Market Research Agent — Price, Indicators, and News Pipeline

This project builds a lightweight research workflow for equities: we fetch historical prices, compute common technical indicators (SMA, RSI), and pull headline summaries with basic sentiment. A small JSON cache keeps runs fast and reproducible while avoiding API rate limits. The notebook(s) walk through data loading, quick EDA, feature prep, and simple evaluation.

Course: MSAAI 520-02 — Group 5
Date: October 18, 2025

Group 5 Members

- Sunitha Kosireddy
- Victor Salcedo
- Ali Azizi

Professor: Dr. Kahila Mokhtari

Our course instructor has been added as a **GitHub contributor** to this project repository

Agentic Finance — All-in-One Notebook

This notebook is a demonstration notebook showing the full workflow and plan of our project.

It summarizes the key components we developed across the Agentic Finance system, including data ingestion, analysis, orchestration, and user interface layers

3 explicit workflow patterns

- 5 working AI agents
- Full data pipeline
- Interactive UI
- Professional code structure

To run the full interactive application, please use the Gradio app located at:

/src/ui/gradio_app.py

Run it from the project root with:

python -m ui.gradio_app

This will test the agents and see live results.

GitHub Repository

GitHub: https://github.com/al1az1z1/agentic-finance

Requirements

To reproduce or run this notebook

pandas>=1.5 numpy>=1.23 requests>=2.31 yfinance>=0.2.44 gradio>=4.44 python-dotenv>=1.0 openai>=1.40

Project Path Setup (for Imports)

Purpose

- **This cell just sets the main project folder and makes sure our src/ code can be imported properly inside the notebook.**
- **It adds the project path to Python's import list so we can use things like**
- **from src.data_io.cache import load_cache without errors.**

In [27]:
```
from pathlib import Path
import sys

# Discover the repo root by walking upward until we find a folder containing "src".
ROOT = Path.cwd()
while not (ROOT / "src").exists() and ROOT.parent != ROOT:
    ROOT = ROOT.parent

# Expose the repository root on sys.path so from src... works in this session.
sys.path.insert(0, str(ROOT))  # make "src" importable

# Ensure packages (empty __init__.py files)
for p in [
    ROOT / "src",
    ROOT / "src" / "config",
    ROOT / "src" / "data_io",
    ROOT / "src" / "system",
    ROOT / "src" / "analysis",


]:
    p.mkdir(parents=True, exist_ok=True)
    (p / "__init__.py").touch(exist_ok=True)
```

Basic Feature Engineering — SMA and RSI Computation

Purpose

- We fetch SMA/RSI from Alpha Vantage when possible
- **if the API key is missing or rate-limited, we compute the same indicators locally from prices using our compute_sma and compute_rsi.**
- **We cache results so repeated calls are fast and consistent.**

In [28]:
```
# src/analysis/features.py
from __future__ import annotations
import pandas as pd
import numpy as np

def compute_sma(prices: pd.DataFrame, window: int) -> pd.Series:
    if prices is None or prices.empty:
        return pd.Series(dtype=float)
```

```
    return prices["close"].rolling(window=window).mean()

def compute_rsi(prices: pd.DataFrame, window: int = 14) -> pd.Series:
    if prices is None or prices.empty:
        return pd.Series(dtype=float)
    delta = prices["close"].diff()
    gain = np.where(delta > 0, delta, 0.0)
    loss = np.where(delta < 0, -delta, 0.0)
    gain_s = pd.Series(gain, index=prices.index)
    loss_s = pd.Series(loss, index=prices.index)
    avg_gain = gain_s.rolling(window=window).mean()
    avg_loss = loss_s.rolling(window=window).mean()
    rs = avg_gain / (avg_loss + 1e-10)
    rsi = 100 - (100 / (1 + rs))
    return rsi
```

News basic Preprocessing Prepare news for the pipeline

Make LLM I/O stable

How the orchestrator uses it

preprocess_news(df) -> Parse published_at to UTC, drop dupes by url, fill missing summary.

Why: we want consistent timestamps and no broken rows before routing

add_tags_and_numbers(df) -> Add tags from simple keyword rules and pull numeric tokens (e.g., $10, 3.5%).

Why: tags help later filtering; numbers give agents concrete context.

recent_topk(df, topk, days, required_tags) -> Keep only recent items and (optionally) require certain tags.

Why: trim the context to timely, relevant news so agents stay focused.

How the agents use it

strip_code_fences(s) -> Remove /json wrappers before json.loads.

Why: models sometimes wrap JSON; this prevents parse errors.

to_float(x) -> Best-effort number parsing (also maps words like bullish/neutral to anchors).

Why: agent prompts may yield words or percents—we standardize them.

normalize_score(v) -> Map various scales (−1..1, 0..1, %, 0..10) into −1..1.

Why: every agent's "score" becomes comparable for synthesis.

normalize_conf(v) -> Normalize any confidence-like value into 0..1 (clamped).

Why: uniform confidence makes critique/synthesis logic simpler.

pretty_json_block(obj) -> Safe, truncated JSON block for UI.

Why: readable agent outputs without overloading the notebook/app.

Result: each agent returns consistent, parseable JSON with aligned scales, making synthesis and critique reliable.

In [29]:
```python
# Second approach

from __future__ import annotations
import re
import pandas as pd
from datetime import datetime, timedelta
# from config.settings import SETTINGS

# ------------------------------
# Existing tagging / preprocessing
# ------------------------------

# Lightweight keyword rules for simple topic tagging
TAG_RULES = {
    "earnings": ["earnings", "eps", "guidance", "outlook", "quarter", "revenue"],
    "product":  ["launch", "iphone", "chip", "feature", "service"],
    "legal":    ["lawsuit", "regulator", "antitrust", "fine", "settlement"],
    "macro":    ["inflation", "rates", "fed", "recession", "gdp"]
}

def preprocess_news(df: pd.DataFrame) -> pd.DataFrame:
    if df is None or df.empty:
        return pd.DataFrame(columns=[
            "published_at","source","title","summary","url",
            "overall_sentiment","tags","numbers"
        ])

    df = df.copy()

    # Alpha Vantage format is like "20251017T200143"
    # Parse with explicit format; keep timezone-aware for safety
    df["published_at"] = pd.to_datetime(
        df["published_at"], format="%Y%m%dT%H%M%S", errors="coerce", utc=True
    )

    # Drop rows with no title/url; keep others (don't drop NaT here — the date filter happens later)
    df = df.dropna(subset=["title","url"]).drop_duplicates(subset=["url"])
    df["summary"] = df["summary"].fillna("")
    return df

def classify_tags(text: str) -> list[str]:
    text_l = text.lower()
    tags = [k for k, kws in TAG_RULES.items() if any(kw in text_l for kw in kws)]
    return tags or ["general"]

# Regex for lightweight number extraction:
# integers/floats, optional leading '$', optional trailing '%'
NUM_RE = re.compile(r'(\$?\b\d+(?:\.\d+)?%?)')

def extract_numbers(text: str) -> list[str]:

    return NUM_RE.findall(text or "")[:6] #Pull the first few numeric-looking tokens (e.g., '$10', '3.5%', '2025').

def add_tags_and_numbers(df: pd.DataFrame) -> pd.DataFrame:
    """
```

```
    Add 'tags' and 'numbers' columns using title+summary text.
    """
    if df.empty:
        return df
    df = df.copy()
    df["tags"] = (df["title"] + " " + df["summary"]).apply(classify_tags)
    df["numbers"] = (df["title"] + " " + df["summary"]).apply(extract_numbers)
    return df

def recent_topk(df: pd.DataFrame, topk: int, days: int, required_tags: list[str] | None = None) -> pd.DataFrame:
    """
    Keep only recent items (last `days`), optionally require at least one of `required_tags`,
    then return the most recent `topk`.
    """
    if df.empty:
        return df

    # Make an aware UTC cutoff; df['published_at'] is already UTC-aware
    cutoff = pd.Timestamp.now(tz="UTC") - pd.Timedelta(days=days)
    f = df[df["published_at"] >= cutoff]

    # If tag constraints are provided, we prefer rows that match them.
    if required_tags:
        want = [t.strip().lower() for t in required_tags]
        f_tags = f[f["tags"].apply(lambda ts: any(t in [x.lower() for x in ts] for t in want))]
        f = f_tags if not f_tags.empty else f

    return f.sort_values("published_at", ascending=False).head(topk)

# ----------------------------
# NEW: shared agent utilities
# ----------------------------

import json

def strip_code_fences(s: str) -> str:
    """
    Remove leading/trailing triple backticks (``` or ```json) from model text.
    This helps when models wrap JSON in code fences.
    """
    if not isinstance(s, str):
        return s
    return re.sub(r"^```(?:json)?\s*|\s*```$", "", s.strip(), flags=re.IGNORECASE)

def to_float(x, default: float = 0.0) -> float:
    """Best-effort conversion of model outputs or strings to float
        - Maps common qualitative words to representative numeric levels.
        - Falls back to 'default' if parsing fails
        """
    try:
        if isinstance(x, str):
            xs = x.strip().lower()
            # map common words to numeric anchors
            if xs in ("high", "strong", "bullish", "overbought"):
                return 0.8
            if xs in ("medium", "moderate", "neutral"):
                return 0.5
            if xs in ("low", "weak", "bearish", "oversold"):
                return 0.2
        return float(x)
```

```
    except Exception:
        return default

def clamp(x: float, lo: float, hi: float) -> float:
    """Keep a value within [lo, hi]."""
    return max(lo, min(hi, x))

def normalize_score(v: float) -> float:
    """
    Normalize arbitrary score ranges to [-1, 1].
    Heuristics:
      - If already in [-1,1], keep.
      - If in [0,1], map to [-1,1] via (v-0.5)*2.
      - If in (1,100], treat as percent.
      - If in (1,10], treat as 0-10 and map.
      - Else, clamp.
    """
    try:
        v = float(v)
    except Exception:
        return 0.0
    if -1.0 <= v <= 1.0:
        return v
    if 0.0 <= v <= 1.0:
        return (v - 0.5) * 2.0
    if 1.0 < v <= 100.0:
        v01 = v / 100.0
        return (v01 - 0.5) * 2.0
    if 1.0 < v <= 10.0:
        v01 = v / 10.0
        return (v01 - 0.5) * 2.0
    return clamp(v, -1.0, 1.0)

def normalize_conf(v) -> float:
    """Normalize any confidence-like value to [0,1].
        - Defaults to 0.7 if parsing is uncertain.
    """
    f = to_float(v, 0.7)
    if 1.0 < f <= 100.0:
        f = f / 100.0
    return clamp(f, 0.0, 1.0)

# Optional: helpers to render structured dicts into strings (for external tools)
def pretty_json_block(obj: dict, max_chars: int = 4000) -> str:
    """Return a fenced JSON markdown block, truncated for UI safety."""
    try:
        js = json.dumps(obj, ensure_ascii=False, indent=2)
    except Exception:
        js = str(obj)
    if len(js) > max_chars:
        js = js[: max_chars - 20] + "\n... (truncated)"
    return f"```json\n{js}\n```"
```

Settings & .env: one clean place for keys and knob (safety)

We use a tiny Settings object (loaded from .env) so every module like prices, news, indicators, earnings, and agents—reads config from one spot.

This keeps API keys out of code and Git, standardizes paths (data/, cache/, runs/)

In [30]:
```python
from __future__ import annotations
import os
from pathlib import Path
from dataclasses import dataclass
from dotenv import load_dotenv

def _find_project_root(start: Path) -> Path:
    """
    Walk upward to find the repo root heuristically.
    Treat a folder containing both 'src' and 'data' as the root.
    Fallback to the starting directory if not found.
    """
    for p in [start, *start.parents]:
        if (p / "src").exists() and (p / "data").exists():
            return p
    return start

# project root = repo root
if "__file__" in globals():
    ROOT = Path(__file__).resolve().parents[2]
else:
    # Notebook / REPL: start from CWD and auto-detect root
    ROOT = _find_project_root(Path.cwd())

load_dotenv(ROOT / ".env", override=False)

@dataclass(frozen=True)
class Settings:
    data_dir: Path = ROOT / "data"
    cache_dir: Path = ROOT / "data" / "cache"
    runs_dir: Path = ROOT / "data" / "runs"
    alpha_api_key: str = os.getenv("ALPHAVANTAGE_API_KEY", "BVGUKZR1MHVS0T6B")
    openai_api_key: str = os.getenv("OPENAI_API_KEY", "sk-proj-")
    news_window_days: int = 14
    topk_news: int = 5
    cache_ttl_minutes: int = 60

SETTINGS = Settings()
SETTINGS.cache_dir.mkdir(parents=True, exist_ok=True)
SETTINGS.runs_dir.mkdir(parents=True, exist_ok=True)
```

Component — Simple JSON Cache (TTL + Atomic Writes)

Purpose

Provide a lightweight on-disk cache to reduce redundant network calls and stabilize latency. Each cache entry is stored as a single JSON file with a timestamp for TTL-based expiry. Writes are atomic to avoid corruption.

Scope & Placement

Used by data fetchers (e.g., price/news downloads) prior to any external HTTP call. Implemented as simple functions for ease of reuse and testing.

Inputs / Outputs / Side Effects

- **Inputs:** `key: str` (file name stem), optional `ttl_minutes: int | None`, arbitrary JSON-serializable `data`
- **Outputs:** `load_cache` returns the stored payload (`data`) or `None` ; `save_cache` returns `None`

- **Side Effects:** Creates/updates files under `SETTINGS.cache_dir` as `<key>.json`

Behavior

- `load_cache(key, ttl_minutes=None)`
  - If file missing/corrupt → `None`
  - If `ttl_minutes is None` → returns the stored `data` (ignore age)
  - Else → returns `data` only if `(now - _ts) <= ttl_minutes * 60`
- `save_cache(key, data)`
  - Serializes as `{"_ts": epoch_seconds, "data": <payload>}`
  - Writes to a temp file then atomically replaces the target

Failure Modes & Handling

- Invalid JSON / partial writes → treated as a cache miss (returns `None`)
- Non-serializable objects → coerced via `_json_default` (ISO-8601 for dates, `str()` fallback, `repr()` last resort)

Configuration & Tunables

- `SETTINGS.cache_dir` controls storage location
- TTL per call via `ttl_minutes`; absence means "return whatever is present"

Security & Data Handling

- Do not cache secrets/PII. Payload is plain JSON on disk.

