ChatGPT

Technical Implementation Plan: Agentic Hedge Fund Simulator

Development Environment & Tools

- **Python Version:** Use the latest stable Python 3.x (e.g. Python 3.11) for compatibility with modern libraries and the OpenAI Agents SDK. Ensure consistent environments across team members.
- **Dependencies:** Include the OpenAI API/Agents SDK (for LLM agent orchestration), data access libraries (e.g. `requests`, `pandas`, `yfinance` for Yahoo Finance data), scheduling tools (e.g. `APScheduler`), and any analysis helpers (such as `numpy` for calculations). List these in a `pyproject.toml` or `requirements.txt` for reproducibility.
- **Package Management:** Use **Poetry** for dependency management and virtual environment isolation. For example, the open-source *Hedge_Fund_Agents* project uses Poetry to manage dependencies and environment setup [1]. Poetry makes it easy to pin package versions and ship the project as a reproducible environment. Alternatively, a `venv` + pip requirements file can be used if preferred.
- **Local Development Tools:** Integrate linters and formatters to maintain code quality:
- Use **Black** for auto-formatting and **Flake8** or **pylint** for linting to enforce PEP8 standards.
- Set up **pre-commit hooks** to run formatting/lint checks on every commit (e.g. format with Black, sort imports with `isort`, run Flake8). This prevents style issues from entering the repo.
- Leverage type checking with **mypy** to catch type errors early.
- Recommended IDE: VS Code or PyCharm with Python extensions (enabling Pylance or Pyright for live type checking, and Black integration for on-save formatting).
- **Git & CI:** Use git for version control. Optionally configure a GitHub Actions or local CI pipeline to run tests on each commit (even if the system runs locally in production). For example, you can include a GitHub Actions workflow to run `pytest` and linting on push.

System Architecture

Figure: Conceptual architecture of the multi-agent trading system. A central Agent Manager orchestrates specialized agents (Fundamental, Technical, Sentiment analysts) and services (Risk Manager, Execution Engine), using a shared data pipeline and tools to interact with external data sources.

The system follows a **hub-and-spoke multi-agent architecture**, inspired by designs in recent OpenAI agent examples [2]. A central **Agent Manager** (Portfolio Manager agent) coordinates the workflow and delegates tasks to specialist agents, each focusing on a specific domain (e.g. fundamental analysis, technical analysis, sentiment analysis) [3]. We adopt the "agents-as-tools" collaboration pattern: the main agent invokes other agents as callable tools for sub-tasks, rather than handing off control [4] [5]. This keeps a single thread of control with the orchestrator and ensures transparency in reasoning while allowing parallel analysis of different data aspects [5] [3]. Below is an overview of the core modules/agents and their responsibilities in the architecture:

Module / Agent	Responsibilities
Agent Manager (Orchestrator)	Acts as the central **Portfolio Manager** agent [6]. Orchestrates the daily trading loop and coordinates all other agents. Delegates analysis tasks to specialist agents and aggregates their insights into a final decision [2]. Maintains conversation state if using the OpenAI SDK (ensuring context from various agents is combined). Initiates trades by calling the execution module after risk approval.
Data Pipeline	Handles data ingestion from external **free/open data sources** (market prices, fundamentals, news, macro data). Provides a unified interface (tools/functions) that agents call to fetch data. Implements caching and normalization of data. For example, it may wrap Yahoo Finance API calls for price data and SEC or FRED APIs for fundamentals and macro data [7] [8]. Ensures rate limiting and retries on failures.
Strategy Agents (Specialists)	Multiple **analysis agents** each focusing on a specific strategy or data domain [9] [10]. For instance: a **Fundamental Analyst** agent focuses on financial statements and valuations; a **Technical Analyst** agent looks at price patterns and technical indicators; a **Sentiment Analyst** agent monitors news or social media sentiment. Each agent may use tools (via the Agents SDK) to gather relevant data (e.g. a fundamental agent calls a function to retrieve financial statements, a technical agent calls for price history) and then interprets that data (potentially with LLM reasoning or hard-coded logic) to produce insights or trade signals. These agents act as autonomous experts whose outputs (e.g. "Buy AAPL, it's undervalued" or "Technical trend is bullish") are returned to the Agent Manager [3].
Risk & Compliance Engine	A dedicated module or agent to enforce risk management rules and compliance constraints. It evaluates proposed trades before execution – e.g. checking position sizing, exposure limits, stop-loss levels, or blacklisted assets. If a proposed trade would violate risk limits, this agent can veto or adjust it (for example, downsize the trade or reject it) [11]. It ensures no single trade or aggregate portfolio exceeds the $1,000 paper budget or other configured limits (e.g. max 20% of portfolio in one position). The risk agent runs automatic **sanity checks** to abort trades that are too risky or inconsistent with the strategy.
Paper Trading Engine	Simulates trade execution on a paper trading account. When the Agent Manager decides on a trade (after risk approval), it calls this module to "execute" the trade. The Paper Trading Engine will update a **portfolio state** (cash balance, holdings, P/L). It fills orders at simulated prices (e.g. using the latest market price from the data pipeline) and can apply simple fill logic (market orders assumed filled immediately). It also records each executed trade in a trade log. This module essentially acts as our **broker simulator**.
Dashboard & Reporting	(Optional) A monitoring interface to observe the fund's performance and the agents' decisions. This could be a simple web dashboard or CLI reports. It might display the current portfolio holdings, profit/loss, and a summary of recent agent decisions. It can also alert the user to important events (e.g. a trade aborted by risk limits or a large P/L change). Real-time charting of portfolio value or logging of agent outputs can be included for transparency.

