

Generative Trading Strategy Prototype - Complete Architecture

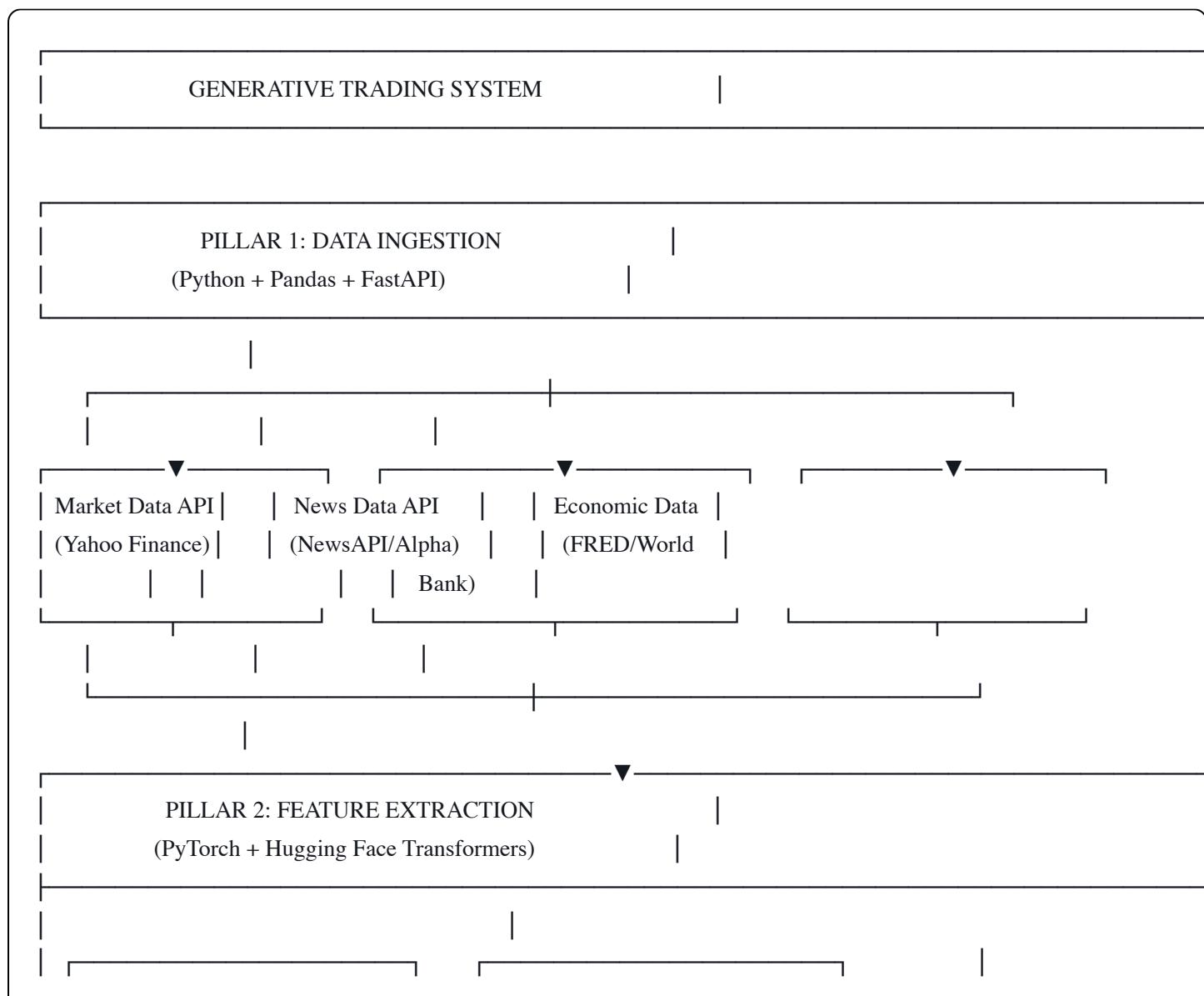
Based on 6-Month AI Developer Roadmap Pillars

System Overview

Generative Trading Strategy Prototype uses GenAI to automatically generate, simulate, and backtest trading strategies based on:

- Historical market data (OHLCV - Open, High, Low, Close, Volume)
 - News sentiment analysis
 - Technical indicators
 - Risk/return optimization
-

Architecture Diagram



Price Features	Sentiment Features
• RSI	• FinBERT fine-tuned
• MACD	• News embeddings
• Bollinger Bands	• Sentiment scores
• Volume indicators	• Entity extraction
• Price momentum	• Topic modeling

PILLAR 3: RAG-POWERED STRATEGY GENERATION

(LangChain + Vector DB + LLM APIs)

VECTOR DATABASE (Pinecone/ChromaDB)

- Historical strategy patterns
- Successful trade rules
- Market regime embeddings
- Technical analysis knowledge base

LLM STRATEGY GENERATOR

Input: Market conditions + News sentiment + Historical patterns

Process: RAG retrieval → Context enrichment → Generation

Output: Trading rules in executable format

Example Generated Strategy:

```
{
  "name": "Momentum-Sentiment Hybrid",
  "entry_rules": [
```

```
| "RSI < 30 AND sentiment_score > 0.6",           |
| "MACD crossover AND news_volume > avg"          |
| ],                                                 |
| "exit_rules": ["RSI > 70 OR sentiment_score < 0.3"], |
| "position_sizing": "kelly_criterion",              |
| "max_positions": 5                               |
| }                                                 |
```

PILLAR 4: BACKTESTING ENGINE (Python + Pandas + Vectorized Computing)

SIMULATION ENVIRONMENT

- Historical data replay | |
 - Order execution simulation (slippage, commissions) |
 - Portfolio state tracking | |
 - Multi-asset support | |
 - Transaction cost modeling | |