Testability

- Unit tests: miss→save→hit, TTL expiry path, corrupt file → miss, atomic replace behavior (temp file present)

In [31]:
```python
# Purpose: lightweight disk cache with TTL and atomic writes
# Context: used by data fetchers (e.g., price downloads) to avoid repeat network calls
# Notes: filenames derive from key under SETTINGS.cache_dir; payload stored as JSON

# cache.py
from __future__ import annotations
import json, time
from datetime import date, datetime
from pathlib import Path
from typing import Any
from src.data_io.cache import load_cache, save_cache
from src.config.settings import SETTINGS

def _cache_path(key: str) -> Path:
    return SETTINGS.cache_dir / f"{key}.json"

def _json_default(o: Any):
    # datetime & pandas.Timestamp (subclass of datetime) → ISO 8601
    if isinstance(o, (datetime, date)):
        return o.isoformat()
    # Fallback: make a best-effort string (covers Decimal, Path, Enum, etc.)
    try:
        return str(o)
    except Exception:
        return repr(o)
```

```python
def load_cache(key: str, ttl_minutes: int | None = None) -> Any | None:
    p = _cache_path(key)
    if not p.exists():
        return None
    try:
        obj = json.loads(p.read_text(encoding="utf-8"))
        if ttl_minutes is None:
            return obj.get("data")  # consistent: always return payload
        if (time.time() - obj.get("_ts", 0)) <= ttl_minutes * 60:
            return obj.get("data")
    except Exception:
        return None
    return None

def save_cache(key: str, data: Any) -> None:
    p = _cache_path(key)
    p.parent.mkdir(parents=True, exist_ok=True)
    tmp = p.with_suffix(p.suffix + ".tmp")
    payload = {"_ts": time.time(), "data": data}
    tmp.write_text(json.dumps(payload, ensure_ascii=False, default=_json_default), encoding="utf-8")
    tmp.replace(p)  # atomic on most OS/filesystems
```

Risk Metrics — Return/Volatility/Sharpe, Max Drawdown, VaR(5%), and Beta vs Benchmark

Purpose

Compute a small, interpretable risk summary for a ticker: average daily return, volatility, Sharpe ratio, max drawdown, 5th-percentile Value-at-Risk, and beta versus a benchmark (default S&P 500, `^GSPC`). Results are cached per `(symbol, start, end)`.

Scope & Placement

Used by reporting and decision agents to contextualize recent performance and risk. Implemented as side-effect-free helpers plus a single orchestration function, `fetch_risk_metrics`.

Inputs / Outputs / Side Effects

- **Inputs:**
  - `symbol: str` — ticker (e.g., `"AAPL"`)
  - `start: Optional[str]`, `end: Optional[str]` — ISO dates or `None` for provider defaults
  - `benchmark: str` — comparison index for beta (default `^GSPC`)
- **Outputs:** `Dict[str, Any]` with keys:
  `avg_daily_return`, `volatility`, `sharpe_ratio`, `max_drawdown`, `var_5`, `beta`
- **Side Effects:** Reads/writes a JSON cache keyed by `(symbol, start, end)` via `load_cache` / `save_cache`.

Method Overview

- **Daily returns:** `pct_change()` on close prices.
- **Volatility:** standard deviation of daily returns.
- **Sharpe (daily):** mean/volatility (no risk-free rate; downstream can annualize).
- **Max drawdown:** min of `(price / rolling_max) - 1`. Reported in percent (negative).
- **VaR(5%):** empirical 5th percentile of daily returns.
- **Beta:** covariance(asset, benchmark) / variance(benchmark), over the overlapping window.

Failure Modes & Handling

- Missing/empty price series → `{}` (empty metrics) cached and returned.
- Benchmark retrieval errors → `beta = NaN` while other metrics remain valid.
- Non-overlapping or constant benchmark returns → `beta = NaN`.

Reproducibility & Reviewability

- Cache stabilizes results for the TTL window defined in `SETTINGS`.
- Deterministic given inputs; explicit column normalization when downloading the benchmark.

In [32]:
```python
# Implementation
from __future__ import annotations
import pandas as pd
import numpy as np
import yfinance as yf
from typing import Dict, Any, Optional
from src.data_io.cache import load_cache, save_cache
from src.config.settings import SETTINGS
from src.data_io.prices import fetch_prices

# Compute daily returns from a price DataFrame.
# Returns an empty float Series if input is None/empty or lacks 'close'
def _daily_returns(df: pd.DataFrame) -> pd.Series:
    if df is None or df.empty or "close" not in df:
        return pd.Series(dtype=float)
    return df["close"].astype(float).pct_change()

# Compute max drawdown as the minimum (price/rolling_peak - 1), returned as percent.
# Returns NaN if input is invalid.
def _max_drawdown_pct(prices: pd.DataFrame) -> float:
    if prices is None or prices.empty or "close" not in prices:
        return float("nan")
    series = prices["close"].astype(float)
    roll_max = series.cummax()
    drawdown = (series / roll_max) - 1.0
    mdd = drawdown.min()
    return float(round(mdd * 100.0, 3))

# Compute beta = cov(asset, bench) / var(bench) on aligned, non-null returns.
# Returns NaN if insufficient data or zero variance.
def _beta_vs_bench(asset_rets: pd.Series, bench_rets: pd.Series) -> float:
    m = pd.concat([asset_rets, bench_rets], axis=1).dropna()
    if m.empty:
        return float("nan")
    cov = np.cov(m.iloc[:, 0], m.iloc[:, 1])[0, 1]
    var = np.var(m.iloc[:, 1])
    if var == 0:
        return float("nan")
    return float(cov / var)

# Compute a compact set of risk/return metrics for `symbol`, with caching.
def fetch_risk_metrics(symbol: str, start: Optional[str], end: Optional[str], benchmark: str = "^GSPC") -> Dict[str, Any]:
    cache_key = f"risk_{symbol}_{start}_{end}"
    cached = load_cache(cache_key, ttl_minutes=SETTINGS.cache_ttl_minutes)
    if cached is not None:
        return cached

    # Fetch prices; bail early if not available.
    prices = fetch_prices(symbol, start, end)
```

```
    if prices is None or prices.empty:
        save_cache(cache_key, {})
        return {}

    # Compute returns; if no variance (e.g., single row), return {}.
    rets = _daily_returns(prices).dropna()
    if rets.empty:
        save_cache(cache_key, {})
        return {}

    # Core metrics on asset returns (daily scale).
    mean_ret = float(rets.mean())
    vol = float(rets.std())
    sharpe = float(mean_ret / vol) if vol > 0 else float("nan")
    mdd_pct = _max_drawdown_pct(prices)
    var_5 = float(np.nanpercentile(rets.values, 5))

    # Download benchmark with explicit auto_adjust to avoid FutureWarning
    beta = float("nan")
    try:
        bench = yf.download(
            benchmark,
            start=prices["date"].min(),
            end=prices["date"].max(),
            progress=False,
            auto_adjust=False,   # <- key change
            threads=False,
        )
        # Normalize possible MultiIndex/flat columns to lower-case names.
        if isinstance(bench.columns, pd.MultiIndex):
            bench.columns = [c[0].lower() for c in bench.columns]
        else:
            bench.columns = [c.lower() for c in bench.columns]
        bench = bench.reset_index().rename(columns={"Date": "date"})
        bench["date"] = bench["date"].astype(str)
        bench_rets = bench["close"].astype(float).pct_change().dropna()
        # Align windows: use the overlapping tail of equal length.
        n = min(len(rets), len(bench_rets))
        beta = _beta_vs_bench(rets.tail(n).reset_index(drop=True), bench_rets.tail(n).reset_index(drop=True))
    except Exception:
        beta = float("nan")

    metrics = {
        "avg_daily_return": round(mean_ret, 6),
        "volatility": round(vol, 6),
        "sharpe_ratio": round(sharpe, 3) if not np.isnan(sharpe) else float("nan"),
        "max_drawdown": mdd_pct,       # percent (negative)
        "var_5": round(var_5, 6),
        "beta": round(beta, 3) if not np.isnan(beta) else float("nan"),
    }
    save_cache(cache_key, metrics)
    return metrics
```

Earnings Ingestion — Yahoo Finance `earnings_dates` with On-Disk Caching

Purpose

Fetch quarterly earnings data (EPS estimate, reported EPS, surprise %) from Yahoo Finance and normalize it for downstream analysis. Cache results per symbol to reduce repeated network calls.

Scope & Placement

Used by reporting/EDA to attach recent earnings context alongside prices/technicals. Implemented in `src/data_io/earnings.py`.

Inputs / Outputs / Side Effects

- **Inputs:** `symbol: str` (e.g., "AAPL")
- **Outputs:** `pd.DataFrame` with columns: `['date','EPS Estimate','Reported EPS','Surprise(%)']` (date as `YYYY-MM-DD`)
- **Side Effects:** Reads/writes JSON records in `SETTINGS.cache_dir` (key: `earnings_{symbol}`)

Behavior

1. Try cache (TTL from `SETTINGS.cache_ttl_minutes`); return on hit.
2. On miss, call `yfinance.Ticker(symbol).earnings_dates`.
3. Normalize the date column to `date` and ensure required columns exist.
4. Coerce `date` to string `YYYY-MM-DD`, cache as records, and return.

Failure Modes & Handling

- Provider returns `None`/empty → return typed empty frame with required columns.
- Schema variations (date as index vs column) → handled via normalization branch.
- Exceptions during fetch/parse → return empty typed frame (safe fail) and cache it.

Reproducibility & Reviewability

- Cached list-of-dicts makes reruns deterministic for the TTL window.
- Stable schema enables straightforward joins and plotting.

In [33]:

```python
# src/data_io/earnings.py
from __future__ import annotations
import pandas as pd
import yfinance as yf
from typing import Dict, Any
from src.data_io.cache import load_cache, save_cache
from src.config.settings import SETTINGS

def fetch_earnings(symbol: str) -> pd.DataFrame:
    """
    Quarterly earnings with EPS actual/estimate/surprise.
    Columns: ['date','EPS Estimate','Reported EPS','Surprise(%)']
    """
    cache_key = f"earnings_{symbol}"
    cached = load_cache(cache_key, ttl_minutes=SETTINGS.cache_ttl_minutes)
    if cached is not None:
        return pd.DataFrame(cached)

    try:
        tk = yf.Ticker(symbol)
        df = tk.earnings_dates
        if df is None or getattr(df, "empty", True):
            df = pd.DataFrame(columns=["Earnings Date","EPS Estimate","Reported EPS","Surprise(%)"])

        # Normalize column named 'Earnings Date' -> 'date'
        if "Earnings Date" in df.columns:
            df = df.reset_index(drop=True).rename(columns={"Earnings Date": "date"})
        elif df.index.name == "Earnings Date":
```

```
            df = df.reset_index().rename(columns={"Earnings Date": "date"})
        else:
            if "date" not in df.columns:
                df = df.reset_index().rename(columns={"index": "date"})

         # Ensure the required output columns exist, filling missing with None.
        keep = ["date","EPS Estimate","Reported EPS","Surprise(%)"]
        for k in keep:
            if k not in df.columns:
                df[k] = None
        df = df[keep].copy()
        df["date"] = pd.to_datetime(df["date"], errors="coerce").dt.strftime("%Y-%m-%d")
    except Exception:
        df = pd.DataFrame(columns=["date","EPS Estimate","Reported EPS","Surprise(%)"])

    save_cache(cache_key, df.to_dict(orient="records"))
    return df
```

Price Ingestion — Yahoo Finance OHLCV with On-Disk Caching

Purpose

Download OHLCV time series from Yahoo Finance (`yfinance`) and return a normalized DataFrame. Use a disk cache to avoid redundant network calls and smooth over API throttling.

Scope & Placement

Called by data preparation steps before feature engineering/EDA. Lives in `src/...` and is imported by notebooks and agents.

Inputs / Outputs / Side Effects

- **Inputs:**
  - `symbol: str` — e.g., `"AAPL"`
  - `start: str | None` — ISO-like date (e.g., `"2020-01-01"`) or `None`
  - `end: str | None` — ISO-like date or `None`
- **Outputs:** `pd.DataFrame` with columns: `date` , `open` , `high` , `low` , `close` , `adj_close` , `volume`
- **Side Effects:** Reads/writes JSON records under `SETTINGS.cache_dir` via `load_cache` / `save_cache`

Behavior

- Compose a cache key from `(symbol, start, end)` and honor `SETTINGS.cache_ttl_minutes` .
- On cache hit, materialize a DataFrame from the cached JSON records.
- On miss, call `yfinance.download` , flatten a possible MultiIndex, standardize column names, coerce `date` to string (JSON-safe), and cache the result.

Failure Modes & Handling

- Network/throttle issues → function returns whatever `yfinance` yields (may be empty); subsequent calls can hit cache if a prior success exists.
- Unknown symbols or empty ranges → valid but empty DataFrame.
- Column shape variations (e.g., MultiIndex) → flattened defensively.

Reproducibility & Reviewability

- The cached JSON (records orient) makes runs reproducible for a TTL window and simplifies inspection.
- Deterministic column naming aids downstream merging and plotting.

In [34]:
```python
# Purpose: download OHLCV from Yahoo Finance and return a normalized DataFrame with caching
# Context: called by data prep steps before features/EDA; avoids repeated network calls
# Notes: flattens MultiIndex cols, standardizes names, stores json-serializable cache

from __future__ import annotations
import pandas as pd
import yfinance as yf
from src.data_io.cache import load_cache, save_cache
from src.config.settings import SETTINGS

def fetch_prices(symbol: str, start: str | None, end: str | None) -> pd.DataFrame:
    cache_key = f"prices_{symbol}_{start}_{end}"
    cached = load_cache(cache_key, ttl_minutes=SETTINGS.cache_ttl_minutes)
    if cached is not None:
        return pd.DataFrame(cached)
    df = yf.download(symbol, start=start, end=end, progress=False)
    if isinstance(df.columns, pd.MultiIndex):
        df.columns = [c[0].lower() for c in df.columns]
    df = df.reset_index().rename(columns={
        "Date": "date", "open":"open","high":"high","low":"low","close":"close","adj close":"adj_close","volume":"volume"
    })
    df["date"] = df["date"].astype(str)
    save_cache(cache_key, df.to_dict(orient="records"))
    return df
```

Technical Indicators — Alpha Vantage SMA/RSI with Cached Local Fallback

Purpose

Retrieve daily SMA/RSI time series using Alpha Vantage when available, with a deterministic local-compute fallback (from Yahoo Finance OHLCV) to maintain functionality under API limits or missing keys.

