Complete Architecture & Design Document

AI-Powered Swing Trading Recommendation System

Version: 1.0
Date: November 18, 2025
Author: System Architecture Team
Document Type: Technical Design Specification

Executive Summary

This document specifies a production-grade, end-to-end swing trading recommendation system for S&P 500 stocks. The system leverages state-of-the-art deep learning (Temporal Fusion Transformers + Graph Neural Networks), statistical models (ARIMA/GARCH), LLM agents for text processing and policy enforcement, and classic technical analysis to generate high-probability trade recommendations daily.

Key Objectives:

- Analyze 500+ stocks every EOD (End of Day)
- Identify swing trading opportunities (5-day horizon)
- Output ranked list: ticker, buy/sell, target%, probability, stop%, rationale
- Combine numeric prediction (DL/ML) with textual context (LLM) and technical signals
- Achieve >65% win rate with 2:1 reward/risk ratio

Design Philosophy:

- **LLMs as context processors, not forecasters**: Use LLMs to structure text and enforce rules, not predict prices
- **Multi-scale signal fusion**: Combine temporal (TFT), relational (GNN), statistical (ARIMA), and textual (LLM) signals
- **Explainability by design**: Every recommendation includes human-readable rationale
- **Production-ready**: Cloud-native, scalable, monitored, with proper CI/CD

Table of Contents

1. System Overview

1.1 Architecture Diagram

```
┌─────────────────────────────────────────────────────────────────────┐
│                           DATA SOURCES                              │
│  Market Data (Polygon)  │  News/Sentiment  │  Fundamentals (Finnhub)│
└────────────┬────────────┴────────┬─────────┴──────────┬─────────────┘
             │                     │                    │
             v                     v                    v
┌─────────────────────────────────────────────────────────────────────┐
│                    DATA INGESTION LAYER (Prefect)                   │
│  ┌─────────────┐  ┌──────────────┐  ┌────────────────┐              │
│  │ Price Feed  │  │ News Feed    │  │ Earnings/Events │             │
│  └──────┬──────┘  └──────┬───────┘  └───────┬────────┘              │
└─────────┼────────────────┼──────────────────┼───────────────────────┘
          │                │                  │
          v                v                  v
┌─────────────────────────────────────────────────────────────────────┐
│                         STORAGE LAYER                               │
│  ┌──────────────┐  ┌─────────────────┐  ┌────────────────┐          │
│  │ S3 Data Lake │  │ TimescaleDB     │  │ Feature Store  │          │
│  │ (Raw OHLCV)  │  │ (Time-series)   │  │ (Feast)        │          │
│  └──────────────┘  └─────────────────┘  └────────────────┘          │
└─────────────────────────────────────────────────────────────────────┘
          │                                  │
          v                                  v
┌─────────────────────────────────────────────────────────────────────┐
│                    FEATURE ENGINEERING LAYER                        │
│  ┌────────────────┐  ┌─────────────────┐  ┌─────────────────┐       │
```

```
│  │   Technical      │  │ Cross-Sectional │  │  Text Features     │     │
│  │   Indicators     │  │  (Peer Ranks)   │  │  (LLM Agent)       │     │
│  └─────────┬─────────┘  └────────┬────────┘  └────────┬───────────┘     │
└────────────┼─────────────────────┼────────────────────┼─────────────────┘
             │                     │                    │
             └─────────────────────┬────────────────────┘
                                   │
                                   v
┌──────────────────────────────────────────────────────────────────────────┐
│                      PREDICTION LAYER                                    │
│                                                                          │
│  ┌────────────────────────────────────────────────────────────────────┐  │
│  │        TFT-GNN HYBRID MODEL (PRIMARY)                              │  │
│  │  ┌────────────────────┐     ┌────────────────────────┐             │  │
│  │  │ Temporal Fusion    │────▶│  Graph Neural Net      │             │  │
│  │  │  Transformer       │     │ (Dynamic Corr Graph)   │             │  │
│  │  └────────────────────┘     └────────────────────────┘             │  │
│  │             │                        │                             │  │
│  │             └─────────────┬──────────┘                             │  │
│  │                           v                                        │  │
│  │              ┌─────────────────────┐                               │  │
│  │              │  Fusion Layer       │                               │  │
│  │              │ [return, prob,      │                               │  │
│  │              │  uncertainty]       │                               │  │
│  │              └─────────────────────┘                               │  │
│  └────────────────────────────────────────────────────────────────────┘  │
│                                                                          │
│  ┌──────────────────────┐  ┌────────────────────┐  ┌────────────────┐    │
│  │ LightGBM Ranker      │  │ ARIMA/GARCH        │  │ Pattern        │    │
│  │ (Cross-sectional)    │  │ (Vol baseline)     │  │ Detector       │    │
│  └──────────┬───────────┘  └─────────┬──────────┘  └───────┬────────┘    │
└─────────────┼────────────────────────┼─────────────────────┼─────────────┘
              │                        │                     │
              └────────────────────────┬─────────────────────┘
                                       │
                                       v
┌──────────────────────────────────────────────────────────────────────────┐
│                      ENSEMBLE & RANKING LAYER                            │
│  ┌────────────────────────────────────────────────────────────────────┐  │
│  │  Priority Score Calculation                                        │  │
│  │  Base × (1 + tech_bonus + sentiment_bonus - penalties)             │  │
│  └────────────────────────────────────────────────────────────────────┘  │
└──────────────────────────────────────────────────────────────────────────┘
                                       │
                                       v
┌──────────────────────────────────────────────────────────────────────────┐
│                      LLM AGENT LAYER (Policy & Curation)                 │
│  ┌───────────────────┐  ┌─────────────────┐  ┌────────────────────┐      │
│  │ Policy Agent      │  │ Pattern Agent   │  │ Explainer Agent    │      │
│  │ (Trade Rules)     │  │ (Technical)     │  │ (Rationale)        │      │
│  └─────────┬─────────┘  └────────┬────────┘  └────────┬───────────┘      │
└────────────┼─────────────────────┼────────────────────┼──────────────────┘
             │                     │                    │
             └─────────────────────┬────────────────────┘
                                   │
                                   v
```

```
┌──────────────────────────────────────────────────────────────────────┐
│                       OUTPUT LAYER                                    │
│  ┌────────────────────────────────────────────────────┐              │
│  │ Final Recommendations (JSON/DB)                   │              │
│  │ ticker│side│target%│prob│stop%│priority│rationale  │              │
│  └────────────────────────────────────────────────────┘              │
└──────────────────────────────────────────────────────────────────────┘
            │
            v
┌──────────────────────────────────────────────────────────────────────┐
│                    USER INTERFACE                                     │
│  ┌──────────────┐  ┌─────────────────┐  ┌──────────────────┐         │
│  │ FastAPI      │  │ React Dashboard │  │ Daily Brief      │         │
│  │ REST API     │  │ (Trade List)    │  │ (Email/Slack)    │         │
│  └──────────────┘  └─────────────────┘  └──────────────────┘         │
└──────────────────────────────────────────────────────────────────────┘
```

1.2 Core Components

Component	Purpose	Technology
Data Ingestion	Fetch market, news, fundamental data	Prefect, Python
Storage	Raw data lake + time-series DB	S3, TimescaleDB, Feast
Feature Engineering	Technical indicators, cross-sectional, text features	pandas, ta-lib, LangChain
TFT-GNN Model	Primary forecasting engine	PyTorch, pytorch-forecasting, torch-geometric
Ensemble Layer	Combine multiple model outputs	LightGBM, statsmodels, sklearn
LLM Agents	Text processing, policy, explanation	LangChain, OpenAI API
Orchestration	Daily pipeline scheduling	Prefect Cloud
API & UI	Serve recommendations	FastAPI, React
Monitoring	Model drift, performance tracking	MLflow, Prometheus, CloudWatch

1.3 Key Design Decisions

Why TFT-GNN Hybrid?

- Recent research (2024-2025) shows TFT-GNN outperforms standalone models
- TFT captures temporal dependencies with attention
- GNN learns dynamic inter-stock relationships (correlations, sector effects)
- SMAPE as low as 0.0022 on individual stocks with proper features

Why LLMs as Context Processors?

- LLMs are poor at numeric forecasting (hallucination, lack of precision)
- LLMs excel at: text summarization, rule enforcement, explanation generation

- Use LLMs to convert unstructured text → structured features for numeric models

Why Multi-Model Ensemble?

- No single model captures all market regimes (trending, mean-reverting, volatile)
- TFT-GNN: temporal + relational patterns
- LightGBM: cross-sectional ranking
- ARIMA/GARCH: volatility baseline + regime detection
- Ensemble reduces overfitting, improves robustness

2. Data Infrastructure

2.1 Data Sources

Market Data (Primary)

Data Type	Provider	Frequency	Cost	API
OHLCV	Polygon.io	Real-time + EOD	$199/mo (Stocks Starter)	REST + WebSocket
Alternative	Tiingo	EOD	$10/mo (Starter)	REST
Backup	Alpha Vantage	EOD	Free (500 calls/day)	REST

Coverage:

- All S&P 500 constituents (~500 stocks)
- Sector ETFs (XLK, XLF, XLE, XLV, XLY, XLP, XLI, XLB, XLRE, XLU, XLC)
- Market indexes (SPY, QQQ, DIA, IWM)
- VIX (volatility index)

News & Sentiment

Data Type	Provider	Frequency	Cost
Financial News	Benzinga News API	Real-time	$150/mo
Alternative	Finnhub News	Real-time	$80/mo
Social Sentiment	Twitter API (if needed)	Real-time	$100/mo
Analyst Ratings	Finnhub Recommendations	Daily	Included in plan

Fundamentals & Events

Data Type	Provider	API
Earnings Calendar	Finnhub	`/calendar/earnings`
Dividend Calendar	Finnhub	`/calendar/dividend`
Market Cap, Sector	Polygon	`/v3/reference/tickers`
Insider Trading	Finnhub	`/stock/insider-transactions`
Options Flow	Unusual Whales (optional)	REST API

Macroeconomic Data

Data Type	Source	API
Interest Rates	FRED	`fredapi` Python library
Treasury Yields	FRED	`/series/DGS10`, `/series/DGS2`
Dollar Index (DXY)	Polygon	Forex API
Crude Oil (CL)	Polygon	Commodities API

2.2 Storage Architecture

S3 Data Lake Structure

```
s3://swing-trading-system/
├── raw/
│   ├── prices/
│   │   ├── 2025/11/18/
│   │   │   ├── AAPL.parquet
│   │   │   ├── MSFT.parquet
│   │   │   └── ...
│   ├── news/
│   │   ├── 2025/11/18/
│   │   │   ├── headlines.jsonl
│   │   │   └── analyst_changes.jsonl
│   └── fundamentals/
│       └── earnings_calendar.parquet
├── processed/
│   ├── features/
│   │   ├── technical_indicators.parquet
│   │   ├── cross_sectional.parquet
│   │   └── text_features.parquet
│   └── graphs/
│       └── correlation_matrices/
│           └── 2025-11-18.npz
└── models/
    ├── tft_gnn_v1.2.pth
```

```
    ├── lgbm_ranker_v1.1.pkl
    └── metadata.json
```