Agent Communication & Tool Use: All agent interactions are orchestrated by the Agent Manager using the OpenAI Agents SDK. In practice, the specialist agents are exposed as tools or functions that the main agent can invoke [5]. For example, the Fundamental agent might be registered as a tool like `analyse_fundamentals(stock_ticker)` which internally triggers that agent's logic. The Agents SDK allows the main agent to call these sub-agents and receive their outputs as if calling an API. The sub-agents themselves can further use lower-level tools for data access or computations. For instance, an agent can call a custom Python function tool to fetch data from Yahoo Finance or FRED [7] [8], or use OpenAI's managed tools like **WebSearch** for news retrieval and **Code Interpreter** for quantitative analysis [12]. All agent-to-agent communication happens via these tool interfaces and shared context in the Agent Manager. This design keeps each agent's scope limited, but allows the orchestrator to integrate their results into a coherent trading strategy.

Parallelism: Because each specialist agent works independently on its subtask, the system can fetch data and analyze in parallel to save time (e.g. fundamental and technical analysis agents can run concurrently) [5]. The Agent Manager can launch these agent-tools asynchronously and then gather results, which is supported by the Agents SDK. This will be important for efficiency if many data sources are used, though initial implementation can run them sequentially for simplicity.

Data Pipeline

Data Ingestion Sources: Utilize free or open data APIs for all required market information: - **Market Price Data:** Use sources like **Yahoo Finance** (via the `yfinance` Python library or directly through an API) to get historical and current OHLCV prices for stocks. Yahoo provides free daily and intraday price quotes. Alternatively, an API like Alpha Vantage (free tier) can be used (with appropriate rate limiting). The OpenAI Agents SDK example uses a local MCP server to fetch Yahoo Finance data, demonstrating how an agent can retrieve price history on demand [7]. - **Fundamentals:** Leverage free financial statement data. For example, one can scrape Yahoo Finance key statistics or use an API like Alpha Vantage for company fundamental metrics (e.g. P/E ratio, earnings). Financial statement endpoints (income statements, balance sheets) are available from services like FinancialModelingPrep or Alphavantage (though free rate limits apply). The **SEC EDGAR** database is another open resource (for filings), but parsing filings is complex; instead, simplified free APIs or datasets are preferable. - **News & Sentiment:** Incorporate news headlines and social sentiment via open sources. This could involve using RSS feeds or a free news API (like NewsAPI with free tier) to pull headlines about portfolio stocks each day. The content can be run through a sentiment analysis model (e.g. a simple NLP model or even an OpenAI sentiment classification prompt) to quantify sentiment. If using the OpenAI Agents SDK tools, the **WebSearch** tool can directly fetch up-to-date news or web data [12], which a sentiment agent can summarize. For social sentiment, one might track Twitter or Reddit (though free access is limited; perhaps use public Reddit datasets or alternative free sentiment sources). - **Macro Data:** Use open economic data for macro context. For example, the **Federal Reserve Economic Data (FRED)** API provides free access to macro indicators (interest rates, CPI, etc.) [8]. We can use this for data like inflation rates or Treasury yields, which the strategy might consider. Other macro indices like S&P 500 index level (from Yahoo) or VIX (volatility index) can be fetched via the price data source as needed.