Scope & Placement

Used by feature pipelines that require daily technical indicators. Implemented in `src/data_io/indicators.py` and consumed by analysis/agent steps.

Inputs / Outputs / Side Effects

- **Inputs:**
  - `symbol: str` — ticker (e.g., `"AAPL"`)
  - `indicator: {"SMA","RSI"}`
  - `time_period: int` — lookback window (default `14`)
- **Outputs:** `pd.DataFrame` with columns `date` and `SMA` or `RSI`, sorted ascending by `date`
- **Side Effects:** Caches JSON records under `SETTINGS.cache_dir` by `(symbol, indicator, time_period)`

Behavior

1. Attempt cache → return on hit (honors `SETTINGS.cache_ttl_minutes`).
2. If Alpha Vantage is unavailable (no key/unknown indicator) or rate-limited/error, compute locally from `fetch_prices` using `compute_sma` or `compute_rsi`.
3. On successful API call, normalize Alpha Vantage payload to a tidy DataFrame (parsed dates, numeric columns), sort ascending, cache, and return.

Failure Modes & Handling

- Network errors / quota messages / malformed payload → fallback to local compute.
- Empty price data in fallback path → return empty DataFrame.

- Non-parsable dates or numeric fields → coerced with `errors="coerce"` and dropped.

Reproducibility & Reviewability

- Cache persists list-of-dict records for deterministic reloads during the TTL window.
- Dates normalized to `datetime64[ns]`; output sorted for stable joins/plots.

In [35]:
```python
# Purpose: fetch SMA/RSI via Alpha Vantage with a cached local-compute fallback
# Context: used by feature pipelines that need daily indicators
# Notes: caches by (symbol, indicator, time_period); normalizes dates and numeric types

# src/data_io/indicators.py
from __future__ import annotations
import requests
import pandas as pd
from typing import Optional
from src.config.settings import SETTINGS
from src.data_io.prices import fetch_prices
from src.analysis.features import compute_sma, compute_rsi
from src.data_io.cache import load_cache, save_cache

BASE = "https://www.alphavantage.co/query"
KEYS = {"SMA": "Technical Analysis: SMA", "RSI": "Technical Analysis: RSI"}


# If AV isn't available (no key/limit), our code falls back to computing indicators locally from prices using our compute_sma / compute_rsi.
def _fallback_from_prices(symbol: str, indicator: str, time_period: int) -> pd.DataFrame:
    prices = fetch_prices(symbol, None, None)
    if prices is None or prices.empty:
        return pd.DataFrame()

    if indicator == "SMA":
        df = pd.DataFrame({"date": prices["date"], "SMA": compute_sma(prices, window=time_period)})
    elif indicator == "RSI":
        df = pd.DataFrame({"date": prices["date"], "RSI": compute_rsi(prices, window=time_period)})
    else:
        return pd.DataFrame()

    df["date"] = pd.to_datetime(df["date"], errors="coerce")
    df = df.dropna(subset=["date"])
    for c in df.columns:
        if c != "date":
            df[c] = pd.to_numeric(df[c], errors="coerce")
    df = df.dropna().sort_values("date", ascending=True).reset_index(drop=True)
    return df

def fetch_indicator(symbol: str, indicator: str, time_period: int = 14) -> pd.DataFrame:
    key = KEYS.get(indicator)

    # Try cache first
    cache_key = f"indicator_{symbol}_{indicator}_{time_period}"
    cached = load_cache(cache_key, ttl_minutes=SETTINGS.cache_ttl_minutes)
    if cached is not None:
        return pd.DataFrame(cached)

    if not SETTINGS.alpha_api_key or key is None:
        df = _fallback_from_prices(symbol, indicator, time_period)
        save_cache(cache_key, df.to_dict(orient="records"))
        return df
```

```
params = {
    "function": indicator,
    "symbol": symbol,
    "interval": "daily",
    "time_period": time_period,
    "series_type": "close",
    "apikey": SETTINGS.alpha_api_key,
}
try:
    resp = requests.get(BASE, params=params, timeout=30)
    resp.raise_for_status()
    data = resp.json()
    # Alpha Vantage quota message handling:
    if (not data or key not in data or not data[key] or "Note" in data or "Information" in data or "Error Message" in data):
        df = _fallback_from_prices(symbol, indicator, time_period)
        save_cache(cache_key, df.to_dict(orient="records"))
        return df
except Exception:
    df = _fallback_from_prices(symbol, indicator, time_period)
    save_cache(cache_key, df.to_dict(orient="records"))
    return df

df = pd.DataFrame.from_dict(data[key], orient="index")
df.index = pd.to_datetime(df.index, errors="coerce")
df.reset_index(inplace=True)
df = df.rename(columns={"index": "date"})
for c in df.columns:
    if c != "date":
        df[c] = pd.to_numeric(df[c], errors="coerce")
df = df.dropna(subset=["date"]).sort_values("date", ascending=True).reset_index(drop=True)
save_cache(cache_key, df.to_dict(orient="records"))
return df
```

News Ingestion — Alpha Vantage Feed with Ticker/Relevance Filtering and Cache

Purpose

Fetch symbol-specific headlines from Alpha Vantage's News Sentiment API, filter to items that explicitly mention the target ticker with sufficient relevance, and cache the normalized rows to reduce redundant calls.

Scope & Placement

Called by downstream reporting/EDA steps to attach recent headlines and high-level sentiment to a ticker. Implemented as a single function for clarity and testability.

Inputs / Outputs / Side Effects

- **Inputs:** `symbol: str` (e.g., `"AAPL"`)
- **Outputs:** `pd.DataFrame` with columns: `published_at`, `source`, `title`, `summary`, `url`, `overall_sentiment`
- **Side Effects:**
  - Reads/writes JSON records under `SETTINGS.cache_dir` using `load_cache` / `save_cache`
  - Performs a network request to Alpha Vantage on cache miss

Behavior

1. If no API key is configured, return an empty DataFrame (safe fail).
2. Check a per-symbol cache; return cached rows on hit.

3. On miss, call `NEWS_SENTIMENT` with the given ticker.
4. Keep only articles where the symbol appears in `ticker_sentiment` **and** `relevance_score ≥ 0.30`.
5. Normalize to a tidy DataFrame and cache as list-of-dict records.

Failure Modes & Handling

- Missing `feed` key or malformed payload → return empty DataFrame.
- Network errors throw from `requests.get` by default; callers can handle exceptions upstream if desired.
- Inconsistent per-item fields are handled with `.get(...)` defaults; missing values propagate as `None/NaN`.

Reproducibility & Reviewability

- Cached records (JSON) make runs deterministic for the TTL window configured in `SETTINGS`.
- Output schema is stable and designed for straightforward joins/plots.

In [36]:
```python
# Purpose: fetch and cache symbol-specific news via Alpha Vantage, filtered by relevance
# Context: called by downstream reporting/EDA to attach headlines and sentiment
# Notes: filters to items where ticker matches and relevance >= 0.30; caches by symbol

from __future__ import annotations
import os, requests, pandas as pd
from src.data_io.cache import load_cache, save_cache
from src.config.settings import SETTINGS

BASE = "https://www.alphavantage.co/query"


def fetch_news(symbol: str) -> pd.DataFrame:
    if not SETTINGS.alpha_api_key:
        return pd.DataFrame()  # safe fail
    cache_key = f"news_{symbol}"
    cached = load_cache(cache_key, ttl_minutes=SETTINGS.cache_ttl_minutes)
    if cached is not None:
        return pd.DataFrame(cached)

    params = {"function":"NEWS_SENTIMENT","tickers":symbol,"apikey":SETTINGS.alpha_api_key}
    r = requests.get(BASE, params=params, timeout=30)
    data = r.json()
    if "feed" not in data:
        return pd.DataFrame()

    rows = []
    for item in data.get("feed", []):
        tickers = item.get("ticker_sentiment", []) or []
        # keep only if our symbol is explicitly mentioned
        keep = any(t.get("ticker", "").upper() == symbol.upper() and float(t.get("relevance_score", 0) or 0) >= 0.30
                   for t in tickers)
        if not keep:
            continue

        rows.append({
            "published_at": item.get("time_published"),
            "source": item.get("source"),
            "title": item.get("title"),
            "summary": item.get("summary"),
            "url": item.get("url"),
            "overall_sentiment": item.get("overall_sentiment_label")
```

```
        })

    # ====== Forth APPROACH =====
    df = pd.DataFrame(rows)
    save_cache(cache_key, df.to_dict(orient="records"))
    return df
```

pick which lanes to run

How it works

This tiny helper tells the orchestrator which agents to run based on what data we actually have.

If there's news, we add the news lane.

If we have both prices and technicals, we add technical

If earnings are available, we add earnings

and we always run risk as a baseline check. It keeps the pipeline efficient—no agent runs without the inputs it needs.

In [37]:
```
from __future__ import annotations

def choose_agents(has_news: bool, has_prices: bool, has_technicals: bool, has_earnings: bool) -> list[str]:
    agents = []
    if has_news:
        agents.append("news")
    if has_technicals and has_prices:
        agents.append("technical")
    if has_earnings:
        agents.append("earnings")
    agents.append("risk")
    return agents
```

Run Memory (JSONL log for audits & debugging)

Purpose

We keep a lightweight run log so the orchestrator can record what happened each run (lanes used, issues from critique, confidences, timestamp)

append_memory() writes one JSON object per line to data/runs/run_notes.jsonl (auto-created via SETTINGS.runs_dir).

In [38]:
```
from __future__ import annotations
import json
from pathlib import Path
from typing import Any
from src.config.settings import SETTINGS

MEM_PATH = SETTINGS.runs_dir / "run_notes.jsonl"

def append_memory(record: dict[str, Any]) -> None:
    MEM_PATH.parent.mkdir(parents=True, exist_ok=True)
```

```
    with MEM_PATH.open("a", encoding="utf-8") as f:
        f.write(json.dumps(record, ensure_ascii=False) + "\n")
```

agents.py

Notes

This section defines the base for a multi-agent financial analysis framework used in the class project. It includes specialized agents:

- **NewsAnalysisAgent** – interprets news sentiment and market tone.
- **EarningsAnalysisAgent** – evaluates EPS trends and surprise ratios.
- **MarketSignalsAgent** – performs basic technical analysis.
- **RiskAssessmentAgent** – quantifies investment risk.
- **SynthesisAgent** – merges all signals into a BUY/HOLD/SELL recommendation.
- **CritiqueAgent** – reviews the synthesis for quality and bias.

Each agent outputs structured JSON with keys like analysis, score, confidence, and key_factors, enabling easy comparison and visualization. The modular design supports testing, replacement, and future extension with additional agents