TimescaleDB Schema

```
-- Core price table (hypertable on time)
CREATE TABLE prices (
    time TIMESTAMPTZ NOT NULL,
    ticker VARCHAR(10) NOT NULL,
    open NUMERIC,
    high NUMERIC,
    low NUMERIC,
    close NUMERIC,
    volume BIGINT,
    vwap NUMERIC,
    PRIMARY KEY (time, ticker)
);

SELECT create_hypertable('prices', 'time');

-- Feature table
CREATE TABLE features (
    time TIMESTAMPTZ NOT NULL,
    ticker VARCHAR(10) NOT NULL,
    -- Technical
    rsi_14 NUMERIC,
    macd NUMERIC,
    macd_signal NUMERIC,
    bbands_upper NUMERIC,
    bbands_lower NUMERIC,
    atr_14 NUMERIC,
    volume_z_score NUMERIC,
    -- Cross-sectional
    return_rank_5d INTEGER,
    return_rank_20d INTEGER,
    sector_relative_strength NUMERIC,
    correlation_to_spy NUMERIC,
    -- Text (from LLM)
    sentiment_score NUMERIC,
    news_intensity VARCHAR(20),
    event_flag_earnings BOOLEAN,
    -- Pattern flags
    breakout_52w BOOLEAN,
    pattern_confidence NUMERIC,
    pattern_type VARCHAR(50),
    PRIMARY KEY (time, ticker)
);

SELECT create_hypertable('features', 'time');

-- Predictions table
CREATE TABLE predictions (
    time TIMESTAMPTZ NOT NULL,
    ticker VARCHAR(10) NOT NULL,
    model_version VARCHAR(20),
```

```
    expected_return NUMERIC,
    hit_prob_long NUMERIC,
    hit_prob_short NUMERIC,
    volatility_forecast NUMERIC,
    quantile_10 NUMERIC,
    quantile_50 NUMERIC,
    quantile_90 NUMERIC,
    priority_score NUMERIC,
    PRIMARY KEY (time, ticker, model_version)
);

SELECT create_hypertable('predictions', 'time');

-- Recommendations table (final output)
CREATE TABLE recommendations (
    date DATE NOT NULL,
    ticker VARCHAR(10) NOT NULL,
    side VARCHAR(10), -- 'buy' or 'sell'
    target_pct NUMERIC,
    stop_pct NUMERIC,
    probability NUMERIC,
    priority_score NUMERIC,
    position_size_pct NUMERIC,
    rationale TEXT[],
    model_version VARCHAR(20),
    created_at TIMESTAMPTZ DEFAULT NOW(),
    PRIMARY KEY (date, ticker)
);

-- Performance tracking
CREATE TABLE performance (
    recommendation_id INTEGER REFERENCES recommendations(id),
    entry_date DATE,
    exit_date DATE,
    entry_price NUMERIC,
    exit_price NUMERIC,
    actual_return NUMERIC,
    hit_target BOOLEAN,
    days_held INTEGER,
    exit_reason VARCHAR(50) -- 'target', 'stop', 'time', 'manual'
);
```

Feature Store (Feast)

```
# feature_repo/features.py
from feast import Entity, Feature, FeatureView, FileSource
from feast.types import Float32, Int32, String
from datetime import timedelta

# Entity definition
ticker = Entity(name="ticker", join_keys=["ticker"])

# Feature source (TimescaleDB via Parquet export)
technical_features_source = FileSource(
    path="s3://swing-trading-system/processed/features/technical_indicators.parquet",
```

```
    timestamp_field="time",
)

# Feature view
technical_features = FeatureView(
    name="technical_features",
    entities=[ticker],
    ttl=timedelta(days=7),
    schema=[
        Feature(name="rsi_14", dtype=Float32),
        Feature(name="macd", dtype=Float32),
        Feature(name="atr_14", dtype=Float32),
        Feature(name="volume_z_score", dtype=Float32),
        # ... more features
    ],
    source=technical_features_source,
)
```

2.3 Data Quality & Validation

Ingestion Validation:

```
def validate_price_data(df: pd.DataFrame) -> bool:
    """Validate OHLCV data quality."""
    checks = [
        # No nulls in critical columns
        df[['open', 'high', 'low', 'close', 'volume']].isnull().sum().sum() == 0,

        # High >= Low
        (df['high'] >= df['low']).all(),

        # OHLC within high/low bounds
        (df['open'] >= df['low']).all() and (df['open'] <= df['high']).all(),
        (df['close'] >= df['low']).all() and (df['close'] <= df['high']).all(),

        # Volume non-negative
        (df['volume'] >= 0).all(),

        # Price changes < 50% (circuit breaker check)
        (df['close'].pct_change().abs() < 0.5).all(),
    ]

    return all(checks)
```

Data Freshness Monitoring:

```
from datetime import datetime, timedelta

def check_data_freshness():
    """Alert if data is stale."""
    latest_timestamp = db.query("SELECT MAX(time) FROM prices").scalar()
```

```
    if datetime.now() - latest_timestamp > timedelta(hours=24):
        send_alert("Data pipeline stale! Latest data: {latest_timestamp}")
```

3. Feature Engineering

3.1 Technical Indicators

Price-based:

```
import talib as ta

def compute_technical_indicators(df: pd.DataFrame) -> pd.DataFrame:
    """Compute technical indicators using TA-Lib."""

    # Trend indicators
    df['sma_20'] = ta.SMA(df['close'], timeperiod=20)
    df['sma_50'] = ta.SMA(df['close'], timeperiod=50)
    df['ema_12'] = ta.EMA(df['close'], timeperiod=12)
    df['ema_26'] = ta.EMA(df['close'], timeperiod=26)

    # MACD
    df['macd'], df['macd_signal'], df['macd_hist'] = ta.MACD(
        df['close'], fastperiod=12, slowperiod=26, signalperiod=9
    )

    # RSI
    df['rsi_14'] = ta.RSI(df['close'], timeperiod=14)

    # Bollinger Bands
    df['bbands_upper'], df['bbands_middle'], df['bbands_lower'] = ta.BBANDS(
        df['close'], timeperiod=20, nbdevup=2, nbdevdn=2
    )
    df['bbands_pct'] = (df['close'] - df['bbands_lower']) / (df['bbands_upper'] - df['bba

    # ATR (volatility)
    df['atr_14'] = ta.ATR(df['high'], df['low'], df['close'], timeperiod=14)

    # Stochastic
    df['stoch_k'], df['stoch_d'] = ta.STOCH(
        df['high'], df['low'], df['close'],
        fastk_period=14, slowk_period=3, slowd_period=3
    )

    # ADX (trend strength)
    df['adx'] = ta.ADX(df['high'], df['low'], df['close'], timeperiod=14)

    # Money Flow Index
    df['mfi'] = ta.MFI(df['high'], df['low'], df['close'], df['volume'], timeperiod=14)

    return df
```

Volume-based:

```python
def compute_volume_features(df: pd.DataFrame) -> pd.DataFrame:
    """Volume-based features."""

    # Volume z-score (20-day rolling)
    df['volume_mean_20'] = df['volume'].rolling(20).mean()
    df['volume_std_20'] = df['volume'].rolling(20).std()
    df['volume_z_score'] = (df['volume'] - df['volume_mean_20']) / df['volume_std_20']

    # Average dollar volume
    df['dollar_volume'] = df['close'] * df['volume']
    df['avg_dollar_volume_20'] = df['dollar_volume'].rolling(20).mean()

    # On-Balance Volume
    df['obv'] = ta.OBV(df['close'], df['volume'])

    # VWAP deviation
    df['vwap_deviation'] = (df['close'] - df['vwap']) / df['vwap']

    return df
```

Price action:

```python
def compute_price_action_features(df: pd.DataFrame) -> pd.DataFrame:
    """Price action features."""

    # Gap %
    df['gap_pct'] = (df['open'] - df['close'].shift(1)) / df['close'].shift(1)

    # Intraday range
    df['intraday_range_pct'] = (df['high'] - df['low']) / df['open']

    # Distance to 52-week high/low
    df['high_52w'] = df['high'].rolling(252).max()
    df['low_52w'] = df['low'].rolling(252).min()
    df['distance_to_52w_high'] = (df['close'] - df['high_52w']) / df['high_52w']
    df['distance_to_52w_low'] = (df['close'] - df['low_52w']) / df['low_52w']

    # Close position in daily range
    df['close_range_position'] = (df['close'] - df['low']) / (df['high'] - df['low'])

    return df
```

Fibonacci levels:

```python
def compute_fibonacci_levels(df: pd.DataFrame, lookback=60) -> pd.DataFrame:
    """Fibonacci retracement levels."""

    high = df['high'].rolling(lookback).max()
    low = df['low'].rolling(lookback).min()
    diff = high - low

    # Retracement levels
    df['fib_0.236'] = high - 0.236 * diff
```

```
    df['fib_0.382'] = high - 0.382 * diff
    df['fib_0.500'] = high - 0.500 * diff
    df['fib_0.618'] = high - 0.618 * diff

    # Distance to nearest fib level
    fib_levels = df[['fib_0.236', 'fib_0.382', 'fib_0.500', 'fib_0.618']].values
    df['distance_to_nearest_fib'] = np.min(np.abs(fib_levels - df['close'].values[:, None

    # Confluence flag (multiple fib levels nearby)
    df['fib_confluence'] = (df['distance_to_nearest_fib'] / df['close'] < 0.01)

    return df
```

3.2 Cross-Sectional Features

Relative strength:

```
def compute_cross_sectional_features(universe_df: pd.DataFrame) -> pd.DataFrame:
    """
    Compute cross-sectional features across all stocks.

    Args:
        universe_df: DataFrame with all stocks for a given day
    """

    # Return rankings (within universe)
    universe_df['return_1d'] = universe_df.groupby('ticker')['close'].pct_change(1)
    universe_df['return_5d'] = universe_df.groupby('ticker')['close'].pct_change(5)
    universe_df['return_20d'] = universe_df.groupby('ticker')['close'].pct_change(20)

    # Rank within universe (1 = best, 500 = worst)
    universe_df['return_rank_1d'] = universe_df.groupby('date')['return_1d'].rank(ascendi
    universe_df['return_rank_5d'] = universe_df.groupby('date')['return_5d'].rank(ascendi
    universe_df['return_rank_20d'] = universe_df.groupby('date')['return_20d'].rank(ascen

    # Sector relative strength
    # Compare stock return vs sector ETF return
    sector_map = get_sector_mapping()  # ticker -> sector ETF

    for ticker, sector_etf in sector_map.items():
        mask = universe_df['ticker'] == ticker
        sector_return = universe_df[universe_df['ticker'] == sector_etf]['return_5d'].val
        stock_return = universe_df[mask]['return_5d'].values[0]
        universe_df.loc[mask, 'sector_relative_strength'] = stock_return - sector_return

    return universe_df
```

Correlation features:

```
def compute_correlation_features(prices: pd.DataFrame) -> pd.DataFrame:
    """Compute correlation to SPY and other indices."""

    # Pivot to ticker columns
```

```
    returns = prices.pivot(index='date', columns='ticker', values='close').pct_change()

    # Rolling 20-day correlation to SPY
    spy_returns = returns['SPY']

    corr_features = pd.DataFrame()
    for ticker in returns.columns:
        if ticker == 'SPY':
            continue
        corr_features[f'{ticker}_corr_spy'] = returns[ticker].rolling(20).corr(spy_returr

    # Melt back to long format
    corr_features_long = corr_features.melt(
        ignore_index=False,
        var_name='ticker',
        value_name='correlation_to_spy'
    ).reset_index()

    return corr_features_long
```

Peer analysis:

```
def compute_peer_features(ticker: str, universe_df: pd.DataFrame) -> dict:
    """Analyze peer stocks (same sector)."""

    sector = get_sector(ticker)
    peers = get_peers(sector)

    # Peer performance
    peer_returns = universe_df[universe_df['ticker'].isin(peers)]['return_5d']

    features = {
        'peer_median_return_5d': peer_returns.median(),
        'peer_outperformance': universe_df[universe_df['ticker'] == ticker]['return_5d'].
        'peer_percentile': (peer_returns < universe_df[universe_df['ticker'] == ticker]['
    }

    return features
```

3.3 Graph Construction

Dynamic correlation graph:

```
import torch
from torch_geometric.data import Data
import numpy as np

def build_correlation_graph(prices: pd.DataFrame, date: str, threshold=0.3) -> Data:
    """
    Build dynamic graph based on rolling correlation.

    Args:
        prices: Historical price data
```

```
        date: Current date
        threshold: Correlation threshold for edge creation

    Returns:
        PyTorch Geometric Data object
    """

    # Get last 20 days of returns
    lookback_start = pd.to_datetime(date) - pd.timedelta(days=30)
    recent_prices = prices[prices['date'] >= lookback_start]

    # Pivot to ticker columns
    returns = recent_prices.pivot(index='date', columns='ticker', values='close').pct_cha

    # Compute correlation matrix
    corr_matrix = returns.corr()

    # Create edges where correlation > threshold
    tickers = list(corr_matrix.columns)
    ticker_to_idx = {ticker: i for i, ticker in enumerate(tickers)}

    edge_index = []
    edge_attr = []

    for i, ticker_i in enumerate(tickers):
        for j, ticker_j in enumerate(tickers):
            if i != j and abs(corr_matrix.iloc[i, j]) > threshold:
                edge_index.append([i, j])
                edge_attr.append(corr_matrix.iloc[i, j])

    edge_index = torch.tensor(edge_index, dtype=torch.long).t().contiguous()
    edge_attr = torch.tensor(edge_attr, dtype=torch.float).unsqueeze(1)

    # Node features (will be filled by TFT embeddings later)
    num_nodes = len(tickers)
    x = torch.zeros((num_nodes, 128))  # Placeholder

    graph = Data(x=x, edge_index=edge_index, edge_attr=edge_attr)

    return graph, ticker_to_idx
```

Heterogeneous graph (optional advanced):

```
from torch_geometric.data import HeteroData

def build_heterogeneous_graph(prices: pd.DataFrame, date: str) -> HeteroData:
    """
    Build heterogeneous graph with:
    - Stock nodes
    - Sector nodes
    - Edges: stock-sector, stock-stock (correlation), stock-stock (supply chain)
    """

    graph = HeteroData()
```

```
    # Stock nodes
    tickers = prices['ticker'].unique()
    graph['stock'].x = torch.zeros((len(tickers), 128))  # Node features

    # Sector nodes
    sectors = get_all_sectors()
    graph['sector'].x = torch.zeros((len(sectors), 64))

    # Stock-Sector edges
    stock_sector_edges = []
    for i, ticker in enumerate(tickers):
        sector_idx = sectors.index(get_sector(ticker))
        stock_sector_edges.append([i, sector_idx])

    graph['stock', 'belongs_to', 'sector'].edge_index = torch.tensor(
        stock_sector_edges, dtype=torch.long
    ).t()

    # Stock-Stock correlation edges (as before)
    # ...

    # Stock-Stock supply chain edges (scraped from 10-Ks)
    # ...

    return graph
```

3.4 Text Features (LLM-derived)

See Section 5: LLM Agent System for full implementation.