Data Flow & Storage: The data pipeline module will manage **fetching, caching, and normalizing** data: - Design this as an abstraction (e.g. a `DataPipeline` class) that has methods like `get_price_data(ticker, start_date, end_date)`, `get_fundamentals(ticker)`, `get_news(ticker)`, etc. These methods handle calling external APIs, parsing the response, and returning data in a standard format (e.g. a Pandas DataFrame or a Python dict). - Implement **caching** to

avoid redundant API calls. For example, maintain a local cache (in-memory or on-disk using JSON/CSV files or SQLite). When an agent requests data for `ticker X` on `date Y`, first check if we already have that data cached. If so, return it immediately; if not, fetch from the API, then store it in the cache for future use. This is critical with free APIs that have rate limits. - Use **Rate Limiting & Retries:** Respect the API rate limits by introducing delays or using token-bucket algorithms. If an API call fails or times out, implement a retry mechanism with exponential backoff. For instance, if a price API call fails, wait a few seconds and try again, up to a small number of retries. Log any failures for transparency. - **Fallback Data Sources:** Define multiple sources for critical data. If Yahoo Finance is not reachable, for example, try an alternate (Alpha Vantage or IEX Cloud's free tier) for price data. For news, maintain at least two feeds or use web search as backup. This ensures the system is robust against one source being down. - **Schema & Transformation:** Normalize data from various sources into common schemas for internal use. For example, for price data use a unified **OHLCV schema**:

Field	Type	Description
`date`	Date/Time	Timestamp of the price data (e.g. daily date or intraday datetime).
`open`	float	Opening price of the period.
`high`	float	Highest price of the period.
`low`	float	Lowest price of the period.
`close`	float	Closing price of the period.
`volume`	int	Volume traded in the period (if available).

All price sources will be converted to this structure so that strategy logic can consume it easily (e.g. as a Pandas DataFrame). Similarly, define a schema for fundamental data (e.g. a dict of key metrics like `{"PE": 15.2, "EPS": 3.1, ...}`) and for news/sentiment data (e.g. list of recent headlines with a sentiment score for each). Using Pydantic models to define these schemas can help with validation and clarity (for example, a `PriceBar` BaseModel with those fields). - **Data Tools for Agents:** In the context of the Agents SDK, wrap the data pipeline methods as **tools** that agents can call. For instance, create a tool function `fetch_prices(ticker, days)` that the Technical Analysis agent can invoke to get recent prices, or `lookup_fundamentals(ticker)` for the Fundamental agent. The OpenAI SDK allows registering custom Python functions as tools easily [13]. This way, agents don't call the APIs directly but go through our pipeline (ensuring caching and consistent formatting).

- **Example:** The Technical agent might call `fetch_prices('AAPL', days=30)` which our data pipeline implements by checking cache or calling Yahoo API (e.g. via `yfinance`), then returns a DataFrame of the last 30 days of OHLCV for AAPL. The Fundamental agent might call `lookup_fundamentals('AAPL')` which returns a dict of metrics or recent earnings data from our cache/requests. By designing these as tools, we abstract away the HTTP calls from the agent's logic.

Task Scheduling

Implement a **daily trading loop** that runs each trading day and coordinates the agents through the stages of ingest, analysis, decision, and execution:

1. **Morning Data Ingestion:** Schedule the system to start early in the day (e.g. 8:30 AM local time, before market open) to fetch the latest data. The Data Pipeline gathers any required **overnight updates** – for example, yesterday's closing prices, relevant news published since the last session, and any updated fundamentals or macro data. This ensures the agents work with fresh inputs each day.
2. **Analysis Phase:** Shortly after data ingestion, the Agent Manager triggers the specialist agents to perform analysis. The **Portfolio Manager agent** (orchestrator) prompts each Strategy agent with the relevant context. For example, it might prompt: *"Analyze stock XYZ given the latest data and your perspective."* Each agent (Fundamental, Technical, Sentiment) will run (potentially in parallel) and produce its analysis or recommendation [3] . The Agent Manager collects these results. This phase could be scheduled around market open (e.g. 9:30 AM) or earlier if preparing to trade at the open.
3. **Decision Making:** The Agent Manager synthesizes the insights. This might involve another LLM call where the Portfolio Manager agent weighs the different analyses and forms a coherent trading plan (e.g. decide to buy, sell, or hold certain positions) [2] . Alternatively, a simple rules engine can combine signals (e.g. majority vote of agents or certain priority rules). The output of this stage is a set of **proposed trades** (e.g. "Buy 5 shares of AAPL" or "Sell 10 shares of GOOGL").
4. **Risk & Compliance Check:** Before executing, the proposed trades are sent to the Risk Management engine. This can be an automated check (pure code) that verifies each trade against constraints. For instance, ensure no trade exceeds a certain dollar amount or fraction of the portfolio (respecting *position size limits*, a key risk practice [11]), ensure the portfolio won't be over-leveraged, and that trades don't violate any compliance rules. If a trade fails these checks, the risk engine can remove or adjust it (e.g. reduce quantity to fit limits, or flag it to abort). This step acts as a gate that must be passed each day.
5. **Execution Phase:** If trades are approved, schedule the **Paper Trading Engine** to execute them. This could be immediate after decision (e.g. right at market open if aiming to simulate market orders at open price). The Paper Trading module records the trade in the portfolio ledger, updates cash and holdings, and logs the execution price (likely using the current market price from the data pipeline). If the strategy waits for a specific time or price, this module could be invoked later in the day, but in a simple approach, executing at a consistent time (like 9:35 AM at near-open prices) is fine.
6. **Post-Trade Updates:** After execution, log the outcome (executed quantities, prices, new portfolio positions). The Data Pipeline may store end-of-day prices for P&L calculations. Optionally, an end-of-day routine (e.g. 4 PM) can run to finalize the day's P&L, perhaps record metrics (like daily return) and refresh the cache for the next day. The Dashboard (if any) can be updated with the day's results at this time.