In [40]:
```
from __future__ import annotations
import os, json
from dataclasses import dataclass
from datetime import datetime
from typing import Any, Dict, List

# Import shared helpers from analysis.text
from src.analysis.text import (
    strip_code_fences,
    to_float,
    clamp,
    normalize_score,
    normalize_conf,
)

# ---------------------------------------------------------------------------
# OpenAI client (safe stub for local/dev)
# ---------------------------------------------------------------------------
# Use the standard env var name
api_key = os.environ.get("OPENAI_API_KEY")

# Optional: print a very short prefix to help you debug locally
if api_key:
    print(f"OPENAI_API_KEY found: {api_key[:6]}***")
else:
    print("OPENAI_API_KEY NOT found! (running in MOCK mode)")
    # Don't set a fake key here; just run in mock.

# Initialize client if possible; otherwise fall back to mock
_client = None
try:
    # If you want to use the newer SDK:
    # from openai import OpenAI
    # _client = OpenAI()
    #
    # Or (legacy) openai.ChatCompletion API — but we'll stick to the new client interface:
```

```
    from openai import OpenAI
    if api_key:
        _client = OpenAI()
except Exception:
    _client = None


# ---------------------------------------------------------------------------
# Shared response container
# ---------------------------------------------------------------------------
@dataclass
class AgentResponse:
    agent_name: str
    analysis: str
    score: float
    confidence: float
    key_factors: List[str]
    timestamp: str


# ---------------------------------------------------------------------------
# BaseAgent
# ---------------------------------------------------------------------------
class BaseAgent:
    def __init__(self, agent_name: str, model: str = "gpt-4o"):
        self.agent_name = agent_name
        self.model = model

    def call_llm(self, system_prompt: str, user_message: str) -> str:
        # Mock path (no API key / no client)
        if _client is None:
            return json.dumps({
                "analysis": f"MOCK: {self.agent_name} processed.",
                "score": 0.0,
                "key_factors": ["mock"],
                "confidence": 0.7
            })
        try:
            resp = _client.chat.completions.create(
                model=self.model,
                messages=[
                    {"role": "system", "content": system_prompt},
                    {"role": "user", "content": user_message}
                ],
                temperature=0.5,
                max_tokens=1000
            )
            return resp.choices[0].message.content
        except Exception as e:
            return json.dumps({
                "analysis": f"Error: {e}",
                "score": 0.0,
                "key_factors": ["error"],
                "confidence": 0.3
            })


# ---------------------------------------------------------------------------
# News
# ---------------------------------------------------------------------------
```

```python
class NewsAnalysisAgent(BaseAgent):
    def __init__(self, model: str = "gpt-4o"):
        super().__init__("News Analysis Agent", model)
        # IMPORTANT: keep everything inside one triple-quoted string
        self.system_prompt = """You are a senior financial analyst with 15+ years of experience in equity research.

Analyze the provided news articles with focus on:
1. SENTIMENT: Quantify market sentiment from -1 (very negative) to +1 (very positive)
2. MATERIALITY: How much will this impact stock price? (high/medium/low)
3. CATALYSTS: Identify specific events that could move the stock
4. RISKS: Note any red flags or concerns mentioned

SCORING GUIDELINES:
+0.8 to +1.0: Major positive catalyst (earnings beat, breakthrough product, strategic win)
+0.4 to +0.7: Positive news (growth signals, analyst upgrades, market share gains)
-0.3 to +0.3: Neutral or mixed signals
-0.7 to -0.4: Negative news (missed targets, regulatory issues, competitive threats)
-1.0 to -0.8: Major negative catalyst (fraud, bankruptcy risk, losing key customers)

IMPORTANT:
- Use actual numbers from articles (revenue, EPS, growth rates)
- Compare to analyst expectations when mentioned
- Note if news is company-specific vs industry-wide
- Higher confidence when multiple sources agree

INSTRUCTIONS:
1. Analyze news articles objectively
2. Consider both positive and negative aspects
3. Provide a sentiment score from -1 (very negative) to +1 (very positive)
4. Identify key factors driving the sentiment
5. Assess potential stock price impact

EXAMPLE OUTPUT:
{
  "sentiment_score": 0.75,
  "analysis": "Strong positive sentiment driven by earnings beat and product launch",
  "key_factors": ["Earnings exceeded expectations", "New product well-received"],
  "confidence": 0.85
}

Return ONLY valid JSON with keys: sentiment_score, analysis, key_factors, confidence"""

    def process(self, data: Dict[str, Any]) -> AgentResponse:
        ticker = data.get('ticker', 'AAPL')
        news_articles = data.get('news', [])

        news_summary = "\n".join([
            f"- {a.get('title','')}: {a.get('description') or a.get('summary','')}"
            for a in news_articles[:5]
        ])

        user_message = f"""Analyze the following recent news about {ticker}:

{news_summary}

Provide sentiment analysis and impact assessment. Return only the JSON."""
        raw = self.call_llm(self.system_prompt, user_message)
        js = strip_code_fences(raw)

        try:
```

```
            result = json.loads(js)
            score = normalize_score(to_float(result.get('sentiment_score', 0), 0.0))
            analysis = result.get('analysis', raw)
            key_factors = result.get('key_factors', [])
            confidence = normalize_conf(result.get('confidence', 0.7))
        except json.JSONDecodeError:
            score = 0.0
            analysis = raw
            key_factors = ["Unable to parse structured response"]
            confidence = 0.6

        return AgentResponse(
            agent_name=self.agent_name,
            analysis=analysis,
            score=float(score),
            confidence=float(confidence),
            key_factors=key_factors,
            timestamp=datetime.now().isoformat()
        )


# ------------------------------------------------------------------------------
# Earnings  (COMPLETED)
# ------------------------------------------------------------------------------
class EarningsAnalysisAgent(BaseAgent):
    """Analyzes earnings reports and patterns (EPS actual vs estimate, surprise history)."""

    def __init__(self, model: str = "gpt-4o"):
        super().__init__("Earnings Analysis Agent", model)
        self.system_prompt = """You are a financial analyst specializing in earnings and fundamental analysis.

INSTRUCTIONS:
1. Analyze the earnings series objectively (EPS actual vs. estimates, surprises).
2. Identify recent beats/misses, average surprise, and beat ratio.
3. Provide a fundamental strength score from -1 (very weak) to +1 (very strong).
4. List concise key factors that justify the score.
5. Be specific with numbers when available.

EXPECTED JSON SCHEMA:
{
  "fundamental_score": float,   // -1..+1
  "analysis": string,
  "key_factors": [string],
  "confidence": float           // 0..1
}

SCORING HINTS:
- Strong positive if repeated beats, positive average surprise, improving trend.
- Negative if repeated misses, negative average surprise, deteriorating margins (if provided).
- Neutral if mixed or sparse data.

Return ONLY valid JSON with keys: fundamental_score, analysis, key_factors, confidence"""

    def process(self, data: Dict[str, Any]) -> AgentResponse:
        ticker = data.get("ticker", "UNKNOWN")
        rows = data.get("earnings", []) or []

        # Compact tabular summary to feed the model (top 8 most recent already supplied upstream)
        def row_line(r: Dict[str, Any]) -> str:
            return (
```

```
                f"- {r.get('date','?')}: estimate={r.get('EPS Estimate','n/a')}, "
                f"reported={r.get('Reported EPS','n/a')}, surprise%={r.get('Surprise(%)','n/a')}"
            )
        table = "\n".join(row_line(r) for r in rows[:12])

        user_message = f"""Company: {ticker}

Recent quarterly earnings (most recent first):
{table}

Analyze this history and return only the JSON object described in the schema."""
        raw = self.call_llm(self.system_prompt, user_message)
        js = strip_code_fences(raw)

        try:
            result = json.loads(js)
            score = normalize_score(to_float(result.get("fundamental_score", 0.0), 0.0))
            analysis = result.get("analysis", raw)
            key_factors = result.get("key_factors", [])
            confidence = normalize_conf(result.get("confidence", 0.7))
        except json.JSONDecodeError:
            score = 0.0
            analysis = raw
            key_factors = ["Unable to parse structured response"]
            confidence = 0.6

        return AgentResponse(
            agent_name=self.agent_name,
            analysis=analysis,
            score=float(score),
            confidence=float(confidence),
            key_factors=key_factors,
            timestamp=datetime.now().isoformat()
        )



# ------------------------------------------------------------------------
# Technicals
# ------------------------------------------------------------------------
class MarketSignalsAgent(BaseAgent):
    """Performs technical analysis on market data"""

    def __init__(self, model: str = "gpt-4o"):
        super().__init__("Market Signals Agent", model)
        self.system_prompt = """You are a technical analyst specializing in market signals and price patterns.

INSTRUCTIONS:
1. Analyze technical indicators objectively
2. Assess technical strength from -1 (very bearish) to +1 (very bullish)
3. Identify support/resistance levels
4. Evaluate trend direction and momentum
5. Consider volume patterns

EXAMPLE OUTPUT:
{
  "technical_score": 0.65,
  "analysis": "Bullish technical setup with price above key moving averages",
  "key_factors": ["Price above 50-day MA", "RSI indicates strength", "Volume confirming uptrend"],
  "confidence": 0.75
```

```
}

Return ONLY valid JSON with keys: technical_score, analysis, key_factors, confidence"""

    def process(self, data: Dict[str, Any]) -> AgentResponse:
        ticker = data.get('ticker', 'UNKNOWN')
        technicals = data.get('technicals', {})

        technical_summary = f"""
Ticker: {ticker}
Current Price: ${technicals.get('current_price', 'N/A')}
50-day MA: ${technicals.get('ma_50', 'N/A')}
200-day MA: ${technicals.get('ma_200', 'N/A')}
RSI: {technicals.get('rsi', 'N/A')}
MACD: {technicals.get('macd', 'N/A')}
Volume: {technicals.get('volume', 'N/A')} (Avg: {technicals.get('avg_volume', 'N/A')})
Support: ${technicals.get('support', 'N/A')}
Resistance: ${technicals.get('resistance', 'N/A')}
"""

        user_message = f"""Analyze the following technical data for {ticker}:

{technical_summary}

Assess technical strength and price momentum. Return only the JSON described above."""
        raw = self.call_llm(self.system_prompt, user_message)
        js = strip_code_fences(raw)

        try:
            result = json.loads(js)
            score = normalize_score(to_float(result.get('technical_score', 0), 0.0))
            analysis = result.get('analysis', raw)
            key_factors = result.get('key_factors', [])
            confidence = normalize_conf(result.get('confidence', 0.7))
        except json.JSONDecodeError:
            score = 0.0
            analysis = raw
            key_factors = ["Unable to parse structured response"]
            confidence = 0.6

        return AgentResponse(
            agent_name=self.agent_name,
            analysis=analysis,
            score=float(score),
            confidence=float(confidence),
            key_factors=key_factors,
            timestamp=datetime.now().isoformat()
        )

# ------------------------------------------------------------------------------
# Risk  (COMPLETED)
# ------------------------------------------------------------------------------
class RiskAssessmentAgent(BaseAgent):
    """Assesses investment risk and portfolio fit"""

    def __init__(self, model: str = "gpt-4o"):
        super().__init__("Risk Assessment Agent", model)
        self.system_prompt = """You are a risk management analyst specializing in portfolio risk assessment.

INSTRUCTIONS:
```

```
1. Analyze risk metrics objectively.
2. Provide a risk level score from 0 (very low risk) to 1 (very high risk).
3. Identify key risk drivers (beta, volatility, VaR, Sharpe, max drawdown, concentration/correlation).
4. Explain portfolio implications and any risk mitigants.

EXPECTED JSON SCHEMA:
{
  "risk_score": float,      // 0..1
  "analysis": string,
  "key_factors": [string],
  "confidence": float       // 0..1
}

GUIDANCE:
- Higher beta/volatility/drawdown/VaR => higher risk_score.
- Higher Sharpe => lowers effective risk_score (risk-adjusted).
- Lack of data => moderate confidence; be explicit.

Return ONLY valid JSON with keys: risk_score, analysis, key_factors, confidence"""

    def process(self, data: Dict[str, Any]) -> AgentResponse:
        ticker = data.get('ticker', 'UNKNOWN')
        risk_data = data.get('risk_metrics', {}) or {}

        # Build a compact, explicit summary. We pass both short-term and full stats if provided.
        risk_summary = f"""
Ticker: {ticker}
Beta: {risk_data.get('beta', 'N/A')}
Volatility (30-day): {risk_data.get('volatility', 'N/A')}%
Sharpe Ratio: {risk_data.get('sharpe_ratio', 'N/A')}
Max Drawdown (%): {risk_data.get('max_drawdown', 'N/A')}
Value at Risk (5% daily return): {risk_data.get('var_5', 'N/A')}
Sector Correlation: {risk_data.get('sector_correlation', 'N/A')}
P/E Ratio: {risk_data.get('pe_ratio', 'N/A')}

# Extended (may be None):
Avg Daily Return: {risk_data.get('avg_daily_return', 'N/A')}
Volatility (full window): {risk_data.get('volatility_full', 'N/A')}
"""

        user_message = f"""Analyze the following risk metrics and return only the JSON per schema:

{risk_summary}

Give a 0..1 risk_score, analysis, key_factors (bullet-style phrases), and confidence."""
        raw = self.call_llm(self.system_prompt, user_message)
        js = strip_code_fences(raw)

        try:
            result = json.loads(js)

            # Keep 0..1 semantics but normalize/clamp
            risk01 = to_float(result.get('risk_score', 0.5), 0.5)
            if 1.0 < risk01 <= 100.0:
                risk01 = risk01 / 100.0
            elif 1.0 < risk01 <= 10.0:
                risk01 = risk01 / 10.0
            risk01 = clamp(risk01, 0.0, 1.0)

            score = risk01
```

```
            analysis = result.get('analysis', raw)
            key_factors = result.get('key_factors', [])
            confidence = normalize_conf(result.get('confidence', 0.8))
        except json.JSONDecodeError:
            score = 0.5
            analysis = raw
            key_factors = ["Unable to parse structured response"]
            confidence = 0.6

        return AgentResponse(
            agent_name=self.agent_name,
            analysis=analysis,
            score=float(score),
            confidence=float(confidence),
            key_factors=key_factors,
            timestamp=datetime.now().isoformat()
        )


# ------------------------------------------------------------------------------
# Synthesis
# ------------------------------------------------------------------------------
class SynthesisAgent(BaseAgent):
    """Combines insights from all agents into final recommendation"""

    def __init__(self, model: str = "gpt-4o"):
        super().__init__("Research Synthesis Agent", model)
        self.system_prompt = """You are a senior investment analyst who synthesizes multiple analyses into actionable recommendations.

INSTRUCTIONS:
1. Review all agent analyses objectively
2. Weigh different factors appropriately
3. Provide clear investment recommendation (STRONG BUY, BUY, HOLD, SELL, STRONG SELL)
4. State confidence level (0 to 1)
5. Summarize key reasoning
6. Note important risks

EXAMPLE OUTPUT:
{
  "recommendation": "BUY",
  "confidence": 0.78,
  "analysis": "Strong fundamentals and positive technical signals support a buy recommendation despite moderate risk",
  "key_points": ["Earnings beat expectations", "Technical breakout", "Acceptable risk profile"],
  "risks": ["Market volatility", "Sector headwinds"]
}

Return ONLY valid JSON with keys: recommendation, confidence, analysis, key_points, risks"""

    def process(self, agent_responses: List[AgentResponse]) -> AgentResponse:
        analyses_summary = "\n\n".join([
            f"{resp.agent_name}:\n"
            f"Score: {resp.score}\n"
            f"Analysis: {resp.analysis}\n"
            f"Key Factors: {', '.join(resp.key_factors)}"
            for resp in agent_responses
        ])

        user_message = f"""Synthesize the following analyses into a final investment recommendation:

{analyses_summary}
```

```
Provide a comprehensive investment recommendation with supporting reasoning. Return only the JSON."""
        raw = self.call_llm(self.system_prompt, user_message)
        js = strip_code_fences(raw)

        try:
            result = json.loads(js)
            recommendation = str(result.get('recommendation', 'HOLD')).upper()
            analysis = result.get('analysis', raw)
            key_factors = result.get('key_points', [])
            confidence = normalize_conf(result.get('confidence', 0.7))

            rec_to_score = {
                'STRONG BUY': 1.0,
                'BUY': 0.6,
                'HOLD': 0.0,
                'SELL': -0.6,
                'STRONG SELL': -1.0
            }
            score = rec_to_score.get(recommendation, 0.0)
        except json.JSONDecodeError:
            score = 0.0
            analysis = raw
            key_factors = ["Unable to parse structured response"]
            confidence = 0.6

        return AgentResponse(
            agent_name=self.agent_name,
            analysis=analysis,
            score=float(score),
            confidence=float(confidence),
            key_factors=key_factors,
            timestamp=datetime.now().isoformat()
        )


# ------------------------------------------------------------------------------
# Critique
# ------------------------------------------------------------------------------
class CritiqueAgent(BaseAgent):
    """Reviews and validates analysis quality"""

    def __init__(self, model: str = "gpt-4o-mini"):
        super().__init__("Critique & Validation Agent", model)
        self.system_prompt = """You are a critique analyst who reviews investment recommendations for biases, logical errors, and completeness.

INSTRUCTIONS:
1. Review the synthesis objectively
2. Identify logical inconsistencies
3. Detect potential biases
4. Note missing considerations
5. Assess data quality
6. Recommend confidence adjustments

EXAMPLE OUTPUT:
{
  "quality_score": 0.82,
  "issues_found": ["Limited macroeconomic analysis"],
  "suggestions": ["Consider Federal Reserve policy impact", "Add sector comparison"],
  "adjusted_confidence": 0.75
```

```
}

Return ONLY valid JSON with keys: quality_score, issues_found, suggestions, adjusted_confidence"""

    def process(self, synthesis_response: AgentResponse) -> AgentResponse:
        user_message = f"""Review this investment analysis for quality and completeness:

Recommendation: {synthesis_response.analysis}
Confidence: {synthesis_response.confidence}
Key Factors: {', '.join(synthesis_response.key_factors)}

Identify any issues, biases, or missing elements. Return only the JSON."""
        raw = self.call_llm(self.system_prompt, user_message)
        js = strip_code_fences(raw)

        try:
            result = json.loads(js)
            quality_score = to_float(result.get('quality_score', 0.7), 0.7)
            # normalize 0..10 or 0..100 to 0..1 (display-style)
            if 1.0 < quality_score <= 10.0:
                quality_score = quality_score / 10.0
            elif 10.0 < quality_score <= 100.0:
                quality_score = quality_score / 100.0
            quality_score = clamp(quality_score, 0.0, 1.0)

            issues = result.get('issues_found', [])
            suggestions = result.get('suggestions', [])
            adjusted_confidence = normalize_conf(
                result.get('adjusted_confidence', synthesis_response.confidence)
            )

            analysis = f"Quality Score: {quality_score}\n"
            if issues:
                analysis += f"Issues Found: {', '.join(issues)}\n"
            if suggestions:
                analysis += f"Suggestions: {', '.join(suggestions)}"

            key_factors = issues if issues else ["No major issues found"]
        except json.JSONDecodeError:
            quality_score = 0.7
            analysis = raw
            adjusted_confidence = synthesis_response.confidence
            key_factors = ["No major issues found"]

        return AgentResponse(
            agent_name=self.agent_name,
            analysis=analysis,
            score=float(quality_score),
            confidence=float(adjusted_confidence),
            key_factors=key_factors,
            timestamp=datetime.now().isoformat()
        )
```

```
OPENAI_API_KEY found: sk-pro***
```