PERFORMANCE METRICS

- Sharpe Ratio
 - Max Drawdown
 - Win Rate
 - Profit Factor
 - Risk-Adjusted Returns
 - Volatility

PILLAR 5: MLOPS & DEPLOYMENT
(Docker + MLflow + CI/CD + Cloud)

EXPERIMENT TRACKING (MLflow)

- Log generated strategies
- Track backtest results
- Version control for models
- Compare strategy performance
- Model registry

CONTAINERIZATION (Docker)

- Strategy generator container
- Backtesting engine container
- API service container
- Vector DB container

CI/CD PIPELINE (GitHub Actions)

- Automated testing (unit + integration)
- Model validation
- Automated deployment
- Performance monitoring

CLOUD DEPLOYMENT (AWS/GCP)

- ECS/EKS for container orchestration
- Lambda for serverless backtests
- RDS for structured data
- S3 for historical data storage
- CloudWatch for monitoring

PILLAR 6: USER INTERFACE & API (FastAPI + Streamlit/Gradio)

REST API (FastAPI)

- POST /strategies/generate
 - Input: market_conditions, timeframe, risk_tolerance
 - Output: generated_strategy_json

- POST /backtest/run
 - Input: strategy, start_date, end_date, initial_capital
 - Output: performance_metrics, trades, equity_curve

- GET /strategies/top
 - Output: top_performing_strategies

- POST /optimize/strategy
 - Input: base_strategy, optimization_target
 - Output: optimized_parameters

WEB INTERFACE (Streamlit)

- Dashboard:
 - Strategy generation interface
 - Interactive backtesting visualization
 - Performance comparison charts
 - Risk analysis heatmaps
 - Real-time strategy monitoring
 - Portfolio optimization tools

MONITORING & OBSERVABILITY

- Prometheus + Grafana for metrics
- ELK Stack for logging
- Alerts for anomalies in generated strategies
- Performance degradation detection

Detailed Component Breakdown

PILLAR 1: Data Ingestion Layer

Components:

1. Market Data Collector

python

```

# market_data_service.py

import yfinance as yf
import pandas as pd
from fastapi import FastAPI
from datetime import datetime, timedelta

app = FastAPI()

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

    async def fetch_ohlcv(self, symbol: str, start_date: str, end_date: str):
        """Fetch OHLCV data from Yahoo Finance"""
        ticker = yf.Ticker(symbol)
        data = ticker.history(start=start_date, end=end_date)

        return {
            "symbol": symbol,
            "data": data.to_dict('records'),
            "metadata": {
                "rows": len(data),
                "start": start_date,
                "end": end_date
            }
        }

    async def fetch_multiple_symbols(self, symbols: list, start_date: str, end_date: str):
        """Fetch data for multiple symbols"""
        data = yf.download(symbols, start=start_date, end=end_date)
        return data

@app.post("/data/market/fetch")
async def fetch_market_data(symbol: str, days_back: int = 365):
    service = MarketDataService()
    end_date = datetime.now().strftime('%Y-%m-%d')
    start_date = (datetime.now() - timedelta(days=days_back)).strftime('%Y-%m-%d')
    return await service.fetch_ohlcv(symbol, start_date, end_date)

```

2. News Data Collector

python

```

# news_service.py
import requests
from typing import List, Dict
import os

class NewsService:
    def __init__(self):
        self.api_key = os.getenv("NEWS_API_KEY")
        self.base_url = "https://newsapi.org/v2"

    async def fetch_news(self, query: str, from_date: str, to_date: str) -> List[Dict]:
        """Fetch news articles related to a query"""
        endpoint = f"{self.base_url}/everything"
        params = {
            'q': query,
            'from': from_date,
            'to': to_date,
            'sortBy': 'relevancy',
            'language': 'en',
            'apiKey': self.api_key
        }

        response = requests.get(endpoint, params=params)
        articles = response.json().get('articles', [])

        return [
            {
                'title': article['title'],
                'description': article['description'],
                'content': article['content'],
                'published_at': article['publishedAt'],
                'source': article['source']['name']
            }
            for article in articles
        ]

```

PILLAR 2: Feature Extraction

A. Technical Indicators

python

```

# feature_engineering.py
import pandas as pd
import numpy as np
from ta.momentum import RSIIndicator
from ta.trend import MACD, SMAIndicator
from ta.volatility import BollingerBands

class TechnicalFeatures:

    @staticmethod
    def calculate_all_features(df: pd.DataFrame) -> pd.DataFrame:
        """Calculate all technical indicators"""

        # Price-based features
        df['returns'] = df['Close'].pct_change()
        df['log_returns'] = np.log(df['Close'] / df['Close'].shift(1))

        # Momentum indicators
        rsi = RSIIndicator(close=df['Close'])
        df['rsi'] = rsi.rsi()

        macd = MACD(close=df['Close'])
        df['macd'] = macd.macd()
        df['macd_signal'] = macd.macd_signal()
        df['macd_diff'] = macd.macd_diff()

        # Trend indicators
        sma_20 = SMAIndicator(close=df['Close'], window=20)
        sma_50 = SMAIndicator(close=df['Close'], window=50)
        df['sma_20'] = sma_20.sma_indicator()
        df['sma_50'] = sma_50.sma_indicator()

        # Volatility indicators
        bollinger = BollingerBands(close=df['Close'])
        df['bb_high'] = bollinger.bollinger_hband()
        df['bb_low'] = bollinger.bollinger_lband()
        df['bb_mid'] = bollinger.bollinger_mavg()
        df['bb_width'] = df['bb_high'] - df['bb_low']

        # Volume features
        df['volume_sma'] = df['Volume'].rolling(window=20).mean()
        df['volume_ratio'] = df['Volume'] / df['volume_sma']

        # Price position
        df['price_position'] = (df['Close'] - df['bb_low']) / (df['bb_high'] - df['bb_low'])