Output schema from TextSummarizerAgent:

```
{
  "ticker": "AAPL",
  "sentiment_score": 0.72,
  "sentiment_confidence": 0.85,
  "key_narratives": ["AI chip demand strong", "Services revenue beat", "China weakness pe
  "event_flags": {
    "earnings_surprise": true,
    "guidance_change": false,
    "regulatory_risk": false,
    "management_change": false
  },
  "news_intensity": "high",
  "contrarian_signals": ["Stock down 2% despite earnings beat - potential overreaction"]
}
```

Feature extraction:

```
def extract_text_features(llm_output: dict) -> pd.Series:
    """Convert LLM JSON output to model features."""

    features = {
```

```
        'sentiment_score': llm_output['sentiment_score'],
        'sentiment_confidence': llm_output['sentiment_confidence'],
        'news_intensity_quiet': llm_output['news_intensity'] == 'quiet',
        'news_intensity_moderate': llm_output['news_intensity'] == 'moderate',
        'news_intensity_high': llm_output['news_intensity'] == 'high',
        'event_flag_earnings': llm_output['event_flags']['earnings_surprise'],
        'event_flag_guidance': llm_output['event_flags']['guidance_change'],
        'event_flag_regulatory': llm_output['event_flags']['regulatory_risk'],
        'has_contrarian_signal': len(llm_output['contrarian_signals']) > 0,
    }

    # Embed key narratives using sentence-transformers
    narrative_text = " ".join(llm_output['key_narratives'])
    narrative_embedding = embed_text(narrative_text)  # 32-dim vector

    for i, val in enumerate(narrative_embedding):
        features[f'narrative_embed_{i}'] = val

    return pd.Series(features)
```

4. Model Architecture

4.1 TFT-GNN Hybrid (Primary Model)

Architecture overview:

```
import torch
import torch.nn as nn
from pytorch_forecasting import TemporalFusionTransformer
from torch_geometric.nn import GATConv

class TFT_GNN_Hybrid(nn.Module):
    """
    Hybrid model combining:
    1. Temporal Fusion Transformer for per-ticker time-series
    2. Graph Attention Network for inter-ticker relationships
    3. Fusion layer for final predictions
    """

    def __init__(
        self,
        tft_config: dict,
        gnn_config: dict,
        output_dim: int = 3  # [return, prob_hit, volatility]
    ):
        super().__init__()

        # 1. TFT encoder
        self.tft = TemporalFusionTransformer.from_dataset(
            training_dataset,
            **tft_config
        )
```

```
        # 2. GNN layers
        self.gnn_layers = nn.ModuleList([
            GATConv(
                in_channels=tft_config['hidden_size'],
                out_channels=gnn_config['hidden_dim'],
                heads=gnn_config['num_heads'],
                dropout=gnn_config['dropout'],
                edge_dim=1  # Edge weight = correlation
            )
            for _ in range(gnn_config['num_layers'])
        ])

        # 3. Fusion layer
        fusion_input_dim = tft_config['hidden_size'] + gnn_config['hidden_dim'] * gnn_cor

        self.fusion = nn.Sequential(
            nn.Linear(fusion_input_dim, 256),
            nn.LayerNorm(256),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(256, 128),
            nn.LayerNorm(128),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(128, output_dim)
        )

        # Quantile prediction heads (for uncertainty)
        self.quantile_heads = nn.ModuleDict({
            'q10': nn.Linear(128, 1),
            'q50': nn.Linear(128, 1),
            'q90': nn.Linear(128, 1),
        })

    def forward(self, batch: dict, graph: Data) -> dict:
        """
        Args:
            batch: TFT batch with features
            graph: PyTorch Geometric graph with edge_index, edge_attr

        Returns:
            predictions: {
                'return': expected 5-day return,
                'prob_hit_long': P(hit target before stop),
                'prob_hit_short': P(hit target short),
                'volatility': predicted volatility,
                'quantiles': {q10, q50, q90}
            }
        """

        # 1. TFT encoding
        # Output shape: (batch_size, hidden_size)
        tft_output = self.tft.encode(batch)
        tft_embeddings = tft_output['encoder_output'][:, -1, :]  # Last time step

        # 2. GNN message passing
```

```python
        # Assume graph.x is initialized with TFT embeddings
        graph.x = tft_embeddings

        gnn_x = graph.x
        for gnn_layer in self.gnn_layers:
            gnn_x = gnn_layer(gnn_x, graph.edge_index, graph.edge_attr)
            gnn_x = torch.relu(gnn_x)

        # 3. Concatenate TFT + GNN embeddings
        combined = torch.cat([tft_embeddings, gnn_x], dim=-1)

        # 4. Fusion layer
        fusion_output = self.fusion[:-1](combined)  # Up to last linear layer

        # Main predictions
        main_output = self.fusion[-1](fusion_output)

        expected_return = main_output[:, 0]
        prob_hit_long = torch.sigmoid(main_output[:, 1])
        volatility = torch.exp(main_output[:, 2])  # Ensure positive

        # Quantile predictions
        quantiles = {
            'q10': self.quantile_heads['q10'](fusion_output).squeeze(),
            'q50': self.quantile_heads['q50'](fusion_output).squeeze(),
            'q90': self.quantile_heads['q90'](fusion_output).squeeze(),
        }

        return {
            'return': expected_return,
            'prob_hit_long': prob_hit_long,
            'volatility': volatility,
            'quantiles': quantiles
        }
```

TFT Configuration:

```python
tft_config = {
    # Architecture
    'hidden_size': 128,
    'lstm_layers': 2,
    'attention_head_size': 4,
    'dropout': 0.1,

    # Input features
    'time_varying_known_categoricals': ['day_of_week', 'month'],
    'time_varying_known_reals': ['days_to_earnings'],

    'time_varying_unknown_categoricals': [],
    'time_varying_unknown_reals': [
        # Price/volume
        'close', 'volume', 'vwap',
        # Technical indicators
        'rsi_14', 'macd', 'macd_signal', 'bbands_pct', 'atr_14',
        'stoch_k', 'adx', 'mfi',
```

```
        # Volume
        'volume_z_score', 'vwap_deviation',
        # Price action
        'gap_pct', 'intraday_range_pct', 'distance_to_52w_high',
    ],

    'static_categoricals': ['sector', 'market_cap_bucket'],
    'static_reals': [],

    # Targets
    'target': 'return_5d',
    'target_normalizer': 'GroupNormalizer',

    # Training
    'learning_rate': 1e-3,
    'max_encoder_length': 60,  # 60 days lookback
    'max_prediction_length': 5,  # 5 days forward
}
```

GNN Configuration:

```
gnn_config = {
    'hidden_dim': 64,
    'num_heads': 8,  # Multi-head attention
    'num_layers': 2,  # Avoid over-smoothing
    'dropout': 0.1,
    'edge_threshold': 0.3,  # Min correlation for edge
}
```

4.2 LightGBM Cross-Sectional Ranker

Purpose: Given all predictions for a day, rank which stocks are most likely to outperform.

```
import lightgbm as lgb

def train_lgbm_ranker(features: pd.DataFrame, labels: pd.DataFrame):
    """
    Train LightGBM ranker.

    Args:
        features: DataFrame with all features per (date, ticker)
        labels: Target returns
    """

    # Create query groups (one group per date)
    features['date'] = pd.to_datetime(features['date'])
    features = features.sort_values('date')

    query_ids = features.groupby('date').size().values  # Sizes of each group

    X = features.drop(['date', 'ticker'], axis=1)
    y = labels['return_5d']
```

```
    # LightGBM ranker
    model = lgb.LGBMRanker(
        objective='lambdarank',
        metric='ndcg',
        boosting_type='gbdt',
        num_leaves=31,
        learning_rate=0.05,
        n_estimators=500,
        max_depth=6,
        min_child_samples=20,
        subsample=0.8,
        colsample_bytree=0.8,
    )

    model.fit(
        X, y,
        group=query_ids,
        eval_set=[(X_val, y_val)],
        eval_group=[query_ids_val],
        early_stopping_rounds=50,
        verbose=50
    )

    return model
```

Feature set for LightGBM:

```
lgbm_features = [
    # From TFT-GNN
    'tft_gnn_return_pred',
    'tft_gnn_prob_long',
    'tft_gnn_volatility',
    'tft_gnn_quantile_10',
    'tft_gnn_quantile_90',

    # Technical
    'rsi_14', 'macd_signal', 'bbands_pct', 'atr_14',
    'distance_to_52w_high', 'distance_to_52w_low',
    'volume_z_score', 'adx', 'mfi',

    # Cross-sectional
    'return_rank_5d', 'return_rank_20d',
    'sector_relative_strength',
    'correlation_to_spy',
    'peer_outperformance',

    # Text (from LLM)
    'sentiment_score', 'sentiment_confidence',
    'news_intensity_high',
    'event_flag_earnings', 'event_flag_guidance',
    'has_contrarian_signal',
    *[f'narrative_embed_{i}' for i in range(32)],

    # Pattern flags
    'breakout_52w', 'pattern_confidence',
```

```
    'fib_confluence',

    # Macro
    'spy_return_5d', 'vix_level', 'treasury_yield_10y',
]
```

4.3 ARIMA/GARCH Baseline

Purpose: Volatility forecasting + regime detection.

```
from statsmodels.tsa.arima.model import ARIMA
from arch import arch_model

def fit_arima_garch(ticker: str, returns: pd.Series) -> dict:
    """
    Fit ARIMA + GARCH for a ticker.

    Args:
        returns: Daily returns

    Returns:
        Forecasts and regime info
    """

    # 1. ARIMA for mean
    arima = ARIMA(returns, order=(1, 0, 1)).fit()
    return_forecast = arima.forecast(steps=5).mean()

    # 2. GARCH for volatility
    garch = arch_model(returns * 100, vol='Garch', p=1, q=1).fit(disp='off')
    volatility_forecast = garch.forecast(horizon=5).variance.values[-1, :].mean() / 100

    # 3. Regime detection
    # Simple: is return series trending or mean-reverting?
    adf_stat = adfuller(returns)[1]  # Augmented Dickey-Fuller test
    hurst = compute_hurst_exponent(returns)

    if hurst > 0.55:
        regime = 'trending'
    elif hurst < 0.45:
        regime = 'mean_reverting'
    else:
        regime = 'random'

    return {
        'arima_return_forecast': return_forecast,
        'garch_volatility_forecast': volatility_forecast ** 0.5,  # Std dev
        'regime': regime,
        'hurst_exponent': hurst,
    }
```

