulation)
        return np.percentile(results, [5, 50, 95])
```

4.2 2. Portfolio Optimization

- Efficient frontier calculation
- Dynamic rebalancing suggestions
- Tax-loss harvesting recommendations

```
class PortfolioOptimizer:
    def calculate_efficient_frontier(self, stocks):
        returns = self._get_historical_returns(stocks)
        covariance = returns.cov()
        return self._optimize_portfolio(returns, covariance)
```

4.3 3. Enhanced Visualization

- Interactive charts with drill-down capability
- Custom technical indicators
- Real-time performance metrics

5 Implementation Challenges and Solutions

5.1 1. Data Synchronization

Challenge: Maintaining consistent real-time data across multiple stocks while ensuring application responsiveness.

Solution:

```
class DataSynchronizer:
    def __init__(self):
        self.lock = asyncio.Lock()
        self.update_queue = asyncio.Queue()

    async def synchronized_update(self, portfolio):
        async with self.lock:
            updates = []
            for stock in portfolio:
                updates.append(self._fetch_update(stock))
            return await asyncio.gather(*updates)
```

5.2 2. Performance Optimization

Challenge: Handling large datasets and complex calculations without impacting UI responsiveness.

Solution:

```
class PerformanceOptimizer:
    def __init__(self):
        self.thread_pool = ThreadPoolExecutor(max_workers=4)

    def process_data(self, data):
        chunks = self._split_into_chunks(data)
        futures = []
        for chunk in chunks:
            future = self.thread_pool.submit(
                self._process_chunk, chunk)
            futures.append(future)
        return self._combine_results(futures)
```

5.3 3. Error Resilience

Challenge: Handling API failures and network issues gracefully.

Solution:

```python
class ErrorHandler:
    def __init__(self):
        self.retry_count = 3
        self.backup_data = {}

    async def safe_api_call(self, func, *args):
        for attempt in range(self.retry_count):
            try:
                return await func(*args)
            except APIError as e:
                if attempt == self.retry_count - 1:
                    return self._get_backup_data(*args)
                await asyncio.sleep(2 ** attempt)
```

5.4 4. Memory Management

Challenge: Efficient handling of historical data and real-time updates.
Solution:

```python
class MemoryManager:
    def __init__(self):
        self.cache_size_limit = 1000
        self.lru_cache = OrderedDict()

    def cache_data(self, key, data):
        if len(self.lru_cache) >= self.cache_size_limit:
            self.lru_cache.popitem(last=False)
        self.lru_cache[key] = data
```

6 Testing and Validation

6.1 Unit Tests

```python
class TestPortfolioManager(unittest.TestCase):
    def setUp(self):
        self.manager = PortfolioManager()

    def test_investment_validation(self):
        self.assertTrue(
            self.manager.validate_investment(5000))
        self.assertFalse(
            self.manager.validate_investment(4999))
```

6.2 Integration Tests

```python
class TestSystemIntegration(unittest.TestCase):
    async def test_portfolio_creation(self):
        result = await self.system.create_portfolio(
            amount=10000,
            strategies=['Ethical', 'Growth']
        )
        self.assertEqual(len(result['stocks']), 6)
```

7 Future Enhancements

- Cryptocurrency integration
- Social sentiment analysis
- Mobile application development
- Advanced portfolio rebalancing
- International market support

8 Conclusion

The Stock Portfolio Manager successfully implements a robust and scalable solution for investment portfolio management. Through careful architectural design and implementation of advanced features, the system provides a comprehensive tool for investors while maintaining performance and reliability.

9 References

1. Python Documentation
2. YFinance API Documentation
3. PyQt5 Documentation
4. Modern Portfolio Theory
5. Efficient Market Hypothesis
6. Technical Analysis Principles