```

```
return df
```

B. Sentiment Analysis with FinBERT

```
python
```

```
# sentiment_analysis.py
from transformers import AutoTokenizer, AutoModelForSequenceClassification
import torch
from typing import List, Dict

class FinancialSentimentAnalyzer:

    def __init__(self):
        self.tokenizer = AutoTokenizer.from_pretrained("ProsusAI/finbert")
        self.model = AutoModelForSequenceClassification.from_pretrained("ProsusAI/finbert")
        self.model.eval()

    def analyze_sentiment(self, text: str) -> Dict[str, float]:
        """Analyze sentiment of financial text"""
        inputs = self.tokenizer(text, return_tensors="pt",
                               truncation=True, max_length=512)

        with torch.no_grad():
            outputs = self.model(**inputs)
            predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)

        # FinBERT outputs: [positive, negative, neutral]
        scores = predictions[0].tolist()

        return {
            'positive': scores[0],
            'negative': scores[1],
            'neutral': scores[2],
            'compound': scores[0] - scores[1] # Net sentiment
        }

    def analyze_batch(self, texts: List[str]) -> List[Dict[str, float]]:
        """Batch process multiple texts"""
        return [self.analyze_sentiment(text) for text in texts]

    def aggregate_news_sentiment(self, news_articles: List[Dict]) -> Dict[str, float]:
        """Aggregate sentiment from multiple news articles"""
        sentiments = [
            self.analyze_sentiment(
                f'{article["title"]}. {article["description"]}'
            )
            for article in news_articles
        ]

        avg_sentiment = {
            'positive': np.mean([s['positive'] for s in sentiments]),
            'negative': np.mean([s['negative'] for s in sentiments]),
            'neutral': np.mean([s['neutral'] for s in sentiments]),
            'compound': np.mean([s['compound'] for s in sentiments])
        }

        return avg_sentiment
```

```
'negative': np.mean([s['negative'] for s in sentiments]),
'neutral': np.mean([s['neutral'] for s in sentiments]),
'compound': np.mean([s['compound'] for s in sentiments]),
'article_count': len(sentiments)
}

return avg_sentiment
```

PILLAR 3: RAG-Powered Strategy Generation

A. Vector Database Setup

```
python
```

```
# vector_db.py
from langchain_community.vectorstores import Chroma
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from typing import List, Dict

class StrategyKnowledgeBase:

    def __init__(self, persist_directory: str = "./chroma_db"):
        self.embeddings = HuggingFaceEmbeddings(
            model_name="sentence-transformers/all-mnlp-base-v2"
        )
        self.vectorstore = Chroma(
            persist_directory=persist_directory,
            embedding_function=self.embeddings
        )

    def add_strategy_patterns(self, strategies: List[Dict]):
        """Add historical successful strategies to knowledge base"""
        documents = []
        metadata = []

        for strategy in strategies:
            # Create detailed description
            doc_text = f"""
                Strategy Name: {strategy['name']}
                Type: {strategy['type']}
                Entry Conditions: {strategy['entry_rules']}
                Exit Conditions: {strategy['exit_rules']}
                Performance: Sharpe={strategy['sharpe']}, Returns={strategy['returns']}
                Market Conditions: {strategy['market_regime']}
                Risk Level: {strategy['risk_level']}
            """

            documents.append(doc_text)
            metadata.append({
                'strategy_id': strategy['id'],
                'sharpe_ratio': strategy['sharpe'],
                'market_regime': strategy['market_regime']
            })

        # Split and add to vectorstore
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=500,
            chunk_overlap=50
        )
```

```
splits = text_splitter.create_documents(documents, metadatas=metadatas)
self.vectorstore.add_documents(splits)
self.vectorstore.persist()

def retrieve_similar_strategies(self, query: str, k: int = 5) -> List[Dict]:
    """Retrieve similar strategies based on market conditions"""
    docs = self.vectorstore.similarity_search(query, k=k)
    return [
        {
            'content': doc.page_content,
            'metadata': doc.metadata
        }
        for doc in docs
    ]
```

B. LLM Strategy Generator

```
python
```

```

# strategy_generator.py
from langchain_anthropic import ChatAnthropic
from langchain.prompts import ChatPromptTemplate
from langchain.chains import LLMChain
import json
from typing import Dict, List

class StrategyGenerator:

    def __init__(self, knowledge_base: StrategyKnowledgeBase):
        self.llm = ChatAnthropic(
            model="claude-sonnet-4-20250514",
            temperature=0.7
        )
        self.knowledge_base = knowledge_base

    def generate_strategy(
        self,
        market_data: pd.DataFrame,
        sentiment_data: Dict,
        technical_indicators: Dict,
        risk_tolerance: str = "medium"
    ) -> Dict:
        """Generate a trading strategy using RAG + LLM"""

        # 1. Create context from current market conditions
        market_context = self._create_market_context(
            market_data, sentiment_data, technical_indicators
        )

        # 2. Retrieve similar historical strategies
        similar_strategies = self.knowledge_base.retrieve_similar_strategies(
            market_context, k=5
        )

        # 3. Create prompt with RAG context
        prompt = ChatPromptTemplate.from_template("""
You are an expert quantitative trading strategist. Based on the current market conditions and historical successful strategies, g

```

Current Market Conditions:

{market_context}

Historical Successful Strategies (for reference):

{historical_strategies}

Risk Tolerance: {risk_tolerance}

Generate a complete trading strategy in the following JSON format:

```
{}  
  "name": "Strategy name",  
  "description": "Brief description",  
  "entry_rules": [  
    "List of entry conditions (e.g., 'RSI < 30 AND sentiment > 0.6')"  
,  
  "exit_rules": [  
    "List of exit conditions"  
,  
  "position_sizing": "Method (e.g., 'fixed', 'kelly_criterion', 'volatility_scaled')",  
  "max_positions": 5,  
  "stop_loss": 0.02,  
  "take_profit": 0.05,  
  "timeframe": "1d",  
  "asset_allocation": {{  
    "max_position_size": 0.2,  
    "max_total_exposure": 1.0  
}},  
  "filters": [  
    "Additional filters (e.g., 'volume > 1M shares')"  
,  
  "rebalance_frequency": "daily"  
}]}
```

