Roadmap for Building

Slice

: A Discretionary Macro Workstation

Slice will be built in phases over 12 months, focusing first on robust data and process pipelines, then layering on the human intuition engine and LLM integration, and lastly adding UI polish. The guiding strategy is **"process-first, UI-last"** – ensuring the quant and intuition engines are solid before worrying about front-end features. The plan is broken into phases with clear goals, design choices, and deliverables, followed by technology comparisons, schema definitions, and a sprint-by-sprint execution plan.

Phase 1: Problem Framing & Architecture (Month 0–1)

Purpose & Goals: In this initial phase, clarify Slice's scope and design a high-level architecture. We identify all required components (data sources, quant engine, intuition engine, LLM module, database, scheduler, UI) and how they interact. The outcome is a detailed specification and project plan that the small dev team (1–3 people) can follow.

Key Design Decisions: Choose fundamental tech stack pieces upfront, aligning with the project constraints (open-source, moderate skills, $3k/year budget):

- **Backtesting Library:** Decide between **VectorBT** and **Backtesting.py** (or others like Backtrader) for the quant engine. We favor a library that supports multi-asset strategies (since macro theses involve several ETFs) and is efficient. *Evaluation:* VectorBT offers blazing speed and multi-asset vectorized backtests (millions of trades in seconds), ideal for testing many scenarios or parameter sweeps. However, it has a steeper learning curve and requires "vectorized thinking" (no step-by-step Python loops), and some advanced features are behind a paywall. Backtesting.py is simpler and great for quick prototyping with concise syntax, but it **does not support multiple assets in one strategy** (only single-asset backtests), which is a deal-breaker for our macro use-case. **Backtrader** is another candidate: it's beginner-friendly with an intuitive event-driven API and good documentation, supports multi-asset and multi-timeframe strategies out of the box, and can integrate with live trading easily (important since we plan broker integration).

Backtrader's downside is slower execution (Python loops) and it's no longer actively developed, but it "just works" for many and is a proven Swiss-army knife for backtesting. **Decision:** *Use Backtrader for the initial quant engine*, due to its ease of use and multi-asset support, while keeping an eye on performance. We will prototype with Backtrader (and possibly simple custom backtests), and later, if needed, incorporate **VectorBT** for more computationally heavy analyses (parameter sweeps or large scale scenario simulations). This hybrid approach mitigates risk: start simple, and adopt VectorBT for speed-critical tasks once the team is comfortable.

- **Database & Data Model:** Use **PostgreSQL** as the single source of truth, with the **pgvector** extension for embeddings. This meets the requirement for structured storage (for time series, trades, theses) and vector search in one system, avoiding complex multi-DB infrastructure. We prefer direct SQL/psycopg for fine-grained control (the team is comfortable with SQL), but will use SQLAlchemy with pgvector bindings for convenient ORM if it doesn't add too much overhead (SQLAlchemy's support for vector columns is available via plugins). The schema will be carefully designed (see Phase 4 and Schema section) with JSON fields or separate tables for structured data like theses and observations.
- **LLM Service:** Use **OpenAI API** (GPT-4/GPT-5 class) for all LLM needs, and OpenAI's embedding model (e.g. text-embedding-ada-002) for generating vector representations of notes. We will call the API directly via the OpenAI Python client, avoiding heavy frameworks initially. This keeps things simple and within known quality – OpenAI models are state-of-the-art and we assume budget covers their usage. The design will be modular enough to swap in an open-source model or another API later, but v1 will lean on OpenAI for reliability.
- **Automation Framework:** Plan how to run scheduled tasks for "Level 4" continuous updates. Likely use **APScheduler** within our Python app for flexibility (scheduling jobs to update data, run checks, etc.), or system cron jobs if deploying on a single VM. APScheduler allows in-app scheduling of jobs with different intervals, which suits our need for daily/weekly tasks without setting up a separate service.
- **Execution Layer Interface:** Define an abstraction for trade execution (ExecutionAdapter). Even though live trading comes later, designing the interface now ensures we log trades uniformly. A SimulatedExecutionAdapter will simulate fills (e.g. always fill at next market open price) and record trades in the database. Later we'll implement SchwabExecutionAdapter that uses the broker's API (the user has some Schwab integration already – we'll integrate that code here). By coding to an interface, the core system (quant engine and LLM advisor) can call execute_order(order) without caring if it's sim or real. We will specify the interface in this phase (methods for placing orders, querying positions, etc.).