Orchestrator -> run the whole agentic pipeline, end-to-end

We build one “traffic controller” that runs the full flow for a ticker:

fetch data (prices, news, earnings, risk), clean and filter news,

Preprocess news (clean text, add tags/numbers) and retrieve the most recent, relevant headlines,

Routedecide which agent lanes to run (news / technical / earnings / risk),

Run agents for each lane and collect their JSON outputs.

Synthesize a first pass (synth_v1), then critique it

If the critique flags low quality or data issues, optimize with a second pass (synth_v2) using the critique as explicit feedback.

Save memory (light for the run) and package evidence DataFrames for the UI.

Result: every agent returns simple, parseable JSON with aligned scales, so synthesis and critique stay reliable. When the optimizer runs, the final answer is synth_v2; otherwise we keep synth_v1.

In [41]:
```python
from __future__ import annotations

from dataclasses import dataclass
from typing import Any, Dict, List
from datetime import datetime, timezone
import time
import json

import pandas as pd
from pandas import DataFrame

from src.config.settings import SETTINGS
from src.data_io.prices import fetch_prices
from src.data_io.news import fetch_news
from src.data_io.indicators import fetch_indicator
from src.data_io.earnings import fetch_earnings
from src.data_io.risk import fetch_risk_metrics
from src.analysis.text import preprocess_news, add_tags_and_numbers, recent_topk
from src.system.router import choose_agents
from src.system.memory import append_memory
from src.agents import (
    NewsAnalysisAgent,
    MarketSignalsAgent,
    RiskAssessmentAgent,
    SynthesisAgent,
    CritiqueAgent,
    AgentResponse,
    EarningsAnalysisAgent,
)

# ------------- helpers -------------
def _as_text(x):
    """We coerce any object into a readable string (pretty JSON for dict/list)."""
    if x is None:
        return ""
    if isinstance(x, (dict, list)):
        try:
            return json.dumps(x, ensure_ascii=False, indent=2)
        except Exception:
            return str(x)
    return str(x)
```

```python
def _as_list_of_text(x):
    """We normalize arbitrary input into a non-null list of strings."""
    if isinstance(x, list):
        return [_as_text(i) for i in x]
    if x is None:
        return []
    return [_as_text(x)]

def now_utc_iso() -> str:
    """We return a stable, timezone-aware timestamp for logs and memory."""
    return datetime.now(timezone.utc).isoformat()

# We use a small stagger to be kind to API rate limits.
_NET_STAGGER = float(getattr(SETTINGS, "net_stagger_secs", 0.5))

@dataclass
class OrchestratorResult:
    """We bundle everything the UI needs after one pipeline run."""
    plan: List[str]
    evidence: Dict[str, DataFrame]
    agent_outputs: List[AgentResponse]
    final: AgentResponse
    critique: AgentResponse


def run_pipeline(
    symbol: str,
    start: str | None,
    end: str | None,
    required_tags: list[str] | None = None
) -> OrchestratorResult:
    # We keep the plan visible so the UI can show progress/explain steps.
    plan = [
        "fetch_prices", "fetch_news", "fetch_earnings", "fetch_risk",
        "preprocess", "classify_extract", "retrieve_topk",
        "route", "run_agents", "synthesize", "critique", "save_memory"
    ]

    # ---------------- 1) FETCH (staggered) ----------------
    prices = fetch_prices(symbol, start, end);                      time.sleep(_NET_STAGGER)
    news   = fetch_news(symbol);                                     time.sleep(_NET_STAGGER)
    earn_df = fetch_earnings(symbol);                                time.sleep(_NET_STAGGER)
    risk_ingested = fetch_risk_metrics(symbol, start, end);          time.sleep(_NET_STAGGER)

    # ---------------- 2) PREPROCESS NEWS ----------------
    # We clean the articles and add lightweight tags/numbers for filtering.
    news_pp = add_tags_and_numbers(preprocess_news(news))

    # ---------------- 3) RETRIEVAL ----------------
    # We keep a small, recent slice for agents to read.
    top_news = recent_topk(
        news_pp,
        topk=SETTINGS.topk_news,
        days=SETTINGS.news_window_days,
        required_tags=required_tags,
    )

    # ---------------- 4) ROUTE PRIMERS ----------------
    has_news     = not top_news.empty
```

```
has_prices   = not prices.empty
has_earnings = (earn_df is not None) and (not earn_df.empty)

# We try indicators if we have prices or a provider key.
attempt_technicals = has_prices or bool(SETTINGS.alpha_api_key)

# ---------------- 5) INDICATORS (conditional) ----------------
rsi = sma20 = sma50 = sma200 = pd.DataFrame()
if attempt_technicals:
    rsi    = fetch_indicator(symbol, "RSI", 14); time.sleep(_NET_STAGGER)
    sma20  = fetch_indicator(symbol, "SMA", 20); time.sleep(_NET_STAGGER)
    sma50  = fetch_indicator(symbol, "SMA", 50); time.sleep(_NET_STAGGER)
    sma200 = fetch_indicator(symbol, "SMA", 200); time.sleep(_NET_STAGGER)

has_technicals = (not rsi.empty) or (not sma20.empty) or (not sma50.empty) or (not sma200.empty)

# We let the router decide which lanes to run (news/technical/earnings/risk).
lanes = choose_agents(has_news, has_prices, has_technicals, has_earnings)

# ---------------- 6) RUN AGENTS ----------------
outputs: List[AgentResponse] = []

# NEWS
if "news" in lanes and has_news:
    # We map to the keys the NewsAnalysisAgent expects.
    news_payload_records = (
        top_news
        .rename(columns={"summary": "description"})
        .loc[:, ["title", "description", "source", "url", "published_at"]]
        .to_dict(orient="records")
    )
    outputs.append(NewsAnalysisAgent().process({"ticker": symbol, "news": news_payload_records}))

# TECHNICALS
if "technical" in lanes and (has_technicals or has_prices):
    # We compute a tiny snapshot of technical state.
    current_price = float(prices["close"].iloc[-1]) if has_prices else None
    volume = int(prices["volume"].iloc[-1]) if has_prices else None
    avg_volume = int(prices["volume"].tail(20).mean()) if has_prices else None

    technicals = {
        "current_price": current_price,
        "rsi": (float(rsi["RSI"].iloc[-1]) if not rsi.empty else None),
        "ma_50": (float(sma50["SMA"].iloc[-1]) if not sma50.empty else
                  (float(sma20["SMA"].iloc[-1]) if not sma20.empty else None)),
        "ma_200": (float(sma200["SMA"].iloc[-1]) if not sma200.empty else None),
        "macd": None,      # reserved for future
        "volume": volume,
        "avg_volume": avg_volume,
        "support": None,   # reserved for future
        "resistance": None # reserved for future
    }
    outputs.append(MarketSignalsAgent().process({"ticker": symbol, "technicals": technicals}))

# EARNINGS
if "earnings" in lanes and has_earnings:
    earn_payload = {
        "ticker": symbol,
        "earnings": (
            earn_df.sort_values("date", ascending=False)
```

```
                    .head(8)
                    .to_dict(orient="records")
            )
        }
        outputs.append(EarningsAnalysisAgent().process(earn_payload))

    # RISK (merge ingestion + a quick realized 30d vol for the UI)
    vol_30d = float(prices["close"].pct_change().tail(30).std() * 100) if has_prices else None
    risk_payload = {
        "ticker": symbol,
        "risk_metrics": {
            "beta":              risk_ingested.get("beta"),
            "volatility":        vol_30d,                       # short-term display (%)
            "var_5":             risk_ingested.get("var_5"),
            "sharpe_ratio":      risk_ingested.get("sharpe_ratio"),
            "max_drawdown":      risk_ingested.get("max_drawdown"),
            "sector_correlation": None,
            "pe_ratio":          None,
            "avg_daily_return":  risk_ingested.get("avg_daily_return"),
            "volatility_full":   risk_ingested.get("volatility"),
        }
    }
    outputs.append(RiskAssessmentAgent().process(risk_payload))

    # ---------------- 7) SYNTHESIZE + CRITIQUE ----------------
    synth_v1 = SynthesisAgent().process(outputs)     # first pass
    crit     = CritiqueAgent().process(synth_v1)     # critique of first pass

    # We gate synth_v2 behind a simple rule to avoid unnecessary extra calls.
    needs_rerun = (crit.score < 0.90) or (
        "data quality" in " ".join(_as_list_of_text(crit.key_factors)).lower()
    )

    synth_final = synth_v1  # default to v1 unless we improve it
    synth_v2 = None         # we keep a handle for telemetry/UI if needed

    if needs_rerun:
        # We turn the critique into an explicit feedback message the synthesizer can read.
        critique_feedback = AgentResponse(
            agent_name="Critique Feedback",
            analysis=_as_text(synth_v1.analysis) + "\n\n[CRITIQUE]\n" + _as_text(crit.analysis),
            score=crit.score,
            confidence=crit.confidence,
            key_factors=_as_list_of_text(crit.key_factors),
            timestamp=now_utc_iso()
        )
        # We re-run synthesis with the feedback appended to the agent outputs.
        synth_v2_inputs = outputs + [critique_feedback]
        synth_v2 = SynthesisAgent().process(synth_v2_inputs)
        synth_final = synth_v2

    # ---------------- 8) MEMORY ----------------
    # We store whether the optimizer path ran and both confidences for later review.
    append_memory({
        "ticker": symbol,
        "lanes": lanes,
        "issues": crit.key_factors,
        "final_confidence_v1": synth_v1.confidence,
        "final_confidence_v2": (synth_v2.confidence if synth_v2 else None),
        "optimizer_triggered": bool(needs_rerun),
```

```
        "timestamp": now_utc_iso()
    })

    # ---------------- 9) EVIDENCE FOR UI ----------------
    earn_evidence = (
        earn_df.sort_values("date", ascending=False).head(8)
        if has_earnings else pd.DataFrame()
    )
    risk_evidence = pd.DataFrame([risk_payload["risk_metrics"]])

    evidence = {
        "top_news": top_news,
        "prices_tail": prices.tail(5),
        "earnings_head": earn_evidence,
        "risk_metrics": risk_evidence,
    }

    # We add the initial synthesis as a separate output so the UI can compare v1 vs final.
    outputs.append(AgentResponse(
        agent_name="Initial Synthesis",
        analysis=_as_text(synth_v1.analysis),
        score=float(synth_v1.score),
        confidence=float(synth_v1.confidence),
        key_factors=_as_list_of_text(synth_v1.key_factors),
        timestamp=synth_v1.timestamp
    ))

    return OrchestratorResult(plan, evidence, outputs, synth_final, crit)
```

Demonstration Notebook — Agentic Workflows in Action

- **This notebook section demonstrates how the full agentic pipeline works in practice.**
- **We already have the real orchestration logic defined above (run_pipeline in the orchestrator module), which runs automatically inside the system and the Gradio app.**
- **However, in a notebook we want to visually and interactively show what happens inside that pipeline — step by step — for clarity, debugging, and presentation**