4.4 Pattern Detection

Chart pattern detection (rule-based):

```
def detect_chart_patterns(df: pd.DataFrame) -> dict:
    """
    Detect common chart patterns using rule-based logic.
    """

    patterns = {
        'breakout_52w_high': False,
        'breakout_52w_low': False,
        'double_bottom': False,
        'double_top': False,
        'head_and_shoulders': False,
        'ascending_triangle': False,
        'descending_triangle': False,
        'bull_flag': False,
        'bear_flag': False,
        'confidence': 0.0,
    }

    # 52-week breakout
    if df['close'].iloc[-1] >= df['high_52w'].iloc[-2]:
        patterns['breakout_52w_high'] = True
        patterns['confidence'] = max(patterns['confidence'], 0.9)

    if df['close'].iloc[-1] <= df['low_52w'].iloc[-2]:
        patterns['breakout_52w_low'] = True
        patterns['confidence'] = max(patterns['confidence'], 0.9)

    # Double bottom (simplified)
    # Look for two local minima at similar price levels
    local_minima = (df['low'].shift(1) > df['low']) & (df['low'].shift(-1) > df['low'])
    minima_prices = df[local_minima]['low'].values[-2:]

    if len(minima_prices) == 2 and abs(minima_prices[0] - minima_prices[1]) / minima_pric
        # Two bottoms within 2% of each other
        if df['close'].iloc[-1] > max(minima_prices) * 1.05:  # Breakout above
            patterns['double_bottom'] = True
            patterns['confidence'] = max(patterns['confidence'], 0.75)

    # Similar logic for other patterns...
    # (In production, use a library like TA-Lib patterns or train a CNN)

    return patterns
```

ML-based pattern detection (optional):

```
import torch.nn as nn

class PatternCNN(nn.Module):
    """CNN for chart pattern classification."""
```

```
    def __init__(self, num_patterns=10):
        super().__init__()

        # Input: (batch, 1, 60, 4) - 60 days, 4 channels (OHLC normalized)
        self.conv_layers = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=(3, 4), stride=1, padding=(1, 0)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 1)),

            nn.Conv2d(32, 64, kernel_size=(3, 1), stride=1, padding=(1, 0)),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 1)),

            nn.Conv2d(64, 128, kernel_size=(3, 1), stride=1, padding=(1, 0)),
            nn.ReLU(),
            nn.AdaptiveAvgPool2d((1, 1)),
        )

        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(64, num_patterns),
        )

    def forward(self, x):
        x = self.conv_layers(x)
        logits = self.fc(x)
        return logits  # Multi-label classification
```

4.5 Meta-Ensemble

Combine all model outputs:

```
def ensemble_predictions(
    tft_gnn_preds: dict,
    lgbm_ranks: np.ndarray,
    arima_garch: dict,
    patterns: dict,
    text_features: dict
) -> dict:
    """
    Ensemble all model predictions.

    Simple approach: Weighted average + adjustments.
    """

    # Base prediction (weighted average)
    base_return = (
        0.5 * tft_gnn_preds['return'] +
        0.3 * lgbm_ranks['return_pred'] +
        0.2 * arima_garch['arima_return_forecast']
    )
```

```
    base_prob = (
        0.6 * tft_gnn_preds['prob_hit_long'] +
        0.4 * lgbm_ranks['prob_pred']
    )

    # Volatility (prefer GARCH for vol)
    volatility = (
        0.4 * tft_gnn_preds['volatility'] +
        0.6 * arima_garch['garch_volatility_forecast']
    )

    # Adjustments based on patterns & text
    return_adj = 0.0
    prob_adj = 0.0

    # Technical pattern bonuses
    if patterns['breakout_52w_high']:
        return_adj += 0.01  # +1% expected return bonus
        prob_adj += 0.05

    if patterns['double_bottom'] and patterns['confidence'] > 0.7:
        return_adj += 0.008
        prob_adj += 0.04

    # Sentiment alignment bonus
    if np.sign(base_return) == np.sign(text_features['sentiment_score']):
        prob_adj += 0.03
    else:
        # Divergence penalty
        prob_adj -= 0.05

    # Regime adjustment
    if arima_garch['regime'] == 'mean_reverting' and abs(base_return) > 0.05:
        # Strong mean reversion signal - reduce expected return
        return_adj -= 0.02

    final_return = base_return + return_adj
    final_prob = np.clip(base_prob + prob_adj, 0, 1)

    return {
        'expected_return': final_return,
        'hit_probability': final_prob,
        'volatility': volatility,
        'quantiles': tft_gnn_preds['quantiles'],
    }
```

5. LLM Agent System

5.1 Agent Architecture

Overview:

```
User Query / Daily Pipeline
        │
        ├──> TextSummarizerAgent (news → structured features)
        │
        ├──> PatternDetectorAgent (chart analysis)
        │
        ├──> PolicyAgent (trade curation + rules)
        │
        ├──> RelatedStockAgent (peer/sympathy plays)
        │
        └──> ExplainerAgent (generate rationale)
```

5.2 TextSummarizerAgent

Implementation:

```
from langchain.agents import Tool, AgentExecutor
from langchain_openai import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from pydantic import BaseModel, Field
import json

class TextSummary(BaseModel):
    """Structured output from TextSummarizerAgent."""
    ticker: str
    sentiment_score: float = Field(description="Sentiment from -1 (bearish) to 1 (bullish
    sentiment_confidence: float = Field(description="Confidence 0-1")
    key_narratives: list[str] = Field(description="2-3 key themes")
    event_flags: dict = Field(description="Binary flags for key events")
    news_intensity: str = Field(description="quiet | moderate | high")
    contrarian_signals: list[str] = Field(description="Text-price divergences")

class TextSummarizerAgent:
    """LLM agent to process news/text into structured features."""

    def __init__(self, model="gpt-4o-mini"):
        self.llm = ChatOpenAI(model=model, temperature=0.1)

        self.prompt = ChatPromptTemplate.from_messages([
            ("system", """You are a financial text analyzer specializing in swing trading

Your job: Analyze news headlines, analyst changes, and social sentiment for a stock, ther

Key focus areas:
- Sentiment (bullish/bearish/neutral)
- Event-driven catalysts (earnings, guidance, management changes, regulatory)
- Narrative themes (e.g., "AI demand", "margin pressure", "sector rotation")
- Contrarian signals (stock price contradicts news sentiment)

Output format: JSON matching TextSummary schema.
```

```
Be concise. Avoid generic statements. Focus on actionable signals."""),
            ("user", """Ticker: {ticker}
Date: {date}

Headlines (last 24h):
{headlines}

Analyst Changes:
{analyst_changes}

Recent Price Action:
- 1-day return: {return_1d:.2%}
- 5-day return: {return_5d:.2%}
- Volume vs avg: {volume_ratio:.1f}x

Analyze and output JSON.""")
        ])

    def summarize(
        self,
        ticker: str,
        headlines: list[str],
        analyst_changes: list[dict],
        price_context: dict,
        date: str
    ) -> TextSummary:
        """
        Summarize text data for a ticker.

        Returns:
            TextSummary object
        """

        # Format inputs
        headlines_str = "\n".join([f"- {h}" for h in headlines])

        analyst_str = "\n".join([
            f"- {a['firm']}: {a['action']} {a['rating']} (target ${a['target']})"
            for a in analyst_changes
        ])

        # Call LLM
        messages = self.prompt.format_messages(
            ticker=ticker,
            date=date,
            headlines=headlines_str or "No news",
            analyst_changes=analyst_str or "No changes",
            return_1d=price_context['return_1d'],
            return_5d=price_context['return_5d'],
            volume_ratio=price_context['volume_ratio']
        )

        response = self.llm.invoke(messages)

        # Parse JSON response
```

```
        try:
            data = json.loads(response.content)
            summary = TextSummary(**data)
        except Exception as e:
            # Fallback to neutral
            summary = TextSummary(
                ticker=ticker,
                sentiment_score=0.0,
                sentiment_confidence=0.5,
                key_narratives=["No significant news"],
                event_flags={},
                news_intensity="quiet",
                contrarian_signals=[]
            )

        return summary

    def batch_summarize(self, tickers: list[str], news_data: dict, price_data: dict) -> d
        """Process multiple tickers in parallel."""

        from concurrent.futures import ThreadPoolExecutor

        def process_one(ticker):
            return self.summarize(
                ticker=ticker,
                headlines=news_data.get(ticker, {}).get('headlines', []),
                analyst_changes=news_data.get(ticker, {}).get('analyst_changes', []),
                price_context=price_data[ticker],
                date=price_data[ticker]['date']
            )

        with ThreadPoolExecutor(max_workers=10) as executor:
            results = list(executor.map(process_one, tickers))

        return {ticker: result for ticker, result in zip(tickers, results)}
```

Example usage:

```
agent = TextSummarizerAgent()

summary = agent.summarize(
    ticker="AAPL",
    headlines=[
        "Apple unveils new AI features in iOS 18",
        "iPhone sales beat estimates in China",
        "Analysts raise price targets on AI optimism"
    ],
    analyst_changes=[
        {"firm": "Morgan Stanley", "action": "raised", "rating": "Overweight", "target":
    ],
    price_context={
        "return_1d": 0.023,
        "return_5d": 0.048,
        "volume_ratio": 1.4,
        "date": "2025-11-18"
```

```
    },
    date="2025-11-18"
)

print(summary.json(indent=2))
```

5.3 PolicyAgent

Role: Apply trading rules, enforce constraints, curate final trade list.

```
class PolicyAgent:
    """LLM agent to apply trading policy and curate final trades."""

    def __init__(self, model="gpt-4o"):
        self.llm = ChatOpenAI(model=model, temperature=0.2)

        self.prompt = ChatPromptTemplate.from_messages([
            ("system", """You are a risk-aware swing trading strategist.

Your job: Review candidate trades from quantitative models and select the best 10-15 trac
1. Have high probability (>60%)
2. Show technical confirmation (breakouts, patterns, indicators aligned)
3. Avoid negative divergences (model says buy but news is terrible)
4. Maintain portfolio diversification (sectors, factors)
5. Respect risk limits (max positions, sector exposure, position sizing)

You receive:
- Candidate list with numeric scores and text summaries
- Current portfolio state
- Risk rules

Output: JSON array of selected trades with rationale for each.

Be selective. Quality > quantity. If a trade has conflicting signals (great quant signal
            ("user", """Date: {date}

Top Candidates:
{candidates_table}

Current Portfolio:
- Open positions: {num_positions}
- Sector exposure: {sector_exposure}
- Available capital: ${cash:,.0f}

Risk Rules:
- Max 15 concurrent positions
- Max 25% allocation per sector
- Min probability: 60%
- Avoid: earnings in next 2 days, extreme volatility

Select up to {max_trades} best trades. For each trade, output:
{{
  "ticker": "...",
  "side": "buy" or "sell",
  "target_pct": <float>,
```

```
  "stop_pct": <float>,
  "probability": <float>,
  "position_size_pct": <suggested % of portfolio>,
  "rationale": ["reason 1", "reason 2", "reason 3"]
}}

Output JSON array.""")
        ])

    def curate_trades(
        self,
        candidates: pd.DataFrame,
        portfolio_state: dict,
        risk_rules: dict,
        date: str,
        max_trades: int = 12
    ) -> list[dict]:
        """
        Curate final trade list from candidates.

        Args:
            candidates: DataFrame with top N candidates
            portfolio_state: Current portfolio info
            risk_rules: Risk constraints
            date: Current date
            max_trades: Max trades to output

        Returns:
            List of trade recommendations
        """

        # First, apply hard filters in Python (fast, deterministic)
        candidates = self._apply_hard_filters(candidates, risk_rules)

        # Format candidates table for LLM
        candidates_table = self._format_candidates_table(candidates)

        # Call LLM
        messages = self.prompt.format_messages(
            date=date,
            candidates_table=candidates_table,
            num_positions=len(portfolio_state['positions']),
            sector_exposure=json.dumps(portfolio_state['sector_exposure'], indent=2),
            cash=portfolio_state['cash'],
            max_trades=max_trades
        )

        response = self.llm.invoke(messages)

        # Parse JSON
        try:
            trades = json.loads(response.content)
        except:
            # Fallback: top trades by priority score
            trades = self._fallback_selection(candidates, max_trades)
```

```
        return trades

    def _apply_hard_filters(self, candidates: pd.DataFrame, rules: dict) -> pd.DataFrame:
        """Apply non-negotiable filters."""

        filtered = candidates[
            (candidates['avg_dollar_volume'] > 5_000_000) &  # Liquidity
            (candidates['bid_ask_spread_pct'] < 0.5) &  # Low spread
            (candidates['probability'] >= rules['min_probability']) &
            (candidates['days_to_earnings'] > 2) &  # No imminent earnings
            (candidates['volatility'] < rules['max_volatility'])
        ]

        return filtered

    def _format_candidates_table(self, df: pd.DataFrame) -> str:
        """Format DataFrame as text table for LLM."""

        cols = [
            'ticker', 'side', 'expected_return', 'probability',
            'priority_score', 'sector', 'sentiment_score',
            'key_narratives', 'pattern_type', 'pattern_confidence'
        ]

        table = df[cols].head(50).to_string(index=False, max_rows=50)

        return table

    def _fallback_selection(self, df: pd.DataFrame, n: int) -> list[dict]:
        """Fallback if LLM fails: simple top-N by priority."""

        top_n = df.nlargest(n, 'priority_score')

        trades = []
        for _, row in top_n.iterrows():
            trades.append({
                'ticker': row['ticker'],
                'side': row['side'],
                'target_pct': row['expected_return'],
                'stop_pct': -row['volatility'] * 2,  # 2-sigma stop
                'probability': row['probability'],
                'position_size_pct': 100 / n,  # Equal weight
                'rationale': ["High priority score", "No LLM rationale available"]
            })

        return trades
```