Automation: To run the above steps daily without manual intervention: - Use a scheduler. Two options are: - **Cron Job:** Set up a system cron to invoke the Python program at the desired times (morning analysis, etc.). For example, a cron entry could run `main.py` every weekday at 8:30 AM. The Python script itself would then perform all steps sequentially. - **In-App Scheduler (APScheduler):** Incorporate the Advanced Python Scheduler to schedule jobs within the app. For instance, use `BackgroundScheduler` with a cron trigger: `scheduler.add_job(run_daily_trade, 'cron', day_of_week='mon-fri', hour=8, minute=30)` to call the main routine every weekday at 8:30 AM. You can schedule separate jobs for

different sub-tasks if needed (though a single routine can handle all steps in sequence). The scheduler runs in the background of a long-running process [14]. - **Configuration:** Provide an easy way to adjust scheduling (e.g. via a config file or environment variables for the scheduled times, trading days, etc.). Ensure the scheduler respects market holidays (perhaps simply not running if data is not available or allow manual override). - **Manual Run Mode:** In addition to automation, allow the engineer to run the loop on-demand (for testing or backfilling) by executing the main script. For example, `python main.py --date 2025-11-24` could run the logic for that date (if market was open) so we can do dry-run tests outside of schedule.

Concurrency Considerations: Since the system might fetch data and execute analysis concurrently, ensure thread-safety if using threads or asyncio (Agents SDK async calls) for parallel agent execution. The scheduling of tasks can be done sequentially at first (simpler to implement), then optimized.

CI/CD and Observability

- **Continuous Integration (CI):** Set up a suite of automated tests and quality checks to ensure the simulator remains reliable as it's developed:
- Write unit tests for each module (e.g. test the Data Pipeline caching logic, test that Risk Engine correctly flags an oversized trade, etc.) using **pytest**. Aim for good coverage on critical logic like calculations and risk checks.
- Configure **pytest** to run in a CI pipeline (this could be a GitHub Actions workflow if using GitHub, or a local script to run all tests before a release). Include assertions for expected agent behaviors on sample data.
- Use **code coverage** tools (like `coverage.py`) to monitor test coverage. This encourages thorough testing of all components.
- Optionally, incorporate **integration tests** or simulation tests that run a full daily cycle with known data (perhaps using historical data) to see if the system behaves as expected (e.g. does not crash, produces plausible trade decisions).
- **Continuous Delivery (CD):** Since the simulator runs locally, CD might simply involve packaging the project (Poetry can build a wheel) or creating a Docker container for deployment. A lightweight approach: use git tags or releases to mark stable versions, and maybe a simple script to update the live environment from the repo.
- **Logging Architecture:** Implement comprehensive logging for observability and debugging:
- Use Python's built-in `logging` module to log events with timestamps and severity levels. Configure multiple log handlers:
    - An **agent log** that records the dialogue or outputs from each agent (e.g. what each specialist agent concluded, what decisions were made by the manager). This can be at INFO level, and can help trace the reasoning path on each day.
    - A **trade log** recording each executed trade: include date/time, asset, quantity, price, and resulting portfolio positions. This can be a CSV or a structured log file for easy review or analysis.
    - An **error/alert log** at WARNING/ERROR level capturing exceptions, failed API calls, or risk limit breaches (e.g. "Trade aborted: insufficient data" or "Risk limit hit, reducing position"). These should stand out for the developer to address.
- Ensure logs rotate or prune over time to avoid unlimited growth (use a rotating file handler or daily log files).