Architectural Diagram: We formalize the system architecture. The diagram below shows the major components and data flows in Slice – from data ingestion to quant engine to LLM modules and how they connect via the PostgreSQL database.

High-level Slice architecture: Market and macro data flows into a central Postgres DB, feeding the **Quant Engine** for backtests and risk calculations. The user's **observations** (intuition logs) are parsed by the **LLM** into structured form and vectorized into the same DB (with pgvector). A background **Scheduler** triggers regular updates (data refresh, metrics recompute, scenario checks). The **LLM** and **Quant Engine** collaborate to produce insights (narratives, alerts, Q&A), delivered to the user via a minimal **Dashboard** or CLI interface.

Subtasks & Deliverables:

- *Requirements Refinement:* Write a detailed spec of use-cases (e.g. "Log a thesis with drivers/disconfirmers and get an automated critique"), data requirements (list all assets and indicators from FRED needed), and system capabilities at each "Level" of automation.
- *Architecture Spec:* Produce diagrams and descriptions of how components interact (as above). Define the data model rough draft (entities: TimeSeries data, Thesis, Observation, Trade, Scenario).
- *Tech Choices Confirmation:* Draft a document comparing options (some of which we did above with citations) and lock in decisions for Phase 2 onward – i.e., confirm Backtrader (with potential VectorBT later), confirm Postgres/pgvector setup, confirm using OpenAI API, etc.
- *Project Backlog & Timeline:* Break the project into phases (which we're doing now) and then into 1–2 week sprints with specific goals (we will detail the sprints in the Execution Plan section). This backlog will serve as a roadmap for the team.

Pitfalls: In this phase, a potential pitfall is analysis-paralysis – spending too long evaluating tools. To avoid this, we set a *one-month limit* to finalize choices. We use known community feedback to guide us (e.g. acknowledging that **Backtesting.py cannot do multi-asset**, which justifies elimination). Another pitfall is designing an overly complex architecture for a single-user v1. We mitigate that by sticking to one database and one app server process. The architecture can scale out later if needed.

Success Criteria: By end of Phase 1, we have a clear blueprint and everyone on the team understands the plan. All major tech decisions are made and documented with reasoning. We should have a simple "Hello World" test of each critical component to de-risk them: e.g. connect to Postgres locally, call the OpenAI API once, perhaps run a trivial Backtrader strategy on dummy data – ensuring the pieces are reachable. We'll be ready to start building for real in Phase 2.

Phase 2: Data Pipeline & Data Infrastructure (Month 1–2)

Purpose & Goals: Establish a **robust data pipeline** for market and economic data – the foundation for the quant engine. In this phase we will set up connections to data providers (Twelve Data, yfinance, FRED API) and populate our database with historical data for all required assets and indicators. We'll also create processes for ongoing data refreshes (e.g. daily close prices, monthly economic releases). The goal is to have clean, queryable time-series data for all the ETFs, FX rates, and macro series listed in the Data Coverage (SPY, QQQ, TLT, TIP, DXY, EURUSD, GLD, XLE, CPI, unemployment, etc.).