5.4 PatternDetectorAgent

Role: Validate ambiguous chart patterns using LLM vision (optional advanced).

```
class PatternDetectorAgent:
    """LLM agent to validate chart patterns (uses GPT-4 Vision if charts provided)."""

    def __init__(self):
```

```
        self.rule_based_scanner = detect_chart_patterns  # From Section 4.4
        self.llm = ChatOpenAI(model="gpt-4o", temperature=0.0)

    def detect_patterns(
        self,
        ticker: str,
        ohlcv_data: pd.DataFrame,
        chart_image: str = None  # Base64-encoded chart image (optional)
    ) -> dict:
        """
        Detect chart patterns.

        If chart_image provided, use GPT-4 Vision for validation.
        Otherwise, rely on rule-based detection.
        """

        # 1. Rule-based detection (fast)
        patterns = self.rule_based_scanner(ohlcv_data)

        # 2. If low confidence or ambiguous, ask LLM
        if patterns['confidence'] < 0.7 and chart_image:
            llm_validation = self._validate_with_vision(ticker, chart_image, patterns)
            patterns.update(llm_validation)

        return patterns

    def _validate_with_vision(self, ticker: str, chart_image: str, detected_patterns: dic
        """Use GPT-4 Vision to validate patterns."""

        prompt = f"""Analyze this {ticker} price chart (60-day view).

Detected patterns (algorithmic): {detected_patterns['candidates']}

Questions:
1. Are these patterns valid? (0-1 confidence for each)
2. Any other obvious patterns (breakouts, flags, triangles, H&S)?
3. Overall technical setup quality: strong | moderate | weak

Output JSON:
{{
  "patterns": [
    {{"type": "double_bottom", "confidence": 0.85, "target_price": 152.3}}
  ],
  "technical_setup_quality": "strong"
}}"""

        # In production, you'd encode chart as base64 and pass to GPT-4V
        # For now, placeholder
        # response = self.llm.invoke([{"type": "image_url", "image_url": chart_image}, {'

        # Fallback without vision
        return detected_patterns
```

5.5 ExplainerAgent

Role: Generate human-readable explanations and daily briefs.

```
class ExplainerAgent:
    """LLM agent to explain recommendations and generate daily briefs."""

    def __init__(self):
        self.llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.3)

    def explain_trade(self, trade: dict, features: dict) -> str:
        """Generate explanation for a single trade."""

        prompt = f"""Explain why this swing trade is recommended:

Ticker: {trade['ticker']}
Side: {trade['side']}
Target: {trade['target_pct']:.1%}
Probability: {trade['probability']:.0%}

Model Signals:
- Expected return: {features['expected_return']:.2%}
- Technical setup: {features['pattern_type']} (confidence: {features['pattern_confidence'
- Sentiment: {features['sentiment_score']:.2f}
- Key narratives: {', '.join(features['key_narratives'])}
- Sector strength: {features['sector_relative_strength']:.2%}

Write 2-3 concise bullet points explaining why this is a good opportunity.
Focus on: (1) technical setup, (2) fundamental catalyst, (3) risk factors."""

        response = self.llm.invoke(prompt)

        return response.content

    def generate_daily_brief(self, trades: list[dict], market_context: dict) -> str:
        """Generate daily summary brief."""

        prompt = f"""Generate a daily swing trading brief:

Date: {market_context['date']}

Market Context:
- SPY: {market_context['spy_return']:.2%} (5-day)
- VIX: {market_context['vix']}
- Sector leaders: {', '.join(market_context['top_sectors'])}

Today's Recommendations ({len(trades)} trades):
{self._format_trade_summary(trades)}

Write a 3-paragraph brief:
1. Market backdrop (1-2 sentences)
2. Key themes in today's picks (sector tilts, common patterns, catalysts)
3. Risk considerations

Tone: Professional but accessible. Focus on actionable insights."""
```

```
        response = self.llm.invoke(prompt)

        return response.content

    def _format_trade_summary(self, trades: list[dict]) -> str:
        """Format trades as bullet list."""

        lines = []
        for t in trades:
            lines.append(f"- {t['ticker']} ({t['side']}): {t['target_pct']:.1%} target, {

        return "\n".join(lines)
```

6. Ranking & Prioritization

6.1 Priority Score Formula

Formula:

```
PriorityScore = BaseSignal × Multiplier

Where:
  BaseSignal = 0.5 × P(hit) + 0.3 × Rank + 0.2 × ExpectedReturn

  Multiplier = 1.0
             + TechnicalBonuses
             + SentimentBonuses
             + PeerBonuses
             - Penalties
```

Implementation:

```
def compute_priority_score(
    tft_gnn_preds: dict,
    lgbm_ranks: dict,
    patterns: dict,
    text_features: dict,
    cross_sectional: dict,
    macro: dict
) -> float:
    """
    Compute priority score for a trade candidate.

    Returns:
        priority_score: 0-1.5 scale (capped)
    """

    # ===== BASE SIGNAL (0-1 scale) =====
    prob = tft_gnn_preds['prob_hit_long']
    rank_normalized = 1 - (lgbm_ranks['rank'] / 500)  # Rank 1 → 1.0, rank 500 → 0.0
    expected_return_normalized = np.clip(tft_gnn_preds['return'] / 0.10, 0, 1)  # 10% ret
```

```
base = (
    0.5 * prob +
    0.3 * rank_normalized +
    0.2 * expected_return_normalized
)

# ===== MULTIPLIER (starts at 1.0) =====
multiplier = 1.0

# --- Technical Confirmations (Bonuses) ---

# Breakout (strong signal)
if patterns['breakout_52w_high']:
    multiplier += 0.25

# High-confidence pattern
if patterns['pattern_confidence'] > 0.8:
    multiplier += 0.15
elif patterns['pattern_confidence'] > 0.6:
    multiplier += 0.08

# Fibonacci confluence
if patterns['fib_confluence']:
    multiplier += 0.10

# Strong trend (ADX > 25)
if cross_sectional.get('adx', 0) > 25:
    multiplier += 0.10

# Volume surge
if cross_sectional.get('volume_z_score', 0) > 2.0:
    multiplier += 0.08

# --- Sentiment Alignment (Bonuses/Penalties) ---

sentiment = text_features['sentiment_score']
signal_direction = np.sign(tft_gnn_preds['return'])

if sentiment * signal_direction > 0.5:
    # Sentiment strongly agrees with signal
    multiplier += 0.20
elif sentiment * signal_direction > 0.2:
    # Mild agreement
    multiplier += 0.10
elif abs(sentiment) < 0.1:
    # Neutral sentiment (OK, no headwinds)
    multiplier += 0.05
else:
    # DIVERGENCE: model says buy but sentiment is negative
    multiplier -= 0.30  # Heavy penalty

# High-confidence sentiment
if text_features['sentiment_confidence'] > 0.8:
    multiplier += 0.05
```

```
    # --- Cross-Asset Context (Bonuses) ---

    # Sector momentum
    if cross_sectional.get('sector_relative_strength', 0) > 0.03:  # Sector up 3%+
        multiplier += 0.15

    # High correlation to SPY + SPY trending up
    if cross_sectional.get('correlation_to_spy', 0) > 0.7 and macro.get('spy_trend') == '
        multiplier += 0.10

    # Peer outperformance
    if cross_sectional.get('peer_outperformance', 0) > 0.02:
        multiplier += 0.08

    # --- Risk Factors (Penalties) ---

    # High volatility environment
    if text_features.get('volatility_risk') == 'high' or tft_gnn_preds['volatility'] > 0.
        multiplier -= 0.15

    # Earnings imminent (within 3 days)
    if cross_sectional.get('days_to_earnings', 999) < 3:
        multiplier -= 0.25

    # Low liquidity
    if cross_sectional.get('avg_dollar_volume', 1e9) < 10_000_000:
        multiplier -= 0.20

    # Macro headwinds (e.g., VIX spike)
    if macro.get('vix', 15) > 30:
        multiplier -= 0.15

    # Contrarian signal (price down despite good news - could be overreaction)
    if text_features.get('has_contrarian_signal'):
        # This could be bonus or penalty depending on context
        # For now, small bonus (potential mean reversion)
        multiplier += 0.05

    # ===== FINAL SCORE =====
    priority = base * multiplier

    # Cap at 1.5 (prevent extreme values)
    priority = np.clip(priority, 0, 1.5)

    return priority
```

6.2 Risk-Adjusted Position Sizing

Kelly Criterion (optional):

```
def compute_position_size(
    probability: float,
    target_pct: float,
    stop_pct: float,
    risk_free_rate: float = 0.05
```

```
) -> float:
    """
    Compute Kelly-optimal position size.

    Kelly% = (p × b - (1-p)) / b
    where p = probability, b = (target / stop)

    Returns:
        position_size_pct: 0-100 (% of portfolio)
    """

    # Reward/risk ratio
    b = abs(target_pct / stop_pct)

    # Kelly formula
    kelly_pct = (probability * b - (1 - probability)) / b

    # Fractional Kelly (use 50% of full Kelly for safety)
    fractional_kelly = kelly_pct * 0.5

    # Cap at 10% per position (risk management)
    position_size = np.clip(fractional_kelly * 100, 0, 10)

    return position_size
```

Equal risk sizing (simpler):

```
def equal_risk_position_size(
    stop_pct: float,
    portfolio_risk_per_trade: float = 0.01  # 1% risk per trade
) -> float:
    """
    Size position so each trade risks the same % of portfolio.

    If stop is 3% and you want to risk 1% of portfolio:
    position_size = 1% / 3% = 33% of portfolio
    """

    position_size = (portfolio_risk_per_trade / abs(stop_pct)) * 100

    # Cap at 10%
    position_size = min(position_size, 10)

    return position_size
```

7. Cloud Architecture & Stack

7.1 AWS Architecture

```
┌─────────────────────────────────────────────────────────────────┐
│                     AWS Cloud Infrastructure                    │
│                                                                 │
│  ┌──────────────────┐                                           │
│  │   Route 53       │ DNS                                       │
│  └────────┬─────────┘                                           │
│          │                                                      │
│          v                                                      │
│  ┌──────────────────┐      ┌────────────────────┐               │
│  │     ALB          │─────▶│  ECS Fargate       │ (FastAPI)     │
│  │  (Load Bal.)     │      │ Service            │               │
│  └──────────────────┘      └─────────┬──────────┘               │
│                                │                                │
│                                │                                │
│  ┌─────────────────────────────┴───────────────────────────┐    │
│  │                  Compute Layer                          │    │
│  │                                                         │    │
│  │  ┌──────────────────┐  ┌──────────────────┐  ┌──────────────────┐
│  │  │  EC2 g5.xl       │  │ ECS Fargate      │  │  Lambda          │
│  │  │  (Training)      │  │ (Inference)      │  │ (Webhooks)       │
│  │  └──────────────────┘  └──────────────────┘  └──────────────────┘
│  └─────────────────────────────────────────────────────────┘    │
│                                │                                │
│                                v                                │
│  ┌─────────────────────────────────────────────────────────┐    │
│  │                  Orchestration                          │    │
│  │                                                         │    │
│  │               ┌────────────────────────┐                │    │
│  │               │  Prefect Cloud         │                │    │
│  │               │  (Workflow DAGs)       │                │    │
│  │               └───────────┬────────────┘                │    │
│  └───────────────────────────┼─────────────────────────────┘    │
│                           │                                     │
│                           v                                     │
│  ┌─────────────────────────────────────────────────────────┐    │
│  │                  Storage Layer                          │    │
│  │                                                         │    │
│  │  ┌────────────┐  ┌─────────────────┐  ┌──────────────────┐  │
│  │  │  S3        │  │ RDS Postgres    │  │  Redis           │  │
│  │  │(Data Lake  │  │(TimescaleDB)    │  │(Cache/Queue)     │  │
│  │  └────────────┘  └─────────────────┘  └──────────────────┘  │
│  │                                                         │    │
│  │  ┌────────────────────────┐                             │    │
│  │  │  Feast Feature         │                             │    │
│  │  │     Store              │                             │    │
│  │  └────────────────────────┘                             │    │
│  └─────────────────────────────────────────────────────────┘    │
│                                                                 │
│  ┌─────────────────────────────────────────────────────────┐    │
│  │                Monitoring & MLOps                       │    │
│  │                                                         │    │
│  │  ┌──────────────┐  ┌──────────────┐  ┌──────────────────┐   │
│  │  │ CloudWatch   │  │   MLflow     │  │ Prometheus       │   │
│  │  │  (Logs)      │  │(Experiments) │  │ (Metrics)        │   │
```

```
│  │  └────────┘  └──────────┘  └────────────┘      │  │
│  └──────────────────────────────────────────────────┘   │
└───────────────────────────────────────────────────────┘
```