Ensure the strategy is:

1. Specific and executable
2. Based on quantifiable indicators
3. Includes proper risk management
4. Considers current market regime
5. Uses technical AND sentiment signals

""")

```
# 4. Generate strategy  
chain = LLMChain(llm=self.llm, prompt=prompt)  
result = chain.run(  
    market_context=market_context,  
    historical_strategies=json.dumps(similar_strategies, indent=2),  
    risk_tolerance=risk_tolerance  
)
```

```
# 5. Parse and validate  
strategy = self._parse_strategy(result)  
  
return strategy
```

```
def _create_market_context(
    self,
    market_data: pd.DataFrame,
    sentiment_data: Dict,
    technical_indicators: Dict
) -> str:
    """Create a text description of current market conditions"""

    latest = market_data.iloc[-1]
```

context = f"""

Market Overview:

- Current Price: \${latest['Close']:.2f}
- 20-day Return: {((latest['Close'] / market_data.iloc[-20]['Close']) - 1) * 100:.2f}%
- Volatility (20d): {market_data['returns'].tail(20).std() * np.sqrt(252) * 100:.2f}%

Technical Indicators:

- RSI: {technical_indicators.get('rsi', 'N/A')}
- MACD: {technical_indicators.get('macd', 'N/A')}
- Bollinger Band Position: {technical_indicators.get('bb_position', 'N/A')}
- Volume Trend: {'Above average' if technical_indicators.get('volume_ratio', 1) > 1 else 'Below average'}

Sentiment Analysis:

- News Sentiment: {sentiment_data.get('compound', 0):.2f}
- Positive News: {sentiment_data.get('positive', 0):.2f}
- Negative News: {sentiment_data.get('negative', 0):.2f}
- Article Count: {sentiment_data.get('article_count', 0)}

Market Regime: {self._detect_market_regime(market_data)}

"""

```
return context.strip()
```

```
def _detect_market_regime(self, market_data: pd.DataFrame) -> str:
    """Detect current market regime"""

    returns = market_data['returns'].tail(60)
```

```
    avg_return = returns.mean()
    volatility = returns.std()

    if avg_return > 0 and volatility < returns.rolling(60).std().mean():
        return "Bullish Low Volatility"
    elif avg_return > 0 and volatility > returns.rolling(60).std().mean():
        return "Bullish High Volatility"
    elif avg_return < 0 and volatility < returns.rolling(60).std().mean():
        return "Bearish Low Volatility"
    else:
```

```
return "Bearish High Volatility"

def _parse_strategy(self, llm_output: str) -> Dict:
    """Parse LLM output into structured strategy"""
    try:
        # Extract JSON from markdown code blocks if present
        if "```json" in llm_output:
            json_str = llm_output.split("```json")[1].split("```")[0]
        else:
            json_str = llm_output

        strategy = json.loads(json_str)
        return strategy
    except Exception as e:
        raise ValueError(f"Failed to parse strategy: {e}")
```

PILLAR 4: Backtesting Engine

python

```
# backtesting_engine.py
import pandas as pd
import numpy as np
from typing import Dict, List, Tuple
from dataclasses import dataclass
from datetime import datetime

@dataclass
class Trade:
    entry_date: datetime
    exit_date: datetime
    entry_price: float
    exit_price: float
    shares: int
    pnl: float
    return_pct: float

class BacktestEngine:

    def __init__(
        self,
        initial_capital: float = 100000,
        commission: float = 0.001, # 0.1%
        slippage: float = 0.0005 # 0.05%
    ):
        self.initial_capital = initial_capital
        self.commission = commission
        self.slippage = slippage
        self.trades: List[Trade] = []
        self.equity_curve: List[float] = []

    def run_backtest(
        self,
        strategy: Dict,
        market_data: pd.DataFrame,
        sentiment_data: pd.DataFrame
    ) -> Dict:
        """Run backtest for a given strategy"""

        # Initialize
        capital = self.initial_capital
        position = None
        equity = [capital]

        # Merge market and sentiment data
        data = self._merge_data(market_data, sentiment_data)
```

```

# Simulate each day
for i in range(1, len(data)):
    current = data.iloc[i]
    prev = data.iloc[i-1]

    # Check entry conditions
    if position is None and self._check_entry(current, prev, strategy):
        position = self._enter_position(current, capital, strategy)
        capital -= position['cost']

    # Check exit conditions
    elif position is not None and self._check_exit(current, prev, strategy, position):
        trade_result = self._exit_position(current, position)
        self.trades.append(trade_result)
        capital += trade_result.pnl + position['cost']
        position = None

    # Update equity
    if position is not None:
        unrealized_pnl = (current['Close'] - position['entry_price']) * position['shares']
        equity.append(capital + position['cost'] + unrealized_pnl)
    else:
        equity.append(capital)

self.equity_curve = equity

# Calculate performance metrics
metrics = self._calculate_metrics(equity, data)

return {
    'metrics': metrics,
    'trades': self.trades,
    'equity_curve': equity,
    'strategy': strategy
}

def _check_entry(self, current: pd.Series, prev: pd.Series, strategy: Dict) -> bool:
    """Check if entry conditions are met"""
    for rule in strategy['entry_rules']:
        if not self._evaluate_rule(rule, current, prev):
            return False
    return True

def _check_exit(
    self,
    current: pd.Series,