- Implement **structured logging** for important events (JSON logs or specific format) if planning to feed logs into a monitoring system.
- **Alerts:** For critical failures (like data source down or an unhandled exception causing a trading halt), configure an alert. This could be as simple as an email or SMS (using SMTP or a service) or a desktop notification. While not strictly necessary for a local simulator, it helps to not miss issues if the process is running unattended.
- **Observability & Monitoring:** Given this is a local system, a full Prometheus/Grafana stack is optional but can be very useful:
- **Metrics Collection:** Use the Prometheus Python client to record custom metrics, such as number of API calls made, time taken for each phase, number of trades executed, daily profit/loss, etc. The simulator can expose these metrics via an HTTP endpoint (e.g. using `prometheus_client.start_http_server` on a port).
- **Grafana Dashboard:** If Prometheus is collecting metrics, a Grafana instance could visualize them (for example, a time series of portfolio value, a bar chart of win/loss ratio of trades, latency of agent responses, etc.). This provides a real-time view into the agent's performance and behavior trends.
- **Alternative Dashboard:** If setting up Prometheus/Grafana is overkill, you could implement a simple **Streamlit or Flask app** that reads the logs or output files and displays key information (current portfolio, P/L, recent decisions) on a local webpage. This could run on demand.
- **OpenAI Tracing:** When using the OpenAI Agents SDK, take advantage of its built-in tracing tools. OpenAI's platform offers trace visualization where you can see each agent/tool call and response. During development, enable this to debug the agent chain step-by-step. For production runs, you might disable it for performance, but it's useful for observability during testing.
- **Local CI Tools:** In the development environment, use **pre-commit hooks** to enforce tests passing and linters clean before any commit is accepted (you can configure `pre-commit` to run pytest and lint). This acts as a first line of defense. Additionally, keep a `DEV.md` with instructions so any new developer can run tests and understand the logging/monitoring setup easily.

Sanity Checks & Fail-Safes

Robustness is crucial since an autonomous system must handle unexpected situations safely. We implement multiple sanity checks and fail-safes:

- **Pre-Trade Sanity Checks:** Before any trading decision is executed, validate the data and signals:
- Ensure the input data for the day is complete and fresh. For example, verify that we have the latest price for each asset of interest (no null or stale values). If data is missing (API failure), the system should abort trading for that day rather than act on incomplete information.
- Check that each agent has returned a coherent recommendation. If an agent's output is an error or obviously nonsensical (e.g. an LLM agent returns an unrelated answer or a very large trade suggestion that doesn't fit the budget), flag it. The Agent Manager can decide to ignore that agent's input or ask for a retry (if feasible) or just proceed with other agents.
- **Risk Limit Enforcement:** The Risk & Compliance Engine ensures no trade violates predefined rules. For example, it will *set firm position size limits* for each trade (e.g. max $200 per trade for a $1000 account) [11]. If the strategy tries to allocate more (perhaps due to an agent's aggressive suggestion), the risk module will scale it down or cancel it. Similarly, enforce a max number of trades per day to prevent overtrading or "revenge trading" scenarios [15] [16]. These limits act as guardrails to prevent common pitfalls that could blow up the account.