Key Design Decisions:

- **Data Sources & Redundancy:** Use **Twelve Data** as the primary source for daily price history of ETFs and FX (since the user has an API subscription). For redundancy and perhaps to fill history gaps, allow fallback to **yfinance** for equities/ETF prices and to fredapi for macro series. We will design the pipeline to try primary source first, and if data is missing or API limit hit, automatically fall back to secondary. This ensures resilience.
- **Storage Strategy:** We opt to store time-series data in Postgres tables. Each asset/series might have its own table or we use a unified schema (e.g. a table market_data (ticker, date, open, high, low, close, volume) and econ_data (series_id, date, value)). Given moderate data volumes (a few decades of daily data for maybe 20-30 tickers), Postgres can handle it easily. Storing in the DB makes the data readily available to the quant engine and to LLM (if we want to include recent data points in prompts). Alternatively, we considered keeping data in CSVs or Pandas only, but a database ensures **consistent updates and central access**. We'll likely create some indexes on date for fast querying.
- **ETL Tools:** Use Python scripts for ETL. The team can use **pandas** to handle CSV/JSON responses from APIs and transform to the DB schema. We might not need a heavy ETL tool; a simple custom script or Jupyter notebook (later turned into a scheduled job) is sufficient. We will utilize libraries: twelvedata Python SDK if available, otherwise requests to hit Twelve Data's REST endpoints; yfinance for Yahoo data; and fredapi for FRED series.
- **Data Frequency & Updates:** Our use-case is daily or lower frequency (macro monthly data). So we plan a **daily job** to fetch the latest daily close for each asset after market (or next morning), and a **monthly job** for macro data (or trigger macro update when new data is released – FRED provides release calendars or we can just check daily and see if new point appears). We also plan a backfill script to run in this phase: it will fetch full historical data for each symbol (e.g. 20+ years for SPY, or whatever available since inception for ETFs, and sufficiently long macro history).
- **Quality and Adjustments:** We must consider corporate actions (splits, dividends) for ETF equity data. Yahoo data via yfinance usually provides adjusted close; Twelve Data can also return adjusted series. We will ensure to use adjusted prices to reflect total return

if possible (or explicitly track dividends separately if needed). Given these are broad ETFs and indices, corporate actions are minimal (e.g. SPY dividends). We will document whether our data is total return or price return – for now, price return is fine, but we might later integrate dividends for total-return calculations (perhaps using an index like SPXT for total return).

- **Metadata:** Create reference data entries for each asset (name, asset class, perhaps category like "Equity ETF" vs "Rate ETF" vs "FX" vs "Macro indicator"). This will help the LLM or UI to display and also for grouping in risk analysis.

Subtasks & Implementation Steps:

1. **Set Up Database:** Install Postgres (if not done) and pgvector extension. Create tables for price data. For example, a prices table with columns: ticker VARCHAR, date DATE, open, high, low, close FLOAT8, volume FLOAT8. Alternatively, one table per ticker (less ideal). We might prefer one table for all market prices and one for macro series for simplicity.
2. **Historical Backfill Script:** Write a Python script backfill_data.py:
    - For each equity/ETF symbol (SPY, QQQ, EFA, EEM, SGOV, IEF, TLT, TBF, RINF, GLD, XLE, etc) and FX (DXY or UUP, EURUSD=X, etc), fetch full history. e.g. using Twelve Data's batch API if available (to minimize API calls), or yfinance which can fetch years in one call.
    - For each macro series (CPI, Core PCE, Unemployment, Fed Funds Rate, GDP, PMIs, etc.), use fredapi to fetch the series by ID (we'll need to know FRED series IDs like CPIAUCSL for CPI, etc).
    - Normalize the data and insert into DB. (Use SQLAlchemy bulk insert or copy for efficiency if data is large).
    - Log or print summary of how many points inserted for verification.
3. **Data Update Jobs:** Write functions for incremental update:
    - update_daily_prices() – to be run daily: it checks last date in DB for each symbol and fetches new data up to today. If the market is closed today or data not ready, it can safely do nothing.
    - update_macro_data() – to be run maybe daily or weekly: for each macro series, check last date and fetch latest. (FRED updates often have a lag or come monthly – this is fine, our check will just not find new data except when available).
    - Integrate these with a scheduler (though scheduler setup comes in Phase 7, we can for now just manually run them or schedule via cron after testing).
4. **Data Validation:** Ensure data makes sense (no extreme outliers or gaps). For critical series, do a quick plot or statistical summary as a sanity check (e.g. check there are no zero prices or dates misaligned). If issues (like a gap due to API limit), use fallback source or adjust our API queries (maybe split into smaller intervals).
5. **Redundancy Plan:** Implement a fallback in code: e.g. if Twelve Data call fails or returns incomplete data, automatically try yfinance for that symbol. (We can test one or two known symbols to ensure this works.)