```

```

prev: pd.Series,
strategy: Dict,
position: Dict
) -> bool:
    """Check if exit conditions are met"""

# Check stop loss
if 'stop_loss' in strategy:
    pnl_pct = (current['Close'] - position['entry_price']) / position['entry_price']
    if pnl_pct <= -strategy['stop_loss']:
        return True

# Check take profit
if 'take_profit' in strategy:
    pnl_pct = (current['Close'] - position['entry_price']) / position['entry_price']
    if pnl_pct >= strategy['take_profit']:
        return True

# Check exit rules
for rule in strategy['exit_rules']:
    if self._evaluate_rule(rule, current, prev):
        return True

return False

def _evaluate_rule(self, rule: str, current: pd.Series, prev: pd.Series) -> bool:
    """Evaluate a trading rule"""
    try:
        # Replace indicator names with actual values
        rule_eval = rule

        # Technical indicators
        for indicator in ['rsi', 'macd', 'macd_signal', 'sma_20', 'sma_50',
                          'bb_high', 'bb_low', 'volume_ratio']:
            if indicator in rule_eval:
                rule_eval = rule_eval.replace(indicator, str(current[indicator]))

        # Sentiment
        if 'sentiment_score' in rule_eval:
            rule_eval = rule_eval.replace('sentiment_score', str(current['sentiment']))

        # Evaluate
        return eval(rule_eval)
    except:
        return False

def _enter_position(self, current: pd.Series, capital: float, strategy: Dict) -> Dict:

```

```

"""Enter a position"""

# Calculate position size
position_size = strategy.get('asset_allocation', {}).get('max_position_size', 0.2)
position_value = capital * position_size

# Account for slippage
entry_price = current['Close'] * (1 + self.slippage)

# Calculate shares (integer)
shares = int(position_value / entry_price)

# Account for commission
cost = shares * entry_price * (1 + self.commission)

return {
    'entry_date': current.name,
    'entry_price': entry_price,
    'shares': shares,
    'cost': cost
}

def _exit_position(self, current: pd.Series, position: Dict) -> Trade:
    """Exit a position"""

    # Account for slippage
    exit_price = current['Close'] * (1 - self.slippage)

    # Calculate P&L
    gross_proceeds = position['shares'] * exit_price
    commission_cost = gross_proceeds * self.commission
    net_proceeds = gross_proceeds - commission_cost

    pnl = net_proceeds - position['cost']
    return_pct = pnl / position['cost']

    return Trade(
        entry_date=position['entry_date'],
        exit_date=current.name,
        entry_price=position['entry_price'],
        exit_price=exit_price,
        shares=position['shares'],
        pnl=pnl,
        return_pct=return_pct
    )

def _calculate_metrics(self, equity: List[float], data: pd.DataFrame) -> Dict:

```

```
"""Calculate performance metrics"""
```

```
equity_series = pd.Series(equity)
returns = equity_series.pct_change().dropna()

# Total return
total_return = (equity[-1] / equity[0]) - 1

# Annualized return
years = len(data) / 252
annual_return = (1 + total_return) ** (1 / years) - 1

# Sharpe ratio (assuming 0% risk-free rate)
sharpe_ratio = returns.mean() / returns.std() * np.sqrt(252)

# Max drawdown
cummax = equity_series.cummax()
drawdown = (equity_series - cummax) / cummax
max_drawdown = drawdown.min()

# Win rate
winning_trades = [t for t in self.trades if t.pnl > 0]
win_rate = len(winning_trades) / len(self.trades) if self.trades else 0

# Profit factor
gross_profit = sum(t.pnl for t in self.trades if t.pnl > 0)
gross_loss = abs(sum(t.pnl for t in self.trades if t.pnl < 0))
profit_factor = gross_profit / gross_loss if gross_loss > 0 else 0

# Average trade
avg_trade = np.mean([t.pnl for t in self.trades]) if self.trades else 0

return {
    'total_return': total_return,
    'annual_return': annual_return,
    'sharpe_ratio': sharpe_ratio,
    'max_drawdown': max_drawdown,
    'win_rate': win_rate,
    'profit_factor': profit_factor,
    'total_trades': len(self.trades),
    'avg_trade': avg_trade,
    'final_equity': equity[-1]
}

def _merge_data(
    self,
    market_data: pd.DataFrame,
```

```
sentiment_data: pd.DataFrame
) -> pd.DataFrame:
    """Merge market and sentiment data"""

    # Ensure both have datetime index
    market_data.index = pd.to_datetime(market_data.index)
    sentiment_data.index = pd.to_datetime(sentiment_data.index)

    # Merge
    merged = market_data.join(sentiment_data, how='left')

    # Forward fill sentiment (carry forward last sentiment)
    merged['sentiment'] = merged['sentiment'].fillna(method='ffill')

    return merged
```