- Check portfolio impact: if a proposed trade combined with current holdings would exceed diversification rules (say more than 50% of portfolio in one stock) or leverage the account (not applicable if purely cash account), then adjust or skip that trade.
- If **stop-loss / take-profit** levels are part of the strategy, ensure they are set for each new position (even though we might not actively stop out in simulation, it's good to record an intended stop for analysis).
- **Exception Handling:** Surround external calls and agent calls with try/except blocks. For instance, if the data pipeline raises an exception (e.g. HTTP 500 from API), catch it and log an error. In many cases, the system can proceed without crashing by handling the exception:
- If a minor data source fails, perhaps skip that data and proceed (with a warning). But if a critical piece (like price data for a stock to trade) is missing, better to abort the trade.
- If an agent (LLM) call times out or fails, decide on a fallback: maybe retry once, or use a simpler heuristic. Always ensure the exception is logged and doesn't propagate uncaught to bring down the scheduler.
- **Health Checks:** Implement lightweight health checks for the system's components at start-of-day:
- E.g. a **data source ping**: attempt a quick fetch of a known small piece of data (like a known stock price) at 8:00 AM to confirm the data API is reachable. If this fails, you have time to retry or switch sources before the trading logic kicks in.
- An **API key check:** verify that required API keys (OpenAI, data APIs) are set and valid (perhaps by making a trivial request) before the agents start. This was also suggested in the OpenAI example setup [17] [18].
- **Agent response check:** possibly send a test prompt to each agent at startup (or have a diagnostics mode) to ensure they load correctly. For example, ask the LLM agent a simple question in each domain to see if it returns an answer. This could be optional but helps ensure everything is wired up.
- **Fail-Safe Modes:** If something goes wrong mid-run:
- If the data pipeline fails and no alternate source is available, the system should **gracefully cancel trading for that day**. The Agent Manager can log "Data unavailable, skipping trades on 2025-11-24" and the scheduler can try again the next day. It is safer to do nothing than to trade on partial data.
- If the strategy agents produce conflicting signals or no clear consensus, the Agent Manager might decide to **skip trading** that day (or choose the most confident signal). It's a valid outcome to not trade if uncertainty is high.
- If the OpenAI API (for agent reasoning) is down or exceeds quota, the system should revert to a minimal heuristic strategy or do nothing, rather than blindly guessing. For example, have a basic rule-based fallback (like hold cash) if the AI agents are not available.
- Incorporate a **"circuit breaker"**: If the portfolio drawdown exceeds a certain threshold (say 20% loss), the system could halt further trading and notify the developer. This prevents the simulator from driving the account to zero in case of persistent bad decisions – a chance for the engineers to review what's going wrong.
- **Transaction Validations:** The Paper Trading Engine can include sanity checks like: do not allow negative cash (if a trade would overspend cash, adjust down or reject it), and ensure share quantities are whole numbers and positive. Essentially double-check the trade instructions before "executing".
- **Logging & Alerts for Fail-Safe Events:** Any time a fail-safe triggers (trade aborted, system halted, fallback engaged), make it very visible in the logs and any dashboards/alerts. For instance, log an ERROR level with a clear message like "Risk check failed – trade canceled", or "Data missing – no trades today". This helps in debugging and improving the system's reliability over time.

By implementing these layers of checks, we ensure the autonomous agent operates within safe bounds and any unexpected condition leads to a controlled shutdown or skip, not a chaotic trade.

Sprint Breakdown

To build this system, we propose an iterative development plan spread across 5 sprints (2 weeks each, roughly 10 weeks total). Each sprint delivers a set of incremental capabilities:

Sprint 1: Scaffolding & Agent Runner

- **Objectives:** Set up the project structure, development environment, and get a minimal agent loop running.
- **Tasks:**
- Initialize a new Python project with Poetry (or venv) and required dependencies. Configure linting/formatting and set up a git repo.
- Create the fundamental module structure: e.g. directories for `agents/`, `data/`, `trading/`, plus a `main.py` entry point. Stub out classes for AgentManager, DataPipeline, RiskEngine, etc., with method signatures and `NotImplemented` placeholders.
- Integrate the OpenAI Agents SDK: write a simple test agent that uses the OpenAI API. For example, a dummy agent that echoes a greeting or makes a trivial decision. This is to verify that we can call the OpenAI API (or local LLM) successfully and that our environment keys are set.
- Implement a basic **Agent Manager loop** in `main.py` that, for now, just calls one dummy agent and prints a result. This is the skeleton to be expanded in later sprints.
- **Deliverable:** By end of Sprint 1, the project should run a no-op trading cycle: e.g. start up, log "Hello, hedge fund!" from an agent, and terminate. All packaging and dev tools should be in place, so development in further sprints is on solid ground.

Sprint 2: Core Data Ingestion & Caching

- **Objectives:** Build the Data Pipeline module to fetch and cache data from open sources. Ensure the system can gather real market data as needed.
- **Tasks:**
- Implement the `DataPipeline` class with methods to retrieve price data (using Yahoo Finance or equivalent). Test it by fetching historical prices for a known ticker (e.g. 1 month of AAPL data) and verify the data schema.
- Add support for at least one additional data type, e.g. fundamental metrics or news. For instance, implement a method to fetch a company's P/E ratio and last earnings date from an API, or fetch the latest 5 news headlines from an RSS feed. This will exercise making different API calls and parsing JSON/XML responses.
- Introduce a simple **caching layer**: perhaps use a local `cache/` directory where after fetching data, we save it (as CSV or JSON). Implement caching logic in the DataPipeline methods (check if cache file exists and is recent before calling API). Use timestamps or data freshness criteria for cache invalidation (e.g. price data can be cached per day, news maybe per hour).
- Handle rate limiting and add logging for data requests. Possibly use the `time.sleep()` or a counter to not exceed free API limits (document any known limits in code comments).
- Write unit tests for DataPipeline: e.g. test that two calls to `get_price_data` in quick succession only hit the API once (second uses cache), simulate an API failure with a mock and see if retry works.
- **Deliverable:** By end of Sprint 2, the system can successfully fetch required market data and store it. A demo might involve a small script to print today's price of a stock and a cached fundamental metric. The foundation for data-driven decisions is now ready.