7.2 Detailed Stack Specifications

Compute

Component	Service	Instance Type	Purpose	Cost Est.
Model Training	EC2	g5.xlarge (1x A10G GPU)	Weekly TFT-GNN retraining	~$30/week (spot)
Daily Inference	ECS Fargate	4 vCPU, 8GB RAM	EOD batch inference	~$15/month
API Server	ECS Fargate	2 vCPU, 4GB RAM	FastAPI backend	~$20/month
LLM Inference	OpenAI API	-	Text processing	~$8/month
Orchestration	Prefect Cloud	Free tier	Workflow scheduling	Free

Total Compute: ~$140/month

Storage

Component	Service	Capacity	Purpose	Cost Est.
Raw Data	S3 Standard	100 GB	OHLCV, news	~$2/month
Processed Data	S3 Intelligent Tier	50 GB	Features, graphs	~$1/month
Model Artifacts	S3 Standard-IA	10 GB	Model checkpoints	~$0.50/month
Time-Series DB	RDS Postgres (TimescaleDB)	db.t3.medium	Price/features/predictions	~$50/month
Cache	ElastiCache Redis	cache.t3.micro	Feature cache	~$15/month

Total Storage: ~$68/month

Data Sources

Provider	Data Type	Cost
Polygon.io	Market data (OHLCV)	$199/month
Finnhub	News + fundamentals	$80/month
FRED API	Macro data	Free

Total Data: ~$280/month

Total Monthly Cost: ~$490

7.3 Infrastructure as Code (Terraform)

```
# main.tf

terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
}

# S3 Buckets
resource "aws_s3_bucket" "data_lake" {
  bucket = "swing-trading-data-lake"

  tags = {
    Name = "Trading System Data Lake"
  }
}

resource "aws_s3_bucket_versioning" "data_lake_versioning" {
  bucket = aws_s3_bucket.data_lake.id

  versioning_configuration {
    status = "Enabled"
  }
}

# RDS Postgres (TimescaleDB)
resource "aws_db_instance" "timescale" {
  identifier           = "trading-timescaledb"
  engine               = "postgres"
  engine_version       = "15.4"
  instance_class       = "db.t3.medium"
  allocated_storage    = 100
  storage_type         = "gp3"

  db_name  = "trading"
  username = "admin"
  password = var.db_password  # From secrets

  publicly_accessible = false
  vpc_security_group_ids = [aws_security_group.db_sg.id]

  backup_retention_period = 7

  tags = {
```

```
    Name = "Trading TimescaleDB"
  }
}

# ECS Cluster
resource "aws_ecs_cluster" "trading_cluster" {
  name = "trading-system-cluster"
}

# ECS Task Definition (FastAPI)
resource "aws_ecs_task_definition" "api_task" {
  family                   = "trading-api"
  network_mode             = "awsvpc"
  requires_compatibilities = ["FARGATE"]
  cpu                      = "2048"
  memory                   = "4096"

  container_definitions = jsonencode([
    {
      name      = "api"
      image     = "${aws_ecr_repository.api.repository_url}:latest"
      essential = true

      portMappings = [
        {
          containerPort = 8000
          protocol      = "tcp"
        }
      ]

      environment = [
        {
          name  = "DB_HOST"
          value = aws_db_instance.timescale.address
        },
        {
          name  = "REDIS_HOST"
          value = aws_elasticache_cluster.redis.cache_nodes[0].address
        }
      ]

      logConfiguration = {
        logDriver = "awslogs"
        options = {
          "awslogs-group"         = "/ecs/trading-api"
          "awslogs-region"        = "us-east-1"
          "awslogs-stream-prefix" = "ecs"
        }
      }
    }
  ])
}

# ElastiCache Redis
resource "aws_elasticache_cluster" "redis" {
  cluster_id           = "trading-cache"
```

```
  engine               = "redis"
  node_type            = "cache.t3.micro"
  num_cache_nodes      = 1
  parameter_group_name = "default.redis7"
  port                 = 6379

  security_group_ids = [aws_security_group.redis_sg.id]
}

# More resources: ALB, Security Groups, IAM roles, etc.
```

7.4 Docker Containers

API Dockerfile:

```
# Dockerfile.api

FROM python:3.11-slim

WORKDIR /app

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

# Copy application code
COPY ./api /app/api
COPY ./models /app/models
COPY ./utils /app/utils

# Expose port
EXPOSE 8000

# Run FastAPI
CMD ["uvicorn", "api.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Training Dockerfile:

```
# Dockerfile.training

FROM nvidia/cuda:12.1.0-cudnn8-devel-ubuntu22.04

# Install Python
RUN apt-get update && apt-get install -y python3.11 python3-pip

WORKDIR /app

# Install PyTorch + dependencies
COPY requirements-training.txt .
RUN pip install --no-cache-dir -r requirements-training.txt

# Copy training code
COPY ./training /app/training
```

```
COPY ./models /app/models
COPY ./utils /app/utils

# Entry point
CMD ["python3", "training/train_tft_gnn.py"]
```

8. Daily Pipeline (EOD Flow)

8.1 Prefect Workflow

Complete daily pipeline:

```
from prefect import flow, task
from prefect.task_runners import ConcurrentTaskRunner
from datetime import datetime, timedelta
import pandas as pd

@task(retries=2, retry_delay_seconds=60)
def fetch_market_data(date: str) -> pd.DataFrame:
    """Fetch EOD OHLCV data for S&P 500."""
    from polygon import RESTClient

    client = RESTClient(api_key=os.getenv("POLYGON_API_KEY"))

    tickers = get_sp500_tickers()

    data = []
    for ticker in tickers:
        bars = client.get_aggs(ticker, 1, "day", date, date)
        data.append({
            'ticker': ticker,
            'date': date,
            'open': bars[0].open,
            'high': bars[0].high,
            'low': bars[0].low,
            'close': bars[0].close,
            'volume': bars[0].volume,
            'vwap': bars[0].vwap,
        })

    df = pd.DataFrame(data)

    # Save to S3
    df.to_parquet(f"s3://swing-trading-data-lake/raw/prices/{date}/prices.parquet")

    # Save to TimescaleDB
    save_to_db(df, table='prices')

    return df

@task(retries=2)
def fetch_news_data(date: str) -> dict:
    """Fetch news headlines for last 24h."""
```

```
    from benzinga import news

    client = news.News(api_key=os.getenv("BENZINGA_API_KEY"))

    tickers = get_sp500_tickers()

    news_data = {}
    for ticker in tickers:
        headlines = client.get(
            tickers=ticker,
            date_from=(datetime.now() - timedelta(days=1)).isoformat(),
            date_to=datetime.now().isoformat()
        )

        news_data[ticker] = {
            'headlines': [h['title'] for h in headlines],
            'analyst_changes': []  # Separate API call if needed
        }

    # Save to S3
    save_json(news_data, f"s3://swing-trading-data-lake/raw/news/{date}/news.json")

    return news_data

@task
def compute_technical_features(prices: pd.DataFrame) -> pd.DataFrame:
    """Compute technical indicators."""

    features = prices.copy()

    # Group by ticker and compute
    for ticker in features['ticker'].unique():
        ticker_data = features[features['ticker'] == ticker].sort_values('date')

        ticker_data = compute_technical_indicators(ticker_data)
        ticker_data = compute_volume_features(ticker_data)
        ticker_data = compute_price_action_features(ticker_data)
        ticker_data = compute_fibonacci_levels(ticker_data)

        features[features['ticker'] == ticker] = ticker_data

    return features

@task
def compute_cross_sectional_features(prices: pd.DataFrame, date: str) -> pd.DataFrame:
    """Compute cross-sectional rankings."""

    today_data = prices[prices['date'] == date]

    cross_sectional = compute_cross_sectional_features(today_data)
    correlation_features = compute_correlation_features(prices)

    features = today_data.merge(cross_sectional, on=['date', 'ticker'])
    features = features.merge(correlation_features, on=['date', 'ticker'])

    return features
```

```
@task
def process_text_features(news_data: dict, prices: pd.DataFrame, date: str) -> pd.DataFra
    """Process text using TextSummarizerAgent."""

    agent = TextSummarizerAgent()

    # Prepare price context
    price_context = {}
    for ticker in prices['ticker'].unique():
        ticker_prices = prices[prices['ticker'] == ticker].sort_values('date')

        price_context[ticker] = {
            'return_1d': ticker_prices['close'].pct_change().iloc[-1],
            'return_5d': ticker_prices['close'].pct_change(5).iloc[-1],
            'volume_ratio': ticker_prices['volume'].iloc[-1] / ticker_prices['volume'].rc
            'date': date
        }

    # Batch process
    text_summaries = agent.batch_summarize(
        tickers=list(news_data.keys()),
        news_data=news_data,
        price_data=price_context
    )

    # Convert to DataFrame
    text_features = []
    for ticker, summary in text_summaries.items():
        features = extract_text_features(summary.dict())
        features['ticker'] = ticker
        features['date'] = date
        text_features.append(features)

    df = pd.DataFrame(text_features)

    return df

@task
def build_dynamic_graph(prices: pd.DataFrame, date: str):
    """Build correlation graph for today."""

    graph, ticker_map = build_correlation_graph(prices, date, threshold=0.3)

    # Save graph
    import torch
    torch.save({
        'graph': graph,
        'ticker_map': ticker_map
    }, f"s3://swing-trading-data-lake/processed/graphs/{date}.pt")

    return graph, ticker_map

@task(retries=1)
def run_tft_gnn_inference(features: pd.DataFrame, graph, ticker_map: dict) -> pd.DataFram
    """Run TFT-GNN model inference."""
```

```
    # Load model
    model = load_model('tft_gnn', version='latest')
    model.eval()

    # Prepare batch
    batch = prepare_tft_batch(features)

    # Inference
    with torch.no_grad():
        predictions = model(batch, graph)

    # Convert to DataFrame
    preds_df = pd.DataFrame({
        'ticker': [ticker_map[i] for i in range(len(ticker_map))],
        'tft_gnn_return': predictions['return'].cpu().numpy(),
        'tft_gnn_prob': predictions['prob_hit_long'].cpu().numpy(),
        'tft_gnn_volatility': predictions['volatility'].cpu().numpy(),
        'tft_gnn_q10': predictions['quantiles']['q10'].cpu().numpy(),
        'tft_gnn_q50': predictions['quantiles']['q50'].cpu().numpy(),
        'tft_gnn_q90': predictions['quantiles']['q90'].cpu().numpy(),
    })

    return preds_df

@task
def run_lgbm_inference(features: pd.DataFrame) -> pd.DataFrame:
    """Run LightGBM ranker."""

    model = load_model('lgbm_ranker', version='latest')

    X = features[lgbm_features]

    predictions = model.predict(X)

    preds_df = pd.DataFrame({
        'ticker': features['ticker'],
        'lgbm_score': predictions
    })

    return preds_df

@task
def run_arima_garch(prices: pd.DataFrame) -> pd.DataFrame:
    """Run ARIMA/GARCH baselines."""

    results = []

    for ticker in prices['ticker'].unique():
        ticker_prices = prices[prices['ticker'] == ticker].sort_values('date')
        returns = ticker_prices['close'].pct_change().dropna()

        arima_garch = fit_arima_garch(ticker, returns)

        results.append({
            'ticker': ticker,
```

```
            'arima_return': arima_garch['arima_return_forecast'],
            'garch_vol': arima_garch['garch_volatility_forecast'],
            'regime': arima_garch['regime'],
        })

    return pd.DataFrame(results)

@task
def detect_patterns_batch(prices: pd.DataFrame) -> pd.DataFrame:
    """Detect chart patterns for all tickers."""

    agent = PatternDetectorAgent()

    results = []

    for ticker in prices['ticker'].unique():
        ticker_prices = prices[prices['ticker'] == ticker].sort_values('date').tail(60)

        patterns = agent.detect_patterns(ticker, ticker_prices)

        patterns['ticker'] = ticker
        results.append(patterns)

    return pd.DataFrame(results)

@task
def ensemble_all_predictions(
    tft_gnn_preds: pd.DataFrame,
    lgbm_preds: pd.DataFrame,
    arima_garch: pd.DataFrame,
    patterns: pd.DataFrame,
    text_features: pd.DataFrame,
    cross_sectional: pd.DataFrame
) -> pd.DataFrame:
    """Ensemble all model outputs."""