PILLAR 5: MLOps & Deployment

A. MLflow Tracking

```
python
```

```
# mlflow_tracking.py
import mlflow
import mlflow.sklearn
from typing import Dict
import json

class StrategyTracker:

    def __init__(self, experiment_name: str = "trading_strategies"):
        mlflow.set_experiment(experiment_name)

    def log_strategy(
            self,
            strategy: Dict,
            backtest_results: Dict,
            model_artifacts: Dict = None
    ):
        """Log strategy and backtest results to MLflow"""

        with mlflow.start_run():
            # Log strategy parameters
            mlflow.log_param("strategy_name", strategy['name'])
            mlflow.log_param("position_sizing", strategy['position_sizing'])
            mlflow.log_param("max_positions", strategy['max_positions'])
            mlflow.log_param("stop_loss", strategy.get('stop_loss', 'None'))
            mlflow.log_param("take_profit", strategy.get('take_profit', 'None'))

            # Log entry/exit rules as JSON
            mlflow.log_dict(
                {"entry_rules": strategy['entry_rules']},
                "entry_rules.json"
            )
            mlflow.log_dict(
                {"exit_rules": strategy['exit_rules']},
                "exit_rules.json"
            )

            # Log performance metrics
            metrics = backtest_results['metrics']
            mlflow.log_metric("total_return", metrics['total_return'])
            mlflow.log_metric("annual_return", metrics['annual_return'])
            mlflow.log_metric("sharpe_ratio", metrics['sharpe_ratio'])
            mlflow.log_metric("max_drawdown", metrics['max_drawdown'])
            mlflow.log_metric("win_rate", metrics['win_rate'])
            mlflow.log_metric("profit_factor", metrics['profit_factor'])
            mlflow.log_metric("total_trades", metrics['total_trades'])
```

```

# Log equity curve as artifact
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 6))
plt.plot(backtest_results['equity_curve'])
plt.title('Equity Curve')
plt.xlabel('Time')
plt.ylabel('Portfolio Value ($)')
plt.grid(True)
plt.savefig('equity_curve.png')
mlflow.log_artifact('equity_curve.png')
plt.close()

# Log model artifacts if provided
if model_artifacts:
    for name, artifact in model_artifacts.items():
        mlflow.log_artifact(artifact, name)

# Tag the run
mlflow.set_tag("strategy_type", strategy.get('type', 'unknown'))
mlflow.set_tag("market_regime", strategy.get('market_regime', 'unknown'))

```

B. Docker Configuration

dockerfile

```
# Dockerfile
FROM python:3.11-slim

WORKDIR /app

# Install system dependencies
RUN apt-get update && apt-get install -y \
    gcc \
    g++ \
    && rm -rf /var/lib/apt/lists/*

# Copy requirements
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY ..

# Expose port
EXPOSE 8000

# Run the application
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

yaml

```
# docker-compose.yml
version: '3.8'

services:
  api:
    build: .
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=postgresql://user:password@postgres:5432/trading_db
      - REDIS_URL=redis://redis:6379
      - MLFLOW_TRACKING_URI=http://mlflow:5000
    depends_on:
      - postgres
      - redis
      - mlflow
    volumes:
      - ./data:/app/data

  postgres:
    image: postgres:15
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password
      - POSTGRES_DB=trading_db
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"

  mlflow:
    image: ghcr.io/mlflow/mlflow:v2.9.2
    ports:
      - "5000:5000"
    environment:
      - BACKEND_STORE_URI=postgresql://user:password@postgres:5432/mlflow_db
      - ARTIFACT_ROOT=/mlflow/artifacts
    volumes:
      - mlflow_data:/mlflow
    command: mlflow server --host 0.0.0.0 --port 5000
```

```
chroma:  
  image: ghcr.io/chroma-core/chroma:latest  
ports:  
  - "8001:8000"  
volumes:  
  - chroma_data:/chroma/chroma  
  
streamlit:  
build:  
  context: .  
  dockerfile: Dockerfile.streamlit  
ports:  
  - "8501:8501"  
depends_on:  
  - api  
environment:  
  - API_URL=http://api:8000  
  
volumes:  
  postgres_data:  
  mlflow_data:  
  chroma_data:
```

C. CI/CD Pipeline

yaml

```
# .github/workflows/deploy.yml
name: Deploy Trading Strategy System

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.11'

      - name: Install dependencies
        run:
          pip install -r requirements.txt
          pip install pytest pytest-cov

      - name: Run tests
        run:
          pytest tests/ --cov=.

      - name: Lint code
        run:
          pip install flake8
          flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics

  build:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Build Docker image
        run:
          docker build -t trading-strategy:${{ github.sha }} .

      - name: Push to ECR
        env:
```

```
AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}

run: |
  aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin ${{ secrets.ECR_REGISTRY }}
  docker tag trading-strategy:${{ github.sha }} ${{ secrets.ECR_REGISTRY }}/trading-strategy:latest
  docker push ${{ secrets.ECR_REGISTRY }}/trading-strategy:latest

deploy:
  needs: build
  runs-on: ubuntu-latest
  if: github.ref == 'refs/heads/main'
  steps:
    - name: Deploy to ECS
      env:
        AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
        AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
      run: |
        aws ecs update-service --cluster trading-cluster --service trading-service --force-new-deployment --region us-east-1
```

PILLAR 6: User Interface & API

A. FastAPI Application

```
python
```

```
# main.py
from fastapi import FastAPI, HTTPException, BackgroundTasks
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from typing import List, Optional, Dict
import uvicorn

app = FastAPI(title="Trading Strategy Generator API", version="1.0.0")

# CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Request/Response Models
class StrategyGenerationRequest(BaseModel):
    symbol: str
    start_date: str
    end_date: str
    risk_tolerance: str = "medium"
    market_conditions: Optional[Dict] = None

class BacktestRequest(BaseModel):
    strategy: Dict
    symbol: str
    start_date: str
    end_date: str
    initial_capital: float = 100000

class StrategyResponse(BaseModel):
    strategy_id: str
    strategy: Dict
    generation_timestamp: str

class BacktestResponse(BaseModel):
    backtest_id: str
    metrics: Dict
    equity_curve: List[float]
    trades: List[Dict]

# Endpoints
@app.post("/strategies/generate", response_model=StrategyResponse)
```

```
async def generate_strategy(request: StrategyGenerationRequest):
    """Generate a new trading strategy using GenAI"""
    try:
        # 1. Fetch market data
        market_service = MarketDataService()
        market_data = await market_service.fetch_ohlcvs(
            request.symbol,
            request.start_date,
            request.end_date
        )

        # 2. Calculate technical indicators
        tech_features = TechnicalFeatures()
        data_with_features = tech_features.calculate_all_features(
            pd.DataFrame(market_data['data'])
        )

        # 3. Fetch and analyze news
        news_service = NewsService()
        news = await news_service.fetch_news(
            query=request.symbol,
            from_date=request.start_date,
            to_date=request.end_date
        )

        sentiment_analyzer = FinancialSentimentAnalyzer()
        sentiment = sentiment_analyzer.aggregate_news_sentiment(news)