Sprint 3: Agent Logic – Strategy & Risk/Compliance

- **Objectives:** Develop the core decision-making agents: the strategy specialist agents and the risk management agent. Integrate them via the Agent Manager using the OpenAI Agents SDK.
- **Tasks:**
- Implement the **Fundamental, Technical, and Sentiment Strategy agents**. Each can be a function or class that the Agent Manager will call (possibly through the SDK tool interface). For example, create `fundamental_agent.py` with a function `analyze_fundamentals(ticker)` that uses the DataPipeline to get fundamentals and returns a simple recommendation (it could use an LLM prompt: e.g. "Given these financial metrics, is the stock undervalued? Respond with buy/hold/sell." using GPT-4, or a heuristic rule). Do similarly for technical (maybe check if price above 50-day average) and sentiment (e.g. count positive vs negative news).
- Enable the **Agent Manager** to call these agents in sequence or parallel. Using the Agents SDK, register each strategy agent as a tool that the main agent can invoke [5]. Alternatively, orchestrate manually: call each agent function from code and collect results. Ensure the outputs are normalized (e.g. each agent returns a structured result like `{"action": "BUY", "confidence": 0.8, "ticker": "XYZ"}` or similar).
- Implement the **Risk & Compliance Engine** logic. This could be a simple class with a method `assess_trades(trades, portfolio)` that iterates through proposed trades and applies rules. Develop rules such as:
    - Not more than X% of portfolio in one stock,
    - Not trading if less than Y data available,
    - For compliance, maybe a placeholder like "don't trade penny stocks" if needed.
    - Output an approved list of trades or modifications (e.g. clip quantities).
- Integrate the risk check into the Agent Manager's flow: after gathering strategy agent recommendations, call the risk engine to filter them.
- At this stage, decisions can be **simulated** without actual execution: e.g. log what trade would be done. This lets us test the agent logic thoroughly before hooking the trading engine.
- Use an example scenario to test end-to-end: e.g. use historical data from yesterday, run the agents (maybe for a specific ticker or a small set), and have them produce a dummy trade suggestion which the risk engine evaluates. Confirm via logs that each piece ran in the right order and data was flowing (this might be run in a dev/test mode, not live scheduling yet).
- **Deliverable:** By end of Sprint 3, the multi-agent decision workflow is functional: given fresh data, the system's agents produce a trading decision that passes through a risk filter. We should be able to see in logs something like "FundamentalAgent: recommends BUY AAPL", "TechnicalAgent: recommends HOLD AAPL", "SentimentAgent: recommends BUY AAPL", "PortfolioManager: decided BUY 5 shares AAPL", "RiskEngine: approved 5 shares AAPL". No actual trade applied yet, but the logic is in place.

Sprint 4: Paper Trading Execution & Logging

- **Objectives:** Build out the execution layer to actually simulate trades and finalize the logging and monitoring components. At the end of this sprint, the system should be running end-to-end (ready for trial runs).
- **Tasks:**
- Implement the **Paper Trading Engine**. Define a Portfolio state structure (perhaps a dict or a small class with cash balance, holdings dict, etc.). Include methods like `execute_trade(trade)` which adjusts the portfolio. For example, if trade is "BUY 5 AAPL at $150", ensure sufficient cash (cash -= 5*150), add 5 shares to holdings. If selling, ensure holding exists then increase cash accordingly.

Handle edge cases like partial sells (just reduce holding). For simplicity, assume market orders at current price; no need for order book simulation.

- Keep a **trade ledger** (could be a list or file) where each call to execute_trade records an entry. Include fields: date, ticker, action, quantity, price, and perhaps reason (from which agent or strategy).
- Connect the Paper Trading Engine to the Agent Manager flow: after risk approval, for each trade, call `execute_trade`. If a trade is rejected by risk, skip it and log that it was skipped.
- Update the **Dashboard/Reporting** component. Possibly generate a daily summary after trades: e.g. print the new portfolio holdings and cash, and P/L versus yesterday. If using a dashboard, update its data source.
- Enhance Logging: finalize the log formats for agent outputs and trades. Ensure that all important info from the day's run is captured. For instance, log the recommendations of each agent, the final decision, and trade results. This might involve formatting log messages or writing a summary to a file at end of day.
- **Scheduling Integration:** If not already done, integrate the scheduling for daily runs. For example, use `APScheduler` in the main program to schedule `agent_manager.run()` every day at the set time. Alternatively, document how to set up an external cron. Test the scheduled run (you can simulate by triggering the scheduled job manually).
- Conduct a **dry-run simulation** using historical data for multiple days: This is like a mini backtest using the system. One way is to freeze the system date and feed historical data day by day (since we are not connecting to a live broker, we can try using past data to simulate trades). For instance, simulate the week of Jan 1–5, 2025 by overriding data pipeline to load those dates. This will test if the system can handle sequential days and maintain portfolio state correctly.
- Iterate on any issues found (for example, if portfolio goes negative due to some bug, fix that; if an agent output is always "None", adjust prompt or logic).
- **Deliverable:** By end of Sprint 4, the hedge fund simulator is fully operational in paper trading mode. It can run through a full cycle: ingest data, make decisions, execute trades, and log outcomes. The team should be able to run the simulator for a single day or schedule it for continuous daily operation. Logs and (if implemented) the dashboard will show the actions taken.