6. **Store in Memory (optional):** For backtesting speed, sometimes having data in-memory (Pandas DataFrames) is useful. We decide that the quant engine can just query the DB when needed (Postgres is fine for daily data queries), but we might also implement a caching layer: e.g. a simple Python dictionary of dataframes, loaded at app start. Because the number of series is not huge (~ dozens) and daily points maybe a few thousand each, it's feasible to load all into memory for quick access. We will make this an optional step – not strictly needed if DB is fast with indexes, but could optimize later.

Pitfalls: One risk is hitting API limits – since we have a modest number of symbols, and this is a one-time backfill + daily updates, we should be within free or paid limits of Twelve Data. But we will implement sleep/retry logic if needed. Another pitfall is schema design mistakes (e.g. not indexing by date, causing slow queries later). We address that by adding indexes on (ticker, date) and perhaps clustering the table by ticker. Also, timezones could be an issue (we'll store dates in UTC or as plain date without time since daily). We must ensure consistency (e.g. FRED data is end-of-month or specific dates; align them to proper date fields).

Deliverables by end of Phase 2:

- A populated database with historical data for all required instruments and indicators.
- Python scripts or notebooks for data fetching (manual run for backfill, and incremental update routines).
- Confirmation that we can run a sample query (e.g. pull last 5 years of SPY prices from DB) quickly.
- Documentation of how to add a new data series if needed (so the system can grow).

Success Criteria: If a team member or the PM can query any asset's history from the DB and plot it, and the data matches an external source for accuracy, this phase is successful. Also, running the update script should properly add the latest day's prices. We should also have evidence that the data covers the needs of the example theses (e.g. we have real yields via RINF or similar if needed, etc.). Essentially, we have a **data foundation** ready for analysis.

Phase 3: Quant Engine Sandbox (Month 2–4)

Purpose & Goals: Develop the **Quantitative Engine**, i.e., the sandbox where the PM can backtest ideas, compute risk metrics, and evaluate scenarios. In this phase we build the tooling to answer questions like: "If I went long gold when real yields fell, how would that perform?" or

"What's the 1-month VaR of my current portfolio mix?" The engine will provide: strategy backtesting, portfolio aggregation, performance analytics, risk calculations (volatility, correlations, drawdowns), simple factor/regression analysis, and scenario simulation.

Key Design Components:

- **Backtesting Framework Implementation:** We decided to start with **Backtrader**, so we set it up and create a base Strategy class that can handle multi-asset positions. We'll implement simple strategies reflecting the theses. For example, a strategy template that takes signals (like "when indicator X true, go long asset Y") and executes trades. Given our use-case is discretionary macro, many "strategies" are actually conditional trades rather than high-frequency signals. We might implement a simplified backtest: for each thesis, define entry/exit rules and let it run through historical data to see P&L. *For instance:* Thesis: "Long USD vs EUR on policy divergence" – in backtest, this could translate to going long a USD index or long USD/EUR when a certain interest rate differential widens and exiting when it narrows. We'll encode such logic either directly in code or via parameterized rules. Backtrader supports multiple data feeds (we can feed it EURUSD, interest rate differential as a custom indicator, etc.). We will write small strategy classes for each prototype thesis to test the engine.
- **Portfolio & Risk Analytics:** We incorporate **PyPortfolioOpt** (or portions of it) for advanced analytics. PyPortfolioOpt can compute covariance matrices and optimize weights; even if we don't need to optimize, its risk models (shrinkage covariance, semi-variance, etc.) could be useful. We will also use **pyfolio** (Quantopian's pyfolio) or our own code to generate performance metrics and tear-sheet-like analysis. Pyfolio, if we integrate it, can take a returns series and output Sharpe, drawdown, monthly returns breakdown, etc. (We have to ensure it's maintained or use the fork from ml4trading). Alternatively, we manually calculate metrics:
    - **CAGR**, **Sharpe ratio**, **Max Drawdown**, **Volatility (std dev)**, **Calmar ratio**, etc.
    - Rolling metrics: e.g. a function to compute rolling 6-month volatility or rolling correlations between assets.
    - **Factor regression:** Use statsmodels to run regressions of strategy returns vs macro factors (e.g. regress a portfolio's returns against S&P, Dollar index, etc., to see exposures). Statsmodels provides easy OLS so we can do, for example, a 3-factor regression to identify what drives the PnL of a strategy.
- **Regime Detection (Basic):** Implement a basic regime detection utility – e.g., a rolling clustering on macro indicators to label periods as "high inflation" vs "low inflation", etc., or something like yield curve regime (steepening vs flattening). This can be done by simple threshold rules initially (e.g. if 10Y–2Y spread is negative, call it an "inversion regime"). We can later refine with clustering algorithms. But having a notion of regime in the quant engine helps scenario analysis (the system can say "we are currently in a high-inflation, low-growth regime, historically that did X to your strategies").
- **Scenario Analysis Tables:** Provide a way to simulate scenarios: For example, the PM defines a scenario (like "Fed hikes 100 bps, Oil +20%, EURUSD parity") and the engine will output an estimated impact on each asset or portfolio. We implement this by storing

simple factor sensitivities or using historical analogs. For instance, if we know roughly the beta of gold to real yield changes, we can estimate outcome. Another approach: identify a historical period similar to the scenario (e.g. use the 2013 taper tantrum as a scenario for rapid rate rise) and apply those asset moves to current positions. We will start simple: create a JSON for scenario (see Schema section) with key variable changes and a function that uses linear sensitivities to compute P&L impact. The sensitivities could be estimated by regression in the engine (e.g. regression of gold returns vs real yield changes gives a coefficient; use that to project impact). This gives us a "scenario matrix" of outcomes.

- **Risk "Rails" Implementation:** Establish basic risk limits checks. For example:
    - Maximum portfolio allocation to one asset or one theme (e.g. not > 50% in one trade).
    - Correlation check: if the PM's active trades are too correlated (e.g. all long inflation trades), flag it.
    - VaR calculation: use historical simulation VaR or a normal approximation. For instance, compute 95% 1-month VaR of current portfolio by looking at past 1-month returns distribution of a similar asset mix (approximate by weighted sum of asset returns).
    - Stop-loss reminder: if a backtest shows a drawdown beyond X, note it, so the PM can set a mental stop.

These risk rails are mostly analyses rather than enforcement (since human makes decisions), but the system will **alert the PM** when a risk limit is hit (e.g. "Portfolio 1-month 95% VaR is 8%, exceeding your 5% threshold – risk is high!").