    # Merge all features
    combined = tft_gnn_preds.copy()
    combined = combined.merge(lgbm_preds, on='ticker')
    combined = combined.merge(arima_garch, on='ticker')
    combined = combined.merge(patterns, on='ticker')
    combined = combined.merge(text_features, on='ticker')
    combined = combined.merge(cross_sectional, on='ticker')

    # Compute priority score
    macro_context = get_macro_context()  # SPY trend, VIX, etc.

    combined['priority_score'] = combined.apply(
        lambda row: compute_priority_score(
            tft_gnn_preds=row[['tft_gnn_return', 'tft_gnn_prob', 'tft_gnn_volatility']].t
            lgbm_ranks={'rank': row['lgbm_score']},
            patterns=row[['breakout_52w_high', 'pattern_confidence', 'fib_confluence']].t
            text_features=row[['sentiment_score', 'has_contrarian_signal']].to_dict(),
            cross_sectional=row[['sector_relative_strength', 'correlation_to_spy']].to_di
            macro=macro_context
        ),
```

```
        axis=1
    )

    # Save predictions
    save_to_db(combined, table='predictions')

    return combined

@task
def curate_final_trades(
    candidates: pd.DataFrame,
    date: str
) -> list[dict]:
    """Run PolicyAgent to curate final trade list."""

    agent = PolicyAgent()

    # Get portfolio state
    portfolio = get_portfolio_state()

    # Apply policy
    trades = agent.curate_trades(
        candidates=candidates.nlargest(50, 'priority_score'),
        portfolio_state=portfolio,
        risk_rules=RISK_RULES,
        date=date,
        max_trades=12
    )

    # Save recommendations
    save_recommendations(trades, date)

    return trades

@task
def generate_outputs(trades: list[dict], date: str):
    """Generate dashboard, briefing, alerts."""

    explainer = ExplainerAgent()

    # Daily brief
    market_context = get_macro_context()
    market_context['date'] = date

    brief = explainer.generate_daily_brief(trades, market_context)

    # Save outputs
    save_to_s3(brief, f"s3://swing-trading-data-lake/output/{date}/brief.txt")
    save_json(trades, f"s3://swing-trading-data-lake/output/{date}/trades.json")

    # Send alerts
    send_email_alert(trades, brief)
    post_to_slack(trades, brief)

    # Update dashboard
    update_dashboard(trades, date)
```

```python
@flow(
    name="swing_trading_daily_pipeline",
    task_runner=ConcurrentTaskRunner()
)
def daily_pipeline(date: str = None):
    """
    Daily EOD pipeline.

    Runs at 5:05 PM ET after market close.
    """

    if date is None:
        date = datetime.now().strftime("%Y-%m-%d")

    print(f"Running pipeline for {date}")

    # 1. Data ingestion (parallel)
    prices_future = fetch_market_data.submit(date)
    news_future = fetch_news_data.submit(date)

    prices = prices_future.result()
    news_data = news_future.result()

    # 2. Feature engineering (parallel)
    tech_features_future = compute_technical_features.submit(prices)
    cross_sect_future = compute_cross_sectional_features.submit(prices, date)
    text_features_future = process_text_features.submit(news_data, prices, date)
    graph_future = build_dynamic_graph.submit(prices, date)

    tech_features = tech_features_future.result()
    cross_sectional = cross_sect_future.result()
    text_features = text_features_future.result()
    graph, ticker_map = graph_future.result()

    # Merge features
    features = tech_features.merge(cross_sectional, on=['date', 'ticker'])
    features = features.merge(text_features, on=['date', 'ticker'])

    # 3. Model inference (parallel)
    tft_gnn_future = run_tft_gnn_inference.submit(features, graph, ticker_map)
    lgbm_future = run_lgbm_inference.submit(features)
    arima_future = run_arima_garch.submit(prices)
    patterns_future = detect_patterns_batch.submit(prices)

    tft_gnn_preds = tft_gnn_future.result()
    lgbm_preds = lgbm_future.result()
    arima_garch = arima_future.result()
    patterns = patterns_future.result()

    # 4. Ensemble
    candidates = ensemble_all_predictions(
        tft_gnn_preds, lgbm_preds, arima_garch, patterns, text_features, cross_sectional
    )

    # 5. Curate trades
```

```python
    trades = curate_final_trades(candidates, date)

    # 6. Generate outputs
    generate_outputs(trades, date)

    print(f"Pipeline complete. {len(trades)} trades generated.")

    return trades

# Schedule pipeline
if __name__ == "__main__":
    from prefect.deployments import Deployment
    from prefect.server.schemas.schedules import CronSchedule

    deployment = Deployment.build_from_flow(
        flow=daily_pipeline,
        name="daily-eod-pipeline",
        schedule=CronSchedule(cron="5 17 * * 1-5", timezone="America/New_York"),  # 5:05
        work_queue_name="trading-queue"
    )

    deployment.apply()
```

8.2 Execution Timeline

Daily EOD Schedule (EST):

Time	Task	Duration	Dependencies
5:05 PM	Fetch market data (OHLCV)	2 min	Market close
5:05 PM	Fetch news data	2 min	None
5:07 PM	Compute technical indicators	2 min	Market data
5:07 PM	Compute cross-sectional features	2 min	Market data
5:07 PM	Process text (LLM)	3 min	News data
5:07 PM	Build correlation graph	2 min	Market data
5:10 PM	Merge features	1 min	All features
5:11 PM	TFT-GNN inference	3 min	Features + graph
5:11 PM	LightGBM inference	1 min	Features
5:11 PM	ARIMA/GARCH inference	2 min	Market data
5:11 PM	Pattern detection	2 min	Market data
5:14 PM	Ensemble predictions	1 min	All models
5:15 PM	Priority scoring	1 min	Ensemble
5:16 PM	PolicyAgent curation (LLM)	2 min	Top candidates
5:18 PM	Generate outputs	2 min	Final trades

Time	Task	Duration	Dependencies
5:20 PM	Send alerts	1 min	Outputs

Total Pipeline Duration: ~15 minutes

9. Training & Evaluation

9.1 Training Schedule

Weekly retraining:

- **When:** Every Sunday, 2:00 AM ET
- **What:** Retrain TFT-GNN, LightGBM ranker
- **Data:** Last 3 years (rolling window)
- **Validation:** Last 6 months walk-forward

Monthly retraining:

- **When:** First Sunday of month
- **What:** Full model refit + hyperparameter tuning
- **ARIMA/GARCH:** Refit every week per ticker

9.2 Training Pipeline (Prefect)

```
@flow(name="weekly_training_pipeline")
def train_models(start_date: str, end_date: str):
    """
    Weekly retraining pipeline.
    """

    # 1. Load training data
    train_data = load_training_data(start_date, end_date)

    # 2. Compute labels
    train_data['return_5d'] = train_data.groupby('ticker')['close'].pct_change(5).shift(-
    train_data['hit_target_long'] = compute_hit_target(train_data, target_pct=0.08, stop_

    # 3. Train/val split (time-based)
    split_date = pd.to_datetime(end_date) - pd.Timedelta(days=180)

    train_set = train_data[train_data['date'] < split_date]
    val_set = train_data[train_data['date'] >= split_date]

    # 4. Train TFT-GNN
    tft_gnn_model = train_tft_gnn(train_set, val_set)

    # 5. Train LightGBM
    lgbm_model = train_lgbm_ranker(train_set, val_set)
```

```
    # 6. Evaluate
    metrics = evaluate_models(tft_gnn_model, lgbm_model, val_set)

    # 7. Save models
    save_model(tft_gnn_model, 'tft_gnn', version=datetime.now().strftime("%Y%m%d"))
    save_model(lgbm_model, 'lgbm_ranker', version=datetime.now().strftime("%Y%m%d"))

    # 8. Log to MLflow
    log_to_mlflow(metrics, models=[tft_gnn_model, lgbm_model])

    print(f"Training complete. Metrics: {metrics}")

    return metrics
```

9.3 Loss Function

```
class SwingTradingLoss(nn.Module):
    """Multi-task loss for TFT-GNN."""

    def __init__(self, weights={'return': 0.4, 'prob': 0.4, 'quantile': 0.2}):
        super().__init__()
        self.weights = weights

        self.mse = nn.MSELoss()
        self.bce = nn.BCELoss()
        self.quantile_loss = QuantileLoss(quantiles=[0.1, 0.5, 0.9])

    def forward(self, preds: dict, targets: dict) -> dict:
        """
        Args:
            preds: {
                'return': (batch,),
                'prob_hit_long': (batch,),
                'quantiles': {'q10': (batch,), 'q50': (batch,), 'q90': (batch,)}
            }
            targets: {
                'return_5d': (batch,),
                'hit_target_long': (batch,)
            }
        """

        # 1. Return regression
        return_loss = self.mse(preds['return'], targets['return_5d'])

        # 2. Hit probability classification
        prob_loss = self.bce(preds['prob_hit_long'], targets['hit_target_long'])

        # 3. Quantile regression (for uncertainty)
        quantile_losses = []
        for q, pred_q in preds['quantiles'].items():
            q_val = float(q[1:]) / 100  # 'q10' -> 0.1
            quantile_losses.append(self.quantile_loss(pred_q, targets['return_5d'], q_val

        quantile_loss = torch.mean(torch.stack(quantile_losses))
```

```
        # Total loss
        total_loss = (
            self.weights['return'] * return_loss +
            self.weights['prob'] * prob_loss +
            self.weights['quantile'] * quantile_loss
        )

        return {
            'total': total_loss,
            'return_loss': return_loss.item(),
            'prob_loss': prob_loss.item(),
            'quantile_loss': quantile_loss.item()
        }

class QuantileLoss(nn.Module):
    """Quantile regression loss."""

    def __init__(self, quantiles: list[float]):
        super().__init__()
        self.quantiles = quantiles

    def forward(self, preds: torch.Tensor, targets: torch.Tensor, quantile: float) -> tor
        """
        Pinball loss for quantile regression.
        """
        errors = targets - preds
        loss = torch.max((quantile - 1) * errors, quantile * errors)
        return loss.mean()
```

9.4 Evaluation Metrics

Offline backtest metrics:

```
def backtest_strategy(predictions: pd.DataFrame, actual_returns: pd.DataFrame) -> dict:
    """
    Walk-forward backtest.

    For each day:
    - Take top 10-15 predictions
    - Simulate trades
    - Track P&L, win rate, Sharpe, drawdown
    """

    portfolio = BacktestPortfolio(initial_capital=100000)

    dates = sorted(predictions['date'].unique())

    for date in dates:
        # Get today's recommendations
        today_preds = predictions[predictions['date'] == date].nlargest(12, 'priority_sco

        # Execute trades
        for _, row in today_preds.iterrows():
            portfolio.enter_position(
                ticker=row['ticker'],
```

```
                side=row['side'],
                size=row['position_size_pct'] / 100,
                target_pct=row['target_pct'],
                stop_pct=row['stop_pct']
            )

        # Update existing positions (check if hit target/stop)
        portfolio.update(date, actual_returns)

    # Metrics
    metrics = {
        'total_return': portfolio.total_return(),
        'sharpe_ratio': portfolio.sharpe_ratio(),
        'max_drawdown': portfolio.max_drawdown(),
        'win_rate': portfolio.win_rate(),
        'avg_profit_per_trade': portfolio.avg_profit(),
        'avg_loss_per_trade': portfolio.avg_loss(),
        'profit_factor': portfolio.profit_factor(),
        'num_trades': portfolio.num_trades(),
        'avg_hold_time_days': portfolio.avg_hold_time(),
    }

    return metrics
```

Probability calibration:

```
from sklearn.calibration import calibration_curve
import matplotlib.pyplot as plt

def evaluate_calibration(predictions: pd.DataFrame, actuals: pd.DataFrame):
    """
    Check if predicted probabilities match actual hit rates.