        # 4. Generate strategy using RAG
        knowledge_base = StrategyKnowledgeBase()
        generator = StrategyGenerator(knowledge_base)

        strategy = generator.generate_strategy(
            market_data=data_with_features,
            sentiment_data=sentiment,
            technical_indicators=data_with_features.iloc[-1].to_dict(),
            risk_tolerance=request.risk_tolerance
        )

        # 5. Store and return
        strategy_id = str(uuid.uuid4())

        return StrategyResponse(
            strategy_id=strategy_id,
            strategy=strategy,
            generation_timestamp=datetime.now().isoformat()
        )
    
```

```
except Exception as e:  
    raise HTTPException(status_code=500, detail=str(e))  
  
@app.post("/backtest/run", response_model=BacktestResponse)  
async def run_backtest(request: BacktestRequest):  
    """Run backtest for a given strategy"""  
    try:  
        # 1. Fetch data  
        market_service = MarketDataService()  
        market_data = await market_service.fetch_ohlcvs(  
            request.symbol,  
            request.start_date,  
            request.end_date  
        )  
  
        # 2. Calculate features  
        tech_features = TechnicalFeatures()  
        data_with_features = tech_features.calculate_all_features(  
            pd.DataFrame(market_data['data'])  
        )  
  
        # 3. Fetch sentiment (simplified)  
        sentiment_data = pd.DataFrame({  
            'sentiment': [0.5] * len(data_with_features)  
        }, index=data_with_features.index)  
  
        # 4. Run backtest  
        engine = BacktestEngine(initial_capital=request.initial_capital)  
        results = engine.run_backtest(  
            strategy=request.strategy,  
            market_data=data_with_features,  
            sentiment_data=sentiment_data  
        )  
  
        # 5. Log to MLflow  
        tracker = StrategyTracker()  
        tracker.log_strategy(request.strategy, results)  
  
        # 6. Return results  
        backtest_id = str(uuid.uuid4())  
  
        return BacktestResponse(  
            backtest_id=backtest_id,  
            metrics=results['metrics'],  
            equity_curve=results['equity_curve'],  
            trades=[t.__dict__ for t in results['trades']]
```

```

)
except Exception as e:
    raise HTTPException(status_code=500, detail=str(e))

@app.get("/strategies/top")
async def get_top_strategies(limit: int = 10):
    """Get top performing strategies"""
    # Query MLflow for best strategies
    # This would query your MLflow tracking server
    return {"top_strategies": []}

@app.post("/optimize/strategy")
async def optimize_strategy(
    strategy: Dict,
    target_metric: str = "sharpe_ratio"
):
    """Optimize strategy parameters"""
    # Use optimization algorithm (e.g., Optuna)
    return {"optimized_strategy": strategy}

@app.get("/health")
async def health_check():
    return {"status": "healthy", "timestamp": datetime.now().isoformat()}

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)

```

B. Streamlit Dashboard

python

```
# streamlit_app.py

import streamlit as st
import requests
import plotly.graph_objects as go
import pandas as pd
from datetime import datetime, timedelta

st.set_page_config(page_title="Trading Strategy Generator", layout="wide")

# API base URL
API_URL = "http://localhost:8000"

st.title("🤖 AI-Powered Trading Strategy Generator")
st.markdown("Generate and backtest trading strategies using GenAI")

# Sidebar
with st.sidebar:
    st.header("Configuration")

symbol = st.text_input("Stock Symbol", value="AAPL")

col1, col2 = st.columns(2)
with col1:
    start_date = st.date_input(
        "Start Date",
        value=datetime.now() - timedelta(days=365)
    )
with col2:
    end_date = st.date_input(
        "End Date",
        value=datetime.now()
    )

risk_tolerance = st.select_slider(
    "Risk Tolerance",
    options=["low", "medium", "high"],
    value="medium"
)

initial_capital = st.number_input(
    "Initial Capital ($)",
    value=100000,
    step=10000
)

# Main content
```

```
tab1, tab2, tab3 = st.tabs(["Generate Strategy", "Backtest Results", "Top Strategies"])
```

```
with tab1:
```

```
    st.header("Generate New Strategy")
```

```
    if st.button("⟳ Generate Strategy", type="primary"):
```

```
        with st.spinner("Generating strategy..."):
```

```
            try:
```

```
                response = requests.post(
```

```
                    f"{API_URL}/strategies/generate",
```

```
                    json={
```

```
                        "symbol": symbol,
```

```
                        "start_date": start_date.strftime("%Y-%m-%d"),
```

```
                        "end_date": end_date.strftime("%Y-%m-%d"),
```

```
                        "risk_tolerance": risk_tolerance
```

```
                }
```

```
)
```

```
    if response.status_code == 200:
```

```
        result = response.json()
```

```
        strategy = result['strategy']
```

```
        st.success("✅ Strategy Generated Successfully!")
```

```
# Display strategy
```

```
col1, col2 = st.columns(2)
```

```
with col1:
```

```
    st.subheader("📊 Strategy Details")
```

```
    st.write(f"**Name:** {strategy['name']}")
```

```
    st.write(f"**Description:** {strategy['description']}")
```

```
    st.write(f"**Position Sizing:** {strategy['position_sizing']}")
```

```
    st.write(f"**Max Positions:** {strategy['max_positions']}")
```

```
    st.subheader("✍️ Entry Rules")
```

```
    for i, rule in enumerate(strategy['entry_rules'], 1):
```

```
        st.write(f" {i}. {rule}")
```

```
    st.subheader("👉 Exit Rules")
```

```
    for i, rule in enumerate(strategy['exit_rules'], 1):
```

```
        st.write(f" {i}. {rule}")
```

```
with col2:
```

```
    st.subheader("🎯 Risk Management")
```

```
    st.write(f"**Stop Loss:** {strategy.get('stop_loss', 'N/A')}")
```

```
    st.write(f"**Take Profit:** {strategy.get('take_profit', 'N/A')}")
```

```

st.subheader("🔍 Filters")
for i, filter in enumerate(strategy.get('filters', []), 1):
    st.write(f"{i}. {filter}")