Technologies & Implementation Notes:

- We will use **Backtrader's** API to run backtests across multiple assets. We create a data feed for each relevant series (Backtrader has a notion of Cerebro engine where we add data feeds). We might need to extend Backtrader's Analyzer or Observer to collect custom metrics during runs (or just analyze after the fact). If Backtrader shows to be too slow for some reason, we'll adapt by limiting test periods or using vectorized numpy for parts of calculations. (Alternatively, test out VectorBT on one strategy to compare speed).
- For performance metrics, if using Pyfolio, we convert Backtrader output to a returns pandas Series and feed into pyfolio's create_full_tear_sheet. If pyfolio is outdated, we manually compute needed stats and use Plotly or matplotlib to plot equity curve, drawdown chart, etc.
- We incorporate **Matplotlib/Plotly** to generate charts (we won't focus on UI now, but we can generate and save charts to ensure the analysis is correct).

- For factor modeling, use **Statsmodels** OLS (e.g. sm.OLS) to regress monthly returns of strategies on things like MSCI World returns (for beta) or other macro factors. This gives quantitative backing to the PM's intuition about drivers.

Detailed Subtasks:

1. **Backtest Basic Long/Short Trade:** As a starter, code a simple strategy in Backtrader, e.g. "Buy SPY if 50-day moving average > 200-day, else sell" just to validate our setup with our data. This ensures Backtrader is pulling our Postgres data correctly. (We might need to feed data by reading from DB into Pandas, then to Backtrader – or write a custom Backtrader DataFeed that pulls from DB on the fly. Simpler: load data into Pandas first).
2. **Thesis Strategy Prototypes:** For each example thesis the user gave, implement a prototype strategy:
    - *Gold vs Real Yields:* Create an indicator for "real yield" (maybe use RINF ETF or use 10yr yield minus breakeven inflation from FRED). Strategy: when real yield falls below some threshold or moving average, go long GLD.
    - *USD vs EUR on policy divergence:* Use ECB vs Fed rate differential (FRED has policy rates). When US–EU rate spread rising, go long USD (short EURUSD). Represent this by long UUP (Dollar ETF) or short FXE (Euro ETF).
    - *Bull Steepener (yield curve bet):* This could be long TLT (long duration bonds) and simultaneously short something like TBF (inverse 20yr) to simulate curve steepening? Actually bull steepener = long short-term bonds + long long-term bonds? Or long short-term, short long-term? We express via ETFs: perhaps long TLT and long SPTS (short-term) when expecting steepening? We need to clarify, but for sandbox, we can simulate by an approximate strategy.
    - *Overweight Energy on supply tightness:* When oil inventory or capex low (no direct data initially), maybe use moving average of oil price or backwardation as a signal to overweight XLE. We might skip complex signals and just allow user to manually say "from date X to Y, I would be overweight XLE" and backtest that (like a manual trade backtest).

      In short, some strategies will be rule-based, some might just be testing static trades (we can simulate a static trade by entering on a date and exiting on another – which Backtrader can do with a timed strategy or we simply slice data).

3. **Run Backtests & Validate:** Execute these strategy backtests over a reasonable period (last 10–15 years) and gather performance metrics. Ensure the results make sense (e.g. the gold strategy yields better in low real yield environments, etc.). Iterate if needed on logic.
4. **Analytics Module:** Develop a analyze_performance(results) function that takes backtest results and computes:
    - Total Return, CAGR.
    - Annual volatility.

- Sharpe (we assume a risk-free rate ~0 for simplicity or take 3M T-bill from FRED).
- Max Drawdown and maybe Calmar ratio.
- Save the equity curve and drawdown curve for reference.
- Use rolling 1-year windows to compute rolling Sharpe and volatility; produce a simple plot or output summary of worst period.
- If multiple strategies, compute correlation of their returns to see diversification benefit.

5. **Risk Metrics on Current Portfolio:** If the PM has a current portfolio (maybe derived from active theses and weights), we should compute its risk. This means summing up the time series of P&L of each position (weighted by size) to get portfolio returns, then same stats as above. This can be done by storing all strategy returns and then allowing the PM to simulate combining them.
6. **Factor Exposure Analysis:** For each strategy or the overall portfolio, run a regression of its returns vs major factors (S&P 500, US Dollar