In [47]:
```
# src/system/workflows_offline.py
# Offline workflows: prefer CSV prices (from UI saves or mock dir) over APIs.
from __future__ import annotations

# stdlib
import json, time, traceback, os
from datetime import datetime, timedelta, date
from typing import List, Dict, Optional
from pathlib import Path

# third-party
import pandas as pd
import numpy as np

# project
from src.config.settings import SETTINGS
from src.data_io.news import fetch_news
from src.data_io.earnings import fetch_earnings
from src.data_io.risk import fetch_risk_metrics
from src.analysis.text import preprocess_news, add_tags_and_numbers, recent_topk
```

```python
from src.agents import (
    NewsAnalysisAgent,
    MarketSignalsAgent,
    RiskAssessmentAgent,
    EarningsAnalysisAgent,
    AgentResponse,
)
from src.system.orchestrator import run_pipeline, OrchestratorResult

# ──────────────────────────────────────────────────────────────────────────────
# CSV-first prices loader + monkey patch
# ──────────────────────────────────────────────────────────────────────────────

def _as_text(x):
    if x is None:
        return ""
    if isinstance(x, (dict, list)):
        try:
            return json.dumps(x, ensure_ascii=False, indent=2)
        except Exception:
            return str(x)
    return str(x)

def _print_kv(k: str, v) -> None:
    print(f"  {k:<18} {v}")

def _normalize_text(s: str) -> str:
    s = _as_text(s).strip()
    if s.startswith("```"):
        s = s.strip("`").strip()
    return " ".join(s.split())

def _coerce_prices_df(df: pd.DataFrame) -> pd.DataFrame:
    """Normalize columns and types: require ['date','close','volume']."""
    if df is None or df.empty:
        return pd.DataFrame(columns=["date","close","volume"])

    df = df.copy()

    # Find a date-like column
    date_cols = [c for c in df.columns if str(c).lower() in {"date","datetime","timestamp"}]
    if not date_cols:
        # attempt to infer if index is datetime-like
        if isinstance(df.index, pd.DatetimeIndex):
            df = df.reset_index().rename(columns={"index":"date"})
            date_cols = ["date"]
        else:
            # give up
            return pd.DataFrame(columns=["date","close","volume"])

    date_col = date_cols[0]
    df["date"] = pd.to_datetime(df[date_col]).dt.tz_localize(None)

    # Close / adj close candidates
    close_col = None
    for c in ["close","Close","adj_close","Adj Close","adjclose"]:
        if c in df.columns:
            close_col = c
            break
    if close_col is None:
```

```
        # sometimes 'price' or 'close_price'
        for c in df.columns:
            if "close" in str(c).lower() or str(c).lower() == "price":
                close_col = c
                break
    if close_col is None:
        return pd.DataFrame(columns=["date","close","volume"])

    # Volume
    vol_col = None
    for c in ["volume","Volume","vol","Vol"]:
        if c in df.columns:
            vol_col = c
            break
    if vol_col is None:
        # allow missing; fill later
        df["volume"] = np.nan
    else:
        df["volume"] = pd.to_numeric(df[vol_col], errors="coerce")

    df["close"] = pd.to_numeric(df[close_col], errors="coerce")
    df = df.loc[~df["close"].isna()].drop_duplicates(subset=["date"]).sort_values("date")
    return df[["date","close","volume"]]

def _filter_range(df: pd.DataFrame, start: str, end: str) -> pd.DataFrame:
    if df.empty:
        return df
    s = pd.to_datetime(start)
    e = pd.to_datetime(end) + pd.Timedelta(days=1)  # inclusive end
    return df[(df["date"] >= s) & (df["date"] < e)].reset_index(drop=True)

def _find_latest_ui_prices_csv(symbol: str) -> Optional[Path]:
    """
    Look under data/runs/ui_runs/* for the newest folder matching the symbol prefix
    and return its prices.csv if present.
    """
    ui_root = SETTINGS.runs_dir / "ui_runs"
    if not ui_root.exists():
        return None
    # candidates like AAPL_20241020_101010
    candidates = [p for p in ui_root.iterdir() if p.is_dir() and p.name.upper().startswith(symbol.upper() + "_")]
    if not candidates:
        return None
    candidates.sort(key=lambda p: p.stat().st_mtime, reverse=True)
    for run_dir in candidates:
        p = run_dir / "prices.csv"
        if p.exists():
            return p
    return None

def _mock_prices_from_dir(symbol: str) -> Optional[Path]:
    """
    Optional project-level mock file, e.g. data/mock/prices/AAPL.csv
    """
    base = getattr(SETTINGS, "data_dir", Path("data"))
    for rel in [
        Path("mock/prices") / f"{symbol.upper()}.csv",
        Path("mocks/prices") / f"{symbol.upper()}.csv",
        Path("prices") / f"{symbol.upper()}.csv",
    ]:
```

```
        p = Path(base) / rel
        if p.exists():
            return p
    return None

def _load_prices_from_csv(symbol: str, start: str, end: str) -> pd.DataFrame:
    """
    Try (1) latest UI run prices.csv, then (2) data/mock/prices/SYMBOL.csv.
    Return filtered, normalized DataFrame with ['date','close','volume'].
    """
    path = _find_latest_ui_prices_csv(symbol) or _mock_prices_from_dir(symbol)
    if path is None:
        return pd.DataFrame(columns=["date","close","volume"])

    try:
        raw = pd.read_csv(path)
    except Exception:
        return pd.DataFrame(columns=["date","close","volume"])

    df = _coerce_prices_df(raw)
    df = _filter_range(df, start, end)
    return df

def _synthetic_prices(symbol: str, start: str, end: str) -> pd.DataFrame:
    """Deterministic synthetic series (seeded by symbol) for demos without CSV."""
    rng = np.random.default_rng(abs(hash(symbol)) % (2**32))
    idx = pd.date_range(start, end, freq="B")  # business days
    if len(idx) == 0:
        return pd.DataFrame(columns=["date","close","volume"])

    base = 100 + (abs(hash(symbol)) % 500) / 10.0  # 100..150-ish base
    steps = rng.normal(0, 0.01, size=len(idx))  # ~1% daily std
    prices = base * np.exp(np.cumsum(steps))
    volume = rng.integers(1_000_000, 10_000_000, size=len(idx))

    df = pd.DataFrame({"date": idx, "close": prices, "volume": volume})
    return df

def fetch_prices_offline(symbol: str, start: str, end: str) -> pd.DataFrame:
    """
    CSV-first price loader. If not found, generates synthetic data so
    downstream agents/risk have something useful to work with.
    """
    df = _load_prices_from_csv(symbol, start, end)
    if df.empty:
        df = _synthetic_prices(symbol, start, end)
        df = _filter_range(df, start, end)
    return df

def enable_offline_prices_monkey_patch() -> None:
    """
    Patch src.data_io.prices.fetch_prices → fetch_prices_offline so any call
    (including inside run_pipeline) uses CSV/synthetic instead of APIs.
    """
    try:
        import src.data_io.prices as prices_mod
        prices_mod.fetch_prices = fetch_prices_offline  # <-- the patch
        print("[offline] Patched src.data_io.prices.fetch_prices → CSV/synthetic loader.")
    except Exception as e:
        print(f"[offline] WARNING: Could not monkey-patch fetch_prices: {e}")
```

```python
# Enable at import time so callers don’t have to remember.
enable_offline_prices_monkey_patch()

# ──────────────────────────────────────────────────────────────────────────────
# WORKFLOW 1: Prompt chaining (News focused; unchanged except prices aren’t needed)
# ──────────────────────────────────────────────────────────────────────────────

def run_prompt_chaining_workflow(
    symbol: str,
    start: str,
    end: str,
    required_tags: list[str] | None = None
) -> AgentResponse:
    print("\n" + "=" * 80)
    print("WORKFLOW PATTERN 1: PROMPT CHAINING")
    print("=" * 80)
    print(f"Analyzing: {symbol} | Period: {start} → {end}")
    print("=" * 80)

    if getattr(SETTINGS, "skip_news", False):
        print("\n[Notice] News fetching is disabled by SETTINGS.skip_news=True.")
        return AgentResponse(
            agent_name="News Analysis Agent",
            analysis="News workflow skipped by configuration.",
            score=0.0,
            confidence=0.95,
            key_factors=["skip_news=True"],
            timestamp=datetime.now().isoformat()
        )

    # 1) Ingest
    print("\n┌─ STEP 1/5: INGEST ──────────────────────────────────────────┐")
    print("│ Fetching news (provider: NEWS_SENTIMENT)                    │")
    print("└─────────────────────────────────────────────────────────────┘")
    raw_news = fetch_news(symbol)
    _print_kv("fetched_articles:", 0 if raw_news is None else len(raw_news))
    if raw_news is None or raw_news.empty:
        print("  No news data available.")
        return AgentResponse(
            agent_name="News Analysis Agent",
            analysis="No news returned from provider.",
            score=0.0,
            confidence=0.4,
            key_factors=["no_news_data"],
            timestamp=datetime.now().isoformat()
        )

    # 2) Preprocess
    print("\n┌─ STEP 2/5: PREPROCESS ──────────────────────────────────────┐")
    clean = preprocess_news(raw_news)
    _print_kv("after_preprocess:", len(clean))

    # 3) Tag & numbers
    print("\n┌─ STEP 3/5: CLASSIFY ────────────────────────────────────────┐")
    tagged = add_tags_and_numbers(clean)
    _print_kv("after_tagging:", len(tagged))

    # 4) Recent top-K (+ optional tag filter)
    print("\n┌─ STEP 4/5: EXTRACT ─────────────────────────────────────────┐")
```

```
    topk = recent_topk(
        tagged,
        topk=SETTINGS.topk_news,
        days=SETTINGS.news_window_days,
        required_tags=required_tags
    )
    _print_kv("top_articles:", len(topk))

    # 5) Summarize
    print("\n┌─ STEP 5/5: SUMMARIZE ──────────────────────────────────────────┐")
    payload = {
        "ticker": symbol,
        "news": (
            topk.rename(columns={"summary": "description"})
                .loc[:, ["title", "description", "source", "url", "published_at"]]
                .to_dict("records")
            if not topk.empty else []
        ),
    }
    res = NewsAnalysisAgent().process(payload)
    _print_kv("sentiment_score:", f"{res.score:+.2f}")
    _print_kv("confidence:", f"{res.confidence:.0%}")

    print("\n" + "=" * 80)
    print("PROMPT CHAINING COMPLETE")
    print("Pattern: Raw → Clean → Tagged → Top-K → Analysis")
    print("=" * 80 + "\n")
    return res

# ──────────────────────────────────────────────────────────────────────────
# WORKFLOW 2: Parallel execution (uses CSV/synthetic prices via offline loader)
# ──────────────────────────────────────────────────────────────────────────

def run_parallel_workflow(symbol: str, start: str, end: str) -> List[AgentResponse]:
    from concurrent.futures import ThreadPoolExecutor

    print("\n" + "=" * 80)
    print("WORKFLOW PATTERN 2: PARALLEL EXECUTION")
    print("=" * 80)
    print(f"Analyzing: {symbol} | Period: {start} → {end}")
    print("=" * 80)

    print("\n[Preparation] Fetching base data (offline prices)…")
    prices = fetch_prices_offline(symbol, start, end)  # <- CSV-first
    news = pd.DataFrame()
    if not getattr(SETTINGS, "skip_news", False):
        news = fetch_news(symbol)
    earnings = fetch_earnings(symbol)
    risk_ingested = fetch_risk_metrics(symbol, start, end)

    # News agent input
    news_input = {
        "ticker": symbol,
        "news": (
            news.head(5).rename(columns={"summary": "description"})
                .loc[:, ["title", "description", "source", "url", "published_at"]]
                .to_dict("records")
            if not news.empty else []
        ),
    }
```

```python
# Technicals (price-only snapshot)
tech_input = {
    "ticker": symbol,
    "technicals": {
        "current_price": float(prices["close"].iloc[-1]) if not prices.empty else None,
        "volume": int(prices["volume"].iloc[-1]) if (not prices.empty and not pd.isna(prices["volume"].iloc[-1])) else None,
        "avg_volume": int(prices["volume"].tail(20).mean()) if (not prices.empty and not prices["volume"].tail(20).isna().all()) else None,
        "rsi": None, "ma_50": None, "ma_200": None,
        "macd": None, "support": None, "resistance": None,
    },
}

# Risk: realized 30d vol from prices + ingested metrics
vol_30d = float(prices["close"].pct_change().tail(30).std() * 100) if not prices.empty else None
risk_input = {
    "ticker": symbol,
    "risk_metrics": {
        "beta":              risk_ingested.get("beta"),
        "volatility":        vol_30d,
        "var_5":             risk_ingested.get("var_5"),
        "sharpe_ratio":      risk_ingested.get("sharpe_ratio"),
        "max_drawdown":      risk_ingested.get("max_drawdown"),
        "sector_correlation": None,
        "pe_ratio":           None,
        "avg_daily_return":   risk_ingested.get("avg_daily_return"),
        "volatility_full":    risk_ingested.get("volatility"),
    },
}

earn_input = {
    "ticker": symbol,
    "earnings": (
        earnings.sort_values("date", ascending=False).head(8).to_dict("records")
        if earnings is not None and not earnings.empty else []
    ),
}

print("\n[Parallel] Running News + Technical + Risk + Earnings (4 agents)…")
t0 = time.time()
futures = {}
with ThreadPoolExecutor(max_workers=4) as pool:
    if not getattr(SETTINGS, "skip_news", False):
        futures["news"] = pool.submit(NewsAnalysisAgent().process, news_input)
    futures["technical"] = pool.submit(MarketSignalsAgent().process, tech_input)
    futures["risk"] = pool.submit(RiskAssessmentAgent().process, risk_input)
    futures["earnings"] = pool.submit(EarningsAnalysisAgent().process, earn_input)

    results: Dict[str, AgentResponse] = {}
    for name, fut in futures.items():
        results[name] = fut.result()
        print(f"  {name.capitalize():<10} Score={results[name].score:+.2f}  Conf={results[name].confidence:.0%}")

elapsed = time.time() - t0
print("\n" + "=" * 80)
print(f"PARALLEL EXECUTION COMPLETE ({elapsed:.2f}s)")
print("Pattern: Agents run concurrently to shorten wall time.")
print("=" * 80 + "\n")

return list(results.values())
```

```python
# ──────────────────────────────────────────────────────────────────────────────
# WORKFLOW 3: Evaluator-Optimizer (wraps orchestrator; patched to CSV prices)
# ──────────────────────────────────────────────────────────────────────────────

def run_evaluator_optimizer_workflow(
    symbol: str,
    start: str,
    end: str,
    required_tags: list[str] | None = None,
    force_optimizer: bool = False
) -> OrchestratorResult:
    print("\n" + "=" * 80)
    print("WORKFLOW PATTERN 3: EVALUATOR-OPTIMIZER")
    print("=" * 80)
    print(f"Analyzing: {symbol} | Period: {start} → {end}")
    print("=" * 80)

    print("\n[Phase 1] GENERATE: Running pipeline (prices are CSV/synthetic via monkey patch)…")
    result = run_pipeline(symbol, start, end, required_tags)

    initial = next((a for a in result.agent_outputs if "Initial Synthesis" in a.agent_name), None)
    if initial:
        _print_kv("initial_score:", f"{initial.score:+.2f}")
        _print_kv("initial_conf:", f"{initial.confidence:.0%}")

    print("\n[Phase 2] EVALUATE: Critique")
    _print_kv("quality_score:", f"{result.critique.score:.2f}")
    _print_kv("adj_confidence:", f"{result.critique.confidence:.0%}")
    _print_kv("issues_found:", len(result.critique.key_factors))

    optimizer_ran = initial and (_normalize_text(initial.analysis) != _normalize_text(result.final.analysis))
    if force_optimizer:
        optimizer_ran = True

    if optimizer_ran:
        print("\n[Phase 3] OPTIMIZE: Re-synthesized with critique feedback (v2)")
        _print_kv("final_score:", f"{result.final.score:+.2f}")
        _print_kv("final_conf:", f"{result.final.confidence:.0%}")
        if initial:
            delta = result.final.confidence - initial.confidence
            _print_kv("conf_change:", f"{delta:+.0%}")
    else:
        print("\n[Phase 3] OPTIMIZE: Not needed (quality acceptable)")

    print("\n" + "=" * 80)
    print("EVALUATOR-OPTIMIZER COMPLETE")
    print("=" * 80 + "\n")

    return result

# ──────────────────────────────────────────────────────────────────────────────
# DEMO
# ──────────────────────────────────────────────────────────────────────────────

def demo_all_workflows(symbol: str = "AAPL"):
    start = (datetime.now() - timedelta(days=30)).strftime("%Y-%m-%d")
    end = datetime.now().strftime("%Y-%m-%d")

    print("\n" + "*" * 40)
```

```
    print("  DEMONSTRATING 3 AGENTIC WORKFLOW PATTERNS (offline prices)")
    print("*" * 40)
    print(f"\nTicker: {symbol}")
    print(f"Date Range: {start} → {end}\n")

    r1 = run_prompt_chaining_workflow(symbol, start, end)
    r2 = run_parallel_workflow(symbol, start, end)
    r3 = run_evaluator_optimizer_workflow(symbol, start, end, force_optimizer=False)

    print("\n" + "#" * 40)
    print("  ALL 3 WORKFLOW PATTERNS DEMONSTRATED")
    print("#" * 40 + "\n")

    return {"prompt_chaining": r1, "parallel": r2, "evaluator_optimizer": r3}

if __name__ == "__main__":
    demo_all_workflows("AAPL")
```

```
[offline] Patched src.data_io.prices.fetch_prices → CSV/synthetic loader.

**************************************
  DEMONSTRATING 3 AGENTIC WORKFLOW PATTERNS (offline prices)
**************************************

Ticker: AAPL
Date Range: 2025-09-20 → 2025-10-20


================================================================================
WORKFLOW PATTERN 1: PROMPT CHAINING
================================================================================
Analyzing: AAPL | Period: 2025-09-20 → 2025-10-20
================================================================================

┌─ STEP 1/5: INGEST ──────────────────────────────────────┐
│ Fetching news (provider: NEWS_SENTIMENT)                │
└─────────────────────────────────────────────────────────┘
  fetched_articles:  19

┌─ STEP 2/5: PREPROCESS ──────────────────────────────────┐
  after_preprocess:  19

┌─ STEP 3/5: CLASSIFY ────────────────────────────────────┐
  after_tagging:     19

┌─ STEP 4/5: EXTRACT ─────────────────────────────────────┐
  top_articles:      5

┌─ STEP 5/5: SUMMARIZE ───────────────────────────────────┐
  sentiment_score:   +0.85
  confidence:        90%

================================================================================
PROMPT CHAINING COMPLETE
Pattern: Raw → Clean → Tagged → Top-K → Analysis
================================================================================


================================================================================
WORKFLOW PATTERN 2: PARALLEL EXECUTION
================================================================================
Analyzing: AAPL | Period: 2025-09-20 → 2025-10-20
================================================================================

[Preparation] Fetching base data (offline prices)…

[Parallel] Running News + Technical + Risk + Earnings (4 agents)…
  News       Score=+0.85  Conf=90%
  Technical  Score=+0.00  Conf=20%
  Risk       Score=+0.45  Conf=55%
  Earnings   Score=+0.80  Conf=90%

================================================================================
PARALLEL EXECUTION COMPLETE (2.77s)
Pattern: Agents run concurrently to shorten wall time.
================================================================================
```

```
================================================================================
WORKFLOW PATTERN 3: EVALUATOR-OPTIMIZER
================================================================================
Analyzing: AAPL | Period: 2025-09-20 → 2025-10-20
================================================================================

[Phase 1] GENERATE: Running pipeline (prices are CSV/synthetic via monkey patch)…
  initial_score:     +0.60
  initial_conf:      72%

[Phase 2] EVALUATE: Critique
  quality_score:     0.68
  adj_confidence:    65%
  issues_found:      3

[Phase 3] OPTIMIZE: Re-synthesized with critique feedback (v2)
  final_score:       +0.60
  final_conf:        72%
  conf_change:       +0%

================================================================================
EVALUATOR-OPTIMIZER COMPLETE
================================================================================


########################################
  ALL 3 WORKFLOW PATTERNS DEMONSTRATED
########################################
```