Sprint 5: Testing, CI/CD, and Final Refinements

- **Objectives:** Rigorously test the system in various scenarios, set up continuous integration workflows, and perform final refinements before "launching" the simulator for extended runs.
- **Tasks:**
- Write additional **unit and integration tests** covering edge cases: e.g. test risk engine with a trade that should be rejected, test data pipeline handling of API failure (by mocking a 500 response), test that the portfolio calculations are correct after a sequence of trades (perhaps simulate a known series and verify final cash/holdings).
- Perform **load testing** in a simulation context: e.g. run the system for a month of historical data in a loop to see if any performance bottlenecks or memory issues arise. Ensure caching is effective (monitor that API calls per day remain within limits).
- Set up the CI pipeline (if not done) so that all tests and linters run on commits. Aim for a near-100% pass rate and fix any flakiness in tests.
- Finalize **documentation:** Write a README or technical documentation for the system describing how to install, configure (API keys, etc.), and run it. Include instructions for scheduling (cron or otherwise) and how to interpret the logs/dashboard. Also document the agent roles and any configurable strategy parameters (for example, risk thresholds or which stocks to trade).

- Polish the **observability stack:** If using Grafana, create a basic dashboard JSON and include it in the repo. If using Streamlit, finalize the app layout. Make sure these monitoring tools don't interfere with the main trading loop performance (maybe run them in a separate thread or process if needed).
- **Dry Run & Review:** Do a final dry run of the entire system in as real a scenario as possible. For instance, run the simulator each day for a week in a controlled manner, perhaps using current live data but not during market hours, to ensure everything triggers correctly. Monitor the logs for any unexpected behavior (like an agent timing out or a data cache miss).
- Address any remaining issues (tune agent prompts if the LLM outputs are not ideal, adjust risk limits if too tight or too loose based on test trades, etc.). This is the hardening phase.
- **Deliverable:** By the end of Sprint 5, the project is production-ready (for a paper trading simulation). We have confidence through tests and dry-runs that it operates autonomously and safely. The CI pipeline ensures future changes won't break core functionality, and documentation + dashboards make it maintainable for the engineering team. The $1,000 paper trading hedge fund simulator is now ready to run locally and be observed over time.

Each sprint builds on the previous, and by following this plan, a software engineer can incrementally develop the autonomous trading simulator with clear checkpoints and quality controls. The end result is a locally-running, multi-agent hedge fund simulation platform that leverages OpenAI's agent orchestration capabilities with free data sources, complete with risk management and transparency at every step.

---

[1] [3] [6] [9] GitHub - EfthimiosVlahos/Hedge_Fund_Agents: A proof-of-concept AI-powered hedge fund that simulates trading decisions using multiple AI agents. Designed to explore algorithmic investing through fundamental analysis, technical indicators, sentiment analysis, and risk management.
https://github.com/EfthimiosVlahos/Hedge_Fund_Agents

[2] [4] [5] [7] [8] [12] [13] [17] [18] Multi-Agent Portfolio Collaboration with OpenAI Agents SDK
https://cookbook.openai.com/examples/agents_sdk/multi-agent-portfolio-collaboration/multi_agent_portfolio_collaboration

[10] Multi-Agent Hedge Fund Simulation with LangChain and LangGraph | by Mubin Shaikh | Medium
https://shaikhmubin.medium.com/multi-agent-hedge-fund-simulation-with-langchain-and-langgraph-64060aabe711

[11] [15] [16] Day Trading Risk Management: Tips for New Traders | Topstep
https://www.topstep.com/blog/risk-management-for-day-traders/

[14] How to Schedule Python Scripts to Run Automatically: A Beginner's ...
https://python.plainenglish.io/how-to-schedule-python-scripts-to-run-automatically-a-beginners-guide-252b1da1d243