    If model says 70% prob, does it actually hit 70% of the time?
    """

    # Merge predictions with actual outcomes
    df = predictions.merge(actuals, on=['date', 'ticker'])

    # Calibration curve
    prob_true, prob_pred = calibration_curve(
        df['hit_target_actual'],
        df['predicted_probability'],
        n_bins=10
    )

    # Plot
    plt.figure(figsize=(8, 8))
    plt.plot(prob_pred, prob_true, marker='o', label='Model')
    plt.plot([0, 1], [0, 1], linestyle='--', label='Perfect Calibration')
    plt.xlabel('Predicted Probability')
    plt.ylabel('Actual Hit Rate')
    plt.title('Probability Calibration Curve')
    plt.legend()
    plt.savefig('calibration_plot.png')
```

```
    # Brier score (lower is better)
    from sklearn.metrics import brier_score_loss
    brier = brier_score_loss(df['hit_target_actual'], df['predicted_probability'])

    print(f"Brier Score: {brier:.4f}")

    # If miscalibrated, apply isotonic regression
    if abs(prob_true - prob_pred).mean() > 0.05:
        print("Model is miscalibrated. Applying isotonic regression...")

        from sklearn.isotonic import IsotonicRegression

        iso_reg = IsotonicRegression(out_of_bounds='clip')
        iso_reg.fit(df['predicted_probability'], df['hit_target_actual'])

        df['calibrated_probability'] = iso_reg.predict(df['predicted_probability'])

        # Re-plot
        prob_true_cal, prob_pred_cal = calibration_curve(
            df['hit_target_actual'],
            df['calibrated_probability'],
            n_bins=10
        )

        plt.plot(prob_pred_cal, prob_true_cal, marker='s', label='Calibrated Model')
        plt.legend()
        plt.savefig('calibration_plot_fixed.png')

        # Save calibrator
        import joblib
        joblib.dump(iso_reg, 'probability_calibrator.pkl')
```

Feature importance:

```
import shap

def explain_model_predictions(model, features: pd.DataFrame):
    """Use SHAP to explain feature importance."""

    # For tree models (LightGBM)
    explainer = shap.TreeExplainer(model)
    shap_values = explainer.shap_values(features)

    # Summary plot
    shap.summary_plot(shap_values, features, show=False)
    plt.savefig('shap_summary.png')

    # Feature importance ranking
    feature_importance = pd.DataFrame({
        'feature': features.columns,
        'importance': np.abs(shap_values).mean(axis=0)
    }).sort_values('importance', ascending=False)

    print("Top 20 Most Important Features:")
```

```python
    print(feature_importance.head(20))

    return feature_importance
```

9.5 Paper Trading

```python
class PaperTradingTracker:
    """Track live recommendations without real money."""

    def __init__(self):
        self.db = get_db_connection()
        self.positions = []

    def record_recommendation(self, date: str, trades: list[dict]):
        """Save today's recommendations."""

        for trade in trades:
            self.db.execute("""
                INSERT INTO paper_trades (date, ticker, side, entry_price, target_price,
                VALUES (?, ?, ?, ?, ?, ?, ?)
            """, (
                date,
                trade['ticker'],
                trade['side'],
                get_current_price(trade['ticker']),
                get_current_price(trade['ticker']) * (1 + trade['target_pct']),
                get_current_price(trade['ticker']) * (1 + trade['stop_pct']),
                trade['probability']
            ))

    def update_positions(self, date: str):
        """Check if any paper trades hit target/stop."""

        open_trades = self.db.query("""
            SELECT * FROM paper_trades
            WHERE status = 'open' AND date <= ?
        """, (date,))

        for trade in open_trades:
            current_price = get_current_price(trade['ticker'])

            if trade['side'] == 'buy':
                if current_price >= trade['target_price']:
                    self._close_trade(trade, current_price, 'target')
                elif current_price <= trade['stop_price']:
                    self._close_trade(trade, current_price, 'stop')
            else:  # sell
                if current_price <= trade['target_price']:
                    self._close_trade(trade, current_price, 'target')
                elif current_price >= trade['stop_price']:
                    self._close_trade(trade, current_price, 'stop')

            # Time-based exit (5 days)
            if (pd.to_datetime(date) - pd.to_datetime(trade['date'])).days >= 5:
                self._close_trade(trade, current_price, 'time')
```

```
    def _close_trade(self, trade, exit_price, reason):
        """Close a paper trade."""

        if trade['side'] == 'buy':
            pnl_pct = (exit_price - trade['entry_price']) / trade['entry_price']
        else:
            pnl_pct = (trade['entry_price'] - exit_price) / trade['entry_price']

        self.db.execute("""
            UPDATE paper_trades
            SET status = 'closed', exit_price = ?, exit_date = ?, pnl_pct = ?, exit_reaso
            WHERE id = ?
        """, (exit_price, datetime.now(), pnl_pct, reason, trade['id']))

    def get_performance_report(self) -> dict:
        """Generate performance report."""

        closed_trades = self.db.query("SELECT * FROM paper_trades WHERE status = 'closed'

        metrics = {
            'num_trades': len(closed_trades),
            'win_rate': (closed_trades['pnl_pct'] > 0).mean(),
            'avg_win': closed_trades[closed_trades['pnl_pct'] > 0]['pnl_pct'].mean(),
            'avg_loss': closed_trades[closed_trades['pnl_pct'] < 0]['pnl_pct'].mean(),
            'profit_factor': abs(closed_trades[closed_trades['pnl_pct'] > 0]['pnl_pct'].s
                                 closed_trades[closed_trades['pnl_pct'] < 0]['pnl_pct'].su
            'total_return': closed_trades['pnl_pct'].sum(),
        }

        # Compare predicted vs actual
        metrics['calibration_error'] = (
            closed_trades['predicted_prob'] - (closed_trades['exit_reason'] == 'target')
        ).abs().mean()

        return metrics
```

10. User Interface

10.1 FastAPI Backend

```
# api/main.py

from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from typing import List, Optional
import pandas as pd

app = FastAPI(title="Swing Trading API", version="1.0")

# CORS
app.add_middleware(
```

```
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

class TradeRecommendation(BaseModel):
    ticker: str
    side: str
    target_pct: float
    stop_pct: float
    probability: float
    priority_score: float
    position_size_pct: float
    rationale: List[str]

class DailyBrief(BaseModel):
    date: str
    market_context: dict
    brief: str
    trades: List[TradeRecommendation]

@app.get("/")
def root():
    return {"message": "Swing Trading API", "status": "running"}

@app.get("/health")
def health_check():
    return {"status": "healthy"}

@app.get("/recommendations/latest", response_model=List[TradeRecommendation])
def get_latest_recommendations():
    """Get today's trade recommendations."""

    try:
        # Query from DB
        df = pd.read_sql("""
            SELECT * FROM recommendations
            WHERE date = (SELECT MAX(date) FROM recommendations)
            ORDER BY priority_score DESC
        """, get_db_connection())

        if df.empty:
            raise HTTPException(status_code=404, detail="No recommendations for today")

        # Convert to response model
        trades = []
        for _, row in df.iterrows():
            trades.append(TradeRecommendation(
                ticker=row['ticker'],
                side=row['side'],
                target_pct=row['target_pct'],
                stop_pct=row['stop_pct'],
                probability=row['probability'],
                priority_score=row['priority_score'],
```

```
                position_size_pct=row['position_size_pct'],
                rationale=row['rationale']
            ))

        return trades

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

@app.get("/recommendations/{date}", response_model=List[TradeRecommendation])
def get_recommendations_by_date(date: str):
    """Get recommendations for a specific date."""

    df = pd.read_sql("""
        SELECT * FROM recommendations
        WHERE date = ?
        ORDER BY priority_score DESC
    """, get_db_connection(), params=(date,))

    if df.empty:
        raise HTTPException(status_code=404, detail=f"No recommendations for {date}")

    trades = [TradeRecommendation(**row) for _, row in df.iterrows()]

    return trades

@app.get("/brief/latest", response_model=DailyBrief)
def get_daily_brief():
    """Get latest daily brief."""

    # Load from S3
    latest_date = get_latest_date()

    brief_text = load_from_s3(f"s3://swing-trading-data-lake/output/{latest_date}/brief.t
    trades_json = load_from_s3(f"s3://swing-trading-data-lake/output/{latest_date}/trades

    market_context = get_macro_context()

    return DailyBrief(
        date=latest_date,
        market_context=market_context,
        brief=brief_text,
        trades=[TradeRecommendation(**t) for t in trades_json]
    )

@app.get("/explain/{ticker}")
def explain_recommendation(ticker: str, date: Optional[str] = None):
    """Get detailed explanation for a ticker recommendation."""

    if date is None:
        date = get_latest_date()

    # Query features and prediction
    row = pd.read_sql("""
        SELECT * FROM predictions
        WHERE date = ? AND ticker = ?
```

```python
    """, get_db_connection(), params=(date, ticker)).iloc[0]

    # Generate explanation using ExplainerAgent
    agent = ExplainerAgent()

    explanation = agent.explain_trade(
        trade={
            'ticker': ticker,
            'side': 'buy',  # Infer from prediction
            'target_pct': row['expected_return'],
            'probability': row['hit_prob_long']
        },
        features=row.to_dict()
    )

    return {
        'ticker': ticker,
        'date': date,
        'explanation': explanation,
        'features': {
            'expected_return': row['expected_return'],
            'probability': row['hit_prob_long'],
            'sentiment': row['sentiment_score'],
            'pattern_type': row['pattern_type'],
        }
    }

@app.get("/performance/backtest")
def get_backtest_results():
    """Get historical backtest performance."""

    metrics = load_from_s3("s3://swing-trading-data-lake/models/backtest_metrics.json")

    return metrics

@app.get("/performance/paper")
def get_paper_trading_performance():
    """Get paper trading performance."""

    tracker = PaperTradingTracker()
    metrics = tracker.get_performance_report()

    return metrics

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

10.2 React Dashboard (Simplified)

```jsx
// src/App.jsx

import React, { useEffect, useState } from 'react';
import axios from 'axios';
```

```
const API_URL = process.env.REACT_APP_API_URL || 'http://localhost:8000';

function App() {
  const [trades, setTrades] = useState([]);
  const [brief, setBrief] = useState('');
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetchRecommendations();
    fetchBrief();
  }, []);

  const fetchRecommendations = async () => {
    try {
      const response = await axios.get(`${API_URL}/recommendations/latest`);
      setTrades(response.data);
      setLoading(false);
    } catch (error) {
      console.error('Error fetching recommendations:', error);
      setLoading(false);
    }
  };

  const fetchBrief = async () => {
    try {
      const response = await axios.get(`${API_URL}/brief/latest`);
      setBrief(response.data.brief);
    } catch (error) {
      console.error('Error fetching brief:', error);
    }
  };

  if (loading) {
    return <div>Loading...</div>;
  }

  return (
    <div className="App" style={{ padding: '20px', fontFamily: 'Arial, sans-serif' }}>
      <h1>Swing Trading Recommendations</h1>

      <section style={{ marginBottom: '30px' }}>
        <h2>Daily Brief</h2>
        <div style={{
          backgroundColor: '#f5f5f5',
          padding: '15px',
          borderRadius: '8px',
          whiteSpace: 'pre-wrap'
        }}>
          {brief}
        </div>
      </section>

      <section>
        <h2>Top Trades ({trades.length})</h2>
        <table style={{ width: '100%', borderCollapse: 'collapse' }}>
          <thead>
```

```
          <tr style={{ backgroundColor: '#333', color: 'white' }}>
            <th style={{ padding: '10px', textAlign: 'left' }}>Ticker</th>
            <th>Side</th>
            <th>Target</th>
            <th>Stop</th>
            <th>Prob</th>
            <th>Priority</th>
            <th>Size</th>
            <th>Rationale
```