Mock Gradio App — Evaluation and Iteration, Demonstration Interface for the Agentic Pipeline

- **This notebook section provides a mock Gradio user interface designed purely for demonstration and research purposes.**
- **It uses saved data from the local directory:**

  data/runs/ui_runs/

- **to showcase what a full agentic analysis would look like after execution.**

If we want a fully interactive and functional agentic app — where agents fetch live data, process it, and collaborate in real-time we must use the layered codebase

In [49]:
```python
# ui/gradio_app.py — Gradio front-end for the agentic pipeline
# We run one ticker analysis, render agent outputs + evidence, and
# add simple persistence (Save / Load last run) for quick comparisons.

from __future__ import annotations

# --- stdlib / paths ---
import os, sys, traceback, socket, json
from datetime import date, timedelta, datetime
from pathlib import Path

# --- 3rd-party ---
import gradio as gr
```

```python
import pandas as pd

# --- import path shim (repo root) ---
# We make sure ../ is importable so src/ modules resolve when we run from ui/.
# sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), "..")))

# --- project imports ---
from src.system.orchestrator import run_pipeline
from src.config.settings import SETTINGS  # used for runs_dir

# ──────────────────────────────────────────────────────────────────────────────
# Persistence (Save / Load) — we keep lightweight artifacts under data/runs/ui_runs
# ──────────────────────────────────────────────────────────────────────────────

# We store UI runs under SETTINGS.runs_dir/ui_runs so the app and orchestrator share a root.
RUNS_UI_DIR = SETTINGS.runs_dir / "ui_runs"
RUNS_UI_DIR.mkdir(parents=True, exist_ok=True)

def _df_to_csv(path: Path, df: pd.DataFrame) -> None:
    """We write a DataFrame to CSV if it exists and is non-empty."""
    try:
        if df is None or (hasattr(df, "empty") and df.empty):
            return
        df.to_csv(path, index=False)
    except Exception:
        # We keep the UI robust — no hard crashes on save failure.
        pass

def _df_from_csv(path: Path) -> pd.DataFrame:
    """We read a CSV into a DataFrame; return empty frame on any error."""
    try:
        return pd.read_csv(path) if path.exists() else pd.DataFrame()
    except Exception:
        return pd.DataFrame()

def save_current_run(symbol, days_back, tags,
                     plan_txt, agents_txt, crit_txt, final_txt,
                     news_df, prices_df, earnings_df, risk_df) -> str:
    """We persist the current UI state into a timestamped folder."""
    # Guard: only save if something meaningful is present.
    if not any([plan_txt, agents_txt, crit_txt, final_txt]):
        return "Nothing to save yet. Run the analysis first."

    ts = datetime.now().strftime("%Y%m%d_%H%M%S")
    run_dir = RUNS_UI_DIR / f"{(symbol or 'UNKNOWN').strip()}_{ts}"
    run_dir.mkdir(parents=True, exist_ok=True)

    # We keep a small JSON meta for quick reloads.
    meta = {
        "symbol": symbol,
        "days_back": int(days_back) if str(days_back).strip() else None,
        "tags": tags,
        "timestamp": ts,
        "plan": plan_txt,
        "agents": agents_txt,
        "critique": crit_txt,
        "final": final_txt,
    }
    (run_dir / "meta.json").write_text(
        json.dumps(meta, ensure_ascii=False, indent=2), encoding="utf-8"
```

```
    )

    # Evidence tables (optional; saved if non-empty)
    _df_to_csv(run_dir / "news.csv", news_df)
    _df_to_csv(run_dir / "prices.csv", prices_df)
    _df_to_csv(run_dir / "earnings.csv", earnings_df)
    _df_to_csv(run_dir / "risk.csv", risk_df)

    return f"Saved to: {run_dir}"

def load_last_run():
    """We load the newest saved run and return both text panels and tables, plus input defaults."""
    runs = [p for p in RUNS_UI_DIR.iterdir() if p.is_dir()]
    if not runs:
        # When nothing saved yet, return empty panels and sane input defaults.
        return (
            "", "", "", "",
            pd.DataFrame(), pd.DataFrame(), pd.DataFrame(), pd.DataFrame(),
            "", 30, ""   # days_back must be int for the Slider
        )

    # Newest first by mtime
    runs.sort(key=lambda p: p.stat().st_mtime, reverse=True)
    run_dir = runs[0]

    meta_path = run_dir / "meta.json"
    meta = json.loads(meta_path.read_text(encoding="utf-8")) if meta_path.exists() else {}

    plan_txt   = meta.get("plan", "")
    agents_txt = meta.get("agents", "")
    crit_txt   = meta.get("critique", "")
    final_txt  = meta.get("final", "")
    symbol     = str(meta.get("symbol", "") or "")
    tags       = str(meta.get("tags", "") or "")

    # We coerce days_back to int for the Slider widget.
    raw_days = meta.get("days_back", 30)
    try:
        days_back = int(raw_days)
    except Exception:
        days_back = 30

    # Evidence tables
    news_df     = _df_from_csv(run_dir / "news.csv")
    prices_df   = _df_from_csv(run_dir / "prices.csv")
    earnings_df = _df_from_csv(run_dir / "earnings.csv")
    risk_df     = _df_from_csv(run_dir / "risk.csv")

    return (plan_txt, agents_txt, crit_txt, final_txt,
            news_df, prices_df, earnings_df, risk_df,
            symbol, days_back, tags)

def _apply_loaded(plan_txt, agents_txt, crit_txt, final_txt,
                  news_df, prices_df, earnings_df, risk_df,
                  sym, days, tagstr):
    """We push loaded values into UI components; ensure Slider gets a number."""
    try:
        days_val = int(days)
    except Exception:
        days_val = 30
```

```
    status = f"Loaded last run from: {RUNS_UI_DIR}"
    return (
        plan_txt, agents_txt, crit_txt, final_txt,
        news_df, prices_df, earnings_df, risk_df,
        gr.update(value=str(sym or "")),
        gr.update(value=days_val),            # Slider expects a numeric value
        gr.update(value=str(tagstr or "")),
        status
    )

# ─────────────────────────────────────────────────────────────────────────────
# Small UI helpers
# ─────────────────────────────────────────────────────────────────────────────

def _truncate(s: str, max_len: int = 8000) -> str:
    """We shorten long text for responsive UI and websocket limits."""
    if not isinstance(s, str):
        s = str(s)
    return (s[: max_len - 20] + " … (truncated)") if len(s) > max_len else s

def _as_text(x):
    """We coerce anything to text; pretty-print lists/dicts."""
    if x is None:
        return ""
    if isinstance(x, str):
        return x
    if isinstance(x, (dict, list)):
        return json.dumps(x, ensure_ascii=False, indent=2, sort_keys=True)
    return str(x)

def _clean(s: str) -> str:
    """We normalize whitespace and strip accidental code fences."""
    s = _as_text(s).strip()
    if s.startswith("```"):
        s = s.strip("`").strip()
    return s

def _synth_to_prose(obj):
    """We render a compact, readable summary when synthesis is a dict."""
    if not isinstance(obj, dict):
        return _clean(_as_text(obj))

    parts = []

    # --- Technicals block ---
    ms = obj.get("market_signals") or {}
    if ms:
        ms_bits = []
        cp = ms.get("current_price")
        if isinstance(cp, (int, float)):
            ms_bits.append(f"price ${cp:,.2f}")
        ma = ms.get("moving_averages") or {}
        ma50 = ma.get("50_day")
        ma200 = ma.get("200_day")
        if (ma50 is not None) or (ma200 is not None):
            ms_bits.append(f"vs 50D {ma50}, 200D {ma200}")
        rsi = ms.get("RSI")
        if rsi is not None:
            ms_bits.append(f"RSI {rsi}")
        trend = ms.get("trend")
```

```
        if trend:
            ms_bits.append(trend)
        vol = ms.get("volume") or {}
        vcur, vavg = vol.get("current"), vol.get("average")
        if vcur is not None and vavg is not None:
            ms_bits.append(f"volume {vcur:,} vs avg {vavg:,}")
        if ms_bits:
            parts.append("Technicals: " + ", ".join(str(x) for x in ms_bits if x))

    # --- News block ---
    news = obj.get("news") or {}
    if news:
        news_bits = []
        for k in ("sentiment", "growth potential", "competitive landscape"):
            if k in news:
                news_bits.append(f"{k}: {news[k]}")
        for k, v in news.items():
            if k not in ("sentiment", "growth potential", "competitive landscape"):
                news_bits.append(f"{k}: {v}")
        parts.append("News: " + "; ".join(news_bits))

    # --- Risk block ---
    risk = obj.get("risk_assessment") or {}
    if risk:
        risk_bits = []
        for k in ("volatility", "data_gaps", "idiosyncratic_risks"):
            if k in risk:
                risk_bits.append(f"{k}: {risk[k]}")
        for k, v in risk.items():
            if k not in ("volatility", "data_gaps", "idiosyncratic_risks"):
                risk_bits.append(f"{k}: {v}")
        parts.append("Risk: " + "; ".join(risk_bits))

    return "\n".join(parts).strip()

def _to_df(x):
    """We coerce any input to a DataFrame; return empty on failure."""
    if isinstance(x, pd.DataFrame):
        return x
    if x is None:
        return pd.DataFrame()
    try:
        return pd.DataFrame(x)
    except Exception:
        return pd.DataFrame()

# ─────────────────────────────────────────────────────────────────────────────
# Core handler — run the orchestrator and format panels for the UI
# ─────────────────────────────────────────────────────────────────────────────

def run(symbol, days_back, required_tags_csv):
    """We execute the pipeline and build text panels + evidence tables."""
    try:
        # Inputs → time window and optional tag filter
        start = (date.today() - timedelta(days=int(days_back))).isoformat()
        end = date.today().isoformat()
        tags = [t.strip() for t in required_tags_csv.split(",")] if required_tags_csv else None

        # Orchestrator: returns final = synth_v2 if optimizer ran, else synth_v1
        res = run_pipeline(symbol.strip().upper(), start, end, required_tags=tags)
```

```
        # Did the optimizer re-synthesize? We compare the initial synthesis vs final text.
        optimizer_ran = False
        init = next(
            (a for a in res.agent_outputs
             if a.agent_name in {"Initial Synthesis", "Research Synthesis Agent", "SynthesisAgent"}),
            None
        )
        if init is not None:
            init_txt = _clean(_as_text(init.analysis))
            final_txt_norm = _clean(_as_text(res.final.analysis))
            optimizer_ran = (init_txt != final_txt_norm)

        # Step plan
        plan = "\n".join([f"• {step}" for step in res.plan])

        # Agents panel — we keep it compact and truncate long text
        agents_txt = "\n\n".join([
            (
                f"[{a.agent_name}] score={a.score:.2f} conf={a.confidence:.2f}\n"
                f"{_synth_to_prose(a.analysis) if ('synthesis' in a.agent_name.lower()) else _clean(_as_text(a.analysis))}"
            )
            for a in res.agent_outputs
        ])
        agents_txt = _truncate(agents_txt, 15000)

        # Evidence tables for UI
        news_rows      = _to_df(res.evidence.get("top_news", []))
        prices_rows    = _to_df(res.evidence.get("prices_tail", []))
        earnings_rows  = _to_df(res.evidence.get("earnings_head", []))
        risk_rows      = _to_df(res.evidence.get("risk_metrics", []))  # likely single-row DF

        if news_rows.empty:
            agents_txt += "\n\n[Note] No news items matched filters or provider limits today."

        # Critique panel
        crit_txt = (
            f"[Critique]\n"
            f"score={res.critique.score:.2f} adj_conf={res.critique.confidence:.2f}\n"
            f"{_clean(_as_text(res.critique.analysis))}"
        )
        crit_txt = _truncate(crit_txt, 6000)

        # Final panel (labels v2 when optimizer ran)
        headline = "FINAL (v2 after Critique)" if optimizer_ran else "FINAL (v1)"
        opt_line = "[Optimizer ran: YES]" if optimizer_ran else "[Optimizer ran: NO]"
        final_txt = (
            f"{headline}\n{opt_line}\n"
            f"score={res.final.score:.2f} conf={res.final.confidence:.2f}\n"
            f"{_synth_to_prose(res.final.analysis)}\n\nKey: {', '.join(res.final.key_factors)}"
        )
        final_txt = _truncate(final_txt, 8000)

        # Return order MUST match the outputs wiring below.
        return (
            plan,
            agents_txt,
            crit_txt,
            final_txt,
            news_rows,
```