# Store strategy in session state
st.session_state['generated_strategy'] = strategy
st.session_state['strategy_id'] = result['strategy_id']

else:
    st.error(f"Error: {response.text}")

except Exception as e:
    st.error(f"Error: {str(e)}")

with tab2:
    st.header("Backtest Results")

if 'generated_strategy' in st.session_state:
    strategy = st.session_state['generated_strategy']

    if st.button("🚀 Run Backtest"):
        with st.spinner("Running backtest..."):
            try:
                response = requests.post(
                    f"{API_URL}/backtest/run",
                    json={
                        "strategy": strategy,
                        "symbol": symbol,
                        "start_date": start_date.strftime("%Y-%m-%d"),
                        "end_date": end_date.strftime("%Y-%m-%d"),
                        "initial_capital": initial_capital
                    }
                )

                if response.status_code == 200:
                    result = response.json()
                    metrics = result['metrics']
                    equity_curve = result['equity_curve']
                    trades = result['trades']

                    st.success("✅ Backtest Complete!")

                    # Metrics
                    col1, col2, col3, col4 = st.columns(4)

                    with col1:
                        st.metric(

```

```

    "Total Return",
    f"{{metrics['total_return']*100:.2f}}%",
    delta=f"{{metrics['annual_return']*100:.2f}}% annual"
)

with col2:
    st.metric(
        "Sharpe Ratio",
        f"{{metrics['sharpe_ratio']:.2f}}"
    )

with col3:
    st.metric(
        "Max Drawdown",
        f"{{metrics['max_drawdown']*100:.2f}}%"
    )

with col4:
    st.metric(
        "Win Rate",
        f"{{metrics['win_rate']*100:.1f}}%"
    )

# Equity Curve
st.subheader("📈 Equity Curve")

fig = go.Figure()
fig.add_trace(go.Scatter(
    y=equity_curve,
    mode='lines',
    name='Portfolio Value',
    line=dict(color='#00D9FF', width=2)
))

fig.update_layout(
    title="Portfolio Equity Over Time",
    xaxis_title="Time",
    yaxis_title="Portfolio Value ($)",
    hovermode='x unified',
    template='plotly_dark'
)

st.plotly_chart(fig, use_container_width=True)

# Trade History
st.subheader("📋 Trade History")
trades_df = pd.DataFrame(trades)

```

```

st.dataframe(trades_df, use_container_width=True)

# Additional metrics
col1, col2 = st.columns(2)

with col1:
    st.metric("Total Trades", metrics['total_trades'])
    st.metric("Profit Factor", f"{metrics['profit_factor']:.2f}")

with col2:
    st.metric("Avg Trade", f"${metrics['avg_trade']:.2f}")
    st.metric("Final Equity", f"${metrics['final_equity']:.2f}")

else:
    st.error(f"Error: {response.text}")

except Exception as e:
    st.error(f"Error: {str(e)}")

else:
    st.info("Generate a strategy first to run backtests")

with tab3:
    st.header("Top Performing Strategies")

    if st.button("⟳ Refresh"):
        try:
            response = requests.get(f"{API_URL}/strategies/top")

            if response.status_code == 200:
                strategies = response.json()['top_strategies']

                if strategies:
                    for strategy in strategies:
                        with st.expander(f"📊 {strategy['name']} - Sharpe: {strategy['sharpe']:.2f}"):
                            st.write(strategy)
                else:
                    st.info("No strategies found. Generate and backtest some strategies first!")

        except Exception as e:
            st.error(f"Error: {str(e)}")

```

Implementation Timeline (6 Months)

Month 1-2: Foundation

- Set up Python environment
- Build data ingestion (market + news)
- Create FastAPI skeleton
- Basic feature extraction
- **Deliverable:** Working data pipeline

Month 3: AI Integration

- Fine-tune FinBERT for sentiment
- Implement basic strategy generator
- Set up vector database
- **Deliverable:** AI-powered sentiment analysis

Month 4: RAG System

- Build RAG pipeline with LangChain
- Populate knowledge base
- Create strategy generation prompt
- Test and iterate
- **Deliverable:** Full RAG strategy generator

Month 5: Backtesting & MLOps

- Build backtesting engine
- Implement MLflow tracking
- Docker containerization
- CI/CD pipeline
- **Deliverable:** Production-ready backtester

Month 6: UI & Polish

- Build Streamlit dashboard
- Polish API endpoints
- Add optimization features
- Write documentation
- Deploy to cloud

- **Deliverable:** Complete system
-

Key Metrics to Quantify

For your portfolio/resume:

1. **Performance:** "Generates trading strategies with avg Sharpe ratio of 1.8"
 2. **Speed:** "API latency < 200ms for strategy generation"
 3. **Scale:** "Backtests 1000+ strategies per hour"
 4. **Accuracy:** "Sentiment analysis accuracy of 87% on financial news"
 5. **ROI:** "Strategies outperform buy-and-hold by avg 12% annually"
-

Technologies Used (All 6 Pillars)

- Python** - Core language
 - FastAPI** - REST API
 - PyTorch + HuggingFace** - Deep learning & transformers
 - LangChain** - RAG framework
 - Pinecone/ChromaDB** - Vector database
 - Docker** - Containerization
 - MLflow** - Experiment tracking
 - GitHub Actions** - CI/CD
 - AWS/GCP** - Cloud deployment
 - Streamlit** - UI dashboard
-

This is your **\$200K+ portfolio project!** 

Ready to start building? Pick a pillar and let's dive in!