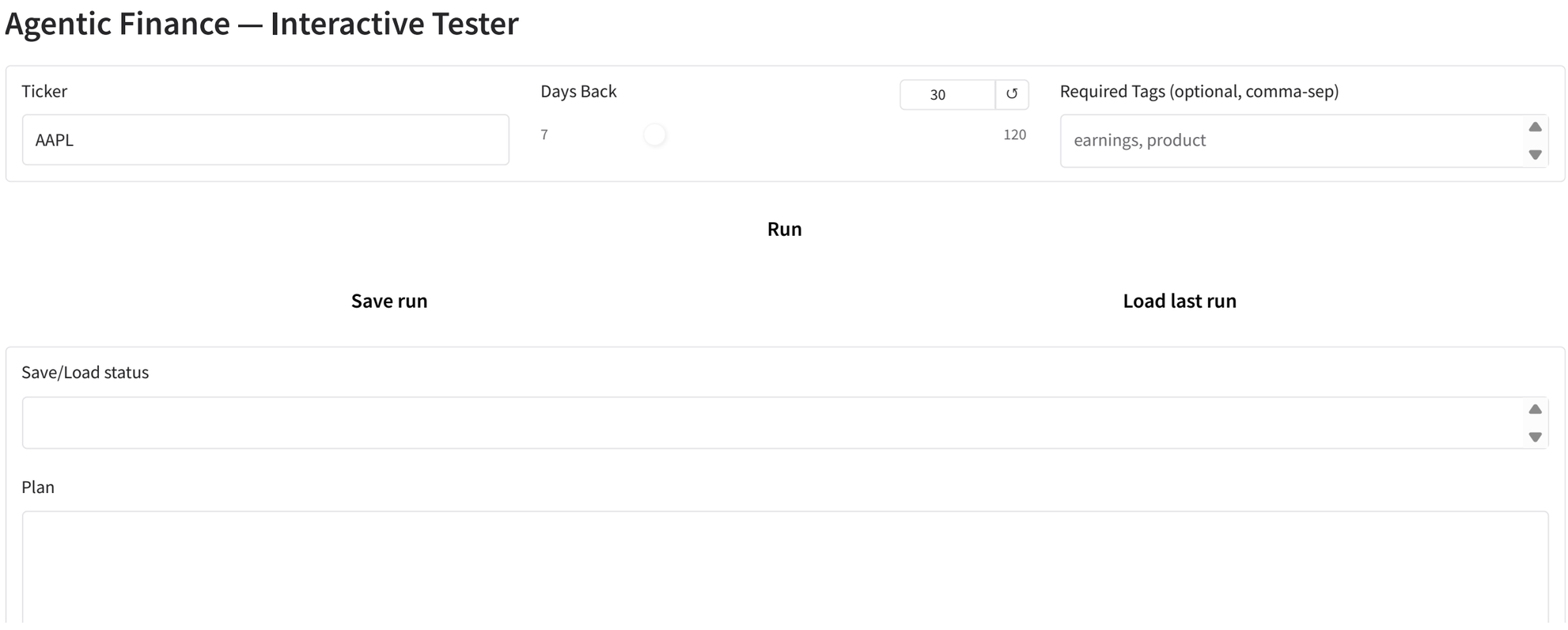
```
            prices_rows,
            earnings_rows,
            risk_rows
        )

    except Exception:
        # We catch all exceptions so the UI stays alive and shows the traceback.
        tb = traceback.format_exc()
        err = f"[FATAL] An exception occurred in run():\n{tb}"
        blank_df = pd.DataFrame()
        return (
            "run() error — see Critique tab",
            _truncate(err, 15000),
            _truncate(err, 6000),
            _truncate(err, 8000),
            blank_df, blank_df, blank_df, blank_df
        )

# ——————————————————————————————————————————————————————————————
# Gradio layout
# ——————————————————————————————————————————————————————————————

with gr.Blocks(title="Agentic Finance") as demo:
    gr.Markdown("# Agentic Finance — Interactive Tester")

    # Inputs row
    with gr.Row():
        symbol = gr.Textbox(label="Ticker", value="AAPL")                # we set the ticker
        days_back = gr.Slider(7, 120, value=30, step=1, label="Days Back")# we choose lookback window
        tags = gr.Textbox(label="Required Tags (optional, comma-sep)", placeholder="earnings, product")
    run_btn = gr.Button("Run")

    # Persistence controls
    with gr.Row():
        save_btn = gr.Button("Save run")
        load_btn = gr.Button("Load last run")
    save_status = gr.Textbox(label="Save/Load status", interactive=False)

    # Text panels
    plan   = gr.Textbox(label="Plan", lines=6)
    agents = gr.Textbox(label="Agent Outputs", lines=14)
    crit   = gr.Textbox(label="Critique", lines=8)
    final  = gr.Textbox(label="Final Recommendation", lines=10)

    # Evidence tables
    news_tbl     = gr.Dataframe(
        headers=["published_at","source","title","summary","url","overall_sentiment","tags","numbers"],
        label="Top News (evidence)",
        wrap=True
    )
    prices_tbl   = gr.Dataframe(label="Recent Prices (evidence)")
    earnings_tbl = gr.Dataframe(label="Earnings (evidence)")
    risk_tbl     = gr.Dataframe(label="Risk Metrics (evidence)")

    # Run click → pipeline
    run_btn.click(
        run,
        inputs=[symbol, days_back, tags],
        outputs=[plan, agents, crit, final, news_tbl, prices_tbl, earnings_tbl, risk_tbl]
    )
```

```python
# Save the current run (panels + tables + inputs)
save_btn.click(
    save_current_run,
    inputs=[symbol, days_back, tags, plan, agents, crit, final, news_tbl, prices_tbl, earnings_tbl, risk_tbl],
    outputs=[save_status],
)

# Load last run into both outputs and inputs
load_btn.click(
    load_last_run,
    inputs=[],
    outputs=[plan, agents, crit, final, news_tbl, prices_tbl, earnings_tbl, risk_tbl, symbol, days_back, tags],
).then(
    _apply_loaded,
    inputs=[plan, agents, crit, final, news_tbl, prices_tbl, earnings_tbl, risk_tbl, symbol, days_back, tags],
    outputs=[plan, agents, crit, final, news_tbl, prices_tbl, earnings_tbl, risk_tbl, symbol, days_back, tags, save_status],
)

# Auto-load last run at app start (nice for demos)
demo.load(
    load_last_run,
    inputs=[],
    outputs=[plan, agents, crit, final, news_tbl, prices_tbl, earnings_tbl, risk_tbl, symbol, days_back, tags],
).then(
    _apply_loaded,
    inputs=[plan, agents, crit, final, news_tbl, prices_tbl, earnings_tbl, risk_tbl, symbol, days_back, tags],
    outputs=[plan, agents, crit, final, news_tbl, prices_tbl, earnings_tbl, risk_tbl, symbol, days_back, tags, save_status],
)
```

```python
# ──────────────────────────────────────────────────────────────────────────────
# Launch
# ──────────────────────────────────────────────────────────────────────────────

def _get_free_port(start=7860, end=7890):
    """We scan for a free localhost port so the app starts reliably."""
    for p in range(start, end + 1):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            try:
                s.bind(("127.0.0.1", p))
                return p
            except OSError:
                continue
    return None  # let Gradio auto-pick if nothing is free

if __name__ == "__main__":
    # We queue for concurrency; handle older Gradio signatures gracefully.
    try:
        demo.queue()
    except TypeError:
        try:
            demo.queue(max_size=16)
        except TypeError:
            pass

    port = _get_free_port()  # None → Gradio will auto-pick

    try:
        demo.launch(
```

```
            share=False,
            server_name="127.0.0.1",
            server_port=port,      # may be None
            show_error=True
        )
    except OSError:
        # Fallback: force auto-pick if the chosen port gets taken meanwhile
        demo.launch(
            share=False,
            server_name="127.0.0.1",
            server_port=None,
            show_error=True
        )
```

```
* Running on local URL:  http://127.0.0.1:7865
* To create a public link, set `share=True` in `launch()`.
```



Agentic Finance — Interactive Demonstration Summary

- **This interactive experiment was executed using the Agentic Finance tester with the ticker AAPL, a 30-day look-back period**
  - **The system automatically triggered the full agentic pipeline: data ingestion (prices, news, earnings, risk), preprocessing, classification, retrieval, routing, multi-agent execution, synthesis, critique, and final recommendation.**
- **Agent Interactions**
  - **News Analysis Agent:** Identified positive sentiment driven by strategic partnerships and exclusive content rights.

    - **Market Signals Agent:** Reported a bullish technical pattern, with price action above key moving averages and an RSI indicating strong upward momentum.
    - **Risk Assessment Agent:** Detected moderate risk: a negative Sharpe ratio and small negative average daily return imply weak risk-adjusted performance despite low volatility.
    - **Synthesis Agent (Initial Synthesis):** Merged the optimistic signals from the news and technical agents with the moderate risk findings. The model recommended "buy with caution".
    - **Critique Agent:** Flagged two key issues: insufficient coverage of the competitive landscape and incomplete risk analysis. Suggested adding macroeconomic context and assessing partnership sustainability.
    - **Optimizer (Final Recommendation Synth v2):** After incorporating critique feedback, the final analysis maintained cautious optimism. The confidence level remained steady at 0.72, indicating consistent reliability across synthesis rounds.
- **Evidence Integration**
    - **Each agent supplied structured "evidence tables"** (news, prices, earnings, risk) so that the synthesis could align numeric data with narrative analysis.
- **Current Limitations and Future Work**
    - **Score Calibration: Some final scores** Agent confidence and scoring logic require fine-tuning so aggregate confidence reflects the combined evidence strength.
    - **Earnings Fetch:** The absence of earnings information is one of the limitations known. A secondary data provider must be included in future versions.

Conclusion:

```
overall, this demonstration proves that the multi-agent system coordination going as planned. Collectively, these interactions build a balanced financial evaluation
pipeline that is realistic. The organizer was able to achieve multi-agent reasoning, data alignment, and critique-based optimization, exemplifying how Agentic Finance
is capable of creating adaptive and explainable market analysis to be used in future studies and incorporation into an enterprise.
```


Acknowledgment: AI Assistance

Parts of this project (e.g., debugging helper functions, improving documentation clarity, and suggesting modularization patterns) were supported using OpenAI ChatGPT (GPT-5). The team used AI-generated suggestions as guidance only — all code was reviewed, edited, and verified manually to ensure understanding and compliance with course requirements.

References

Yahoo Inc (2025). *yfinance* [Computer software]. https://pypi.org/project/yfinance/
Alpha Vantage Inc. (2025). *Alpha Vantage API (free tier)* [Web API]. https://www.alphavantage.co/
Hugging Face. (2025). *Introduction (Unit 0 - 2): Agents course* [Online course module]. https://huggingface.co/learn/agents-course/en/unit0/introduction
University of San Diego, MSAAI. (2025). *Presentation 7.1: Agentic AI* [Lecture video]. University of San Diego.
University of San Diego, MSAAI. (2025). *Module 7 Lab* [Laboratory manual]. University of San Diego.
OpenAI. (2025). *ChatGPT* [Large language model; used for Markdown text cell formatting]. https://chat.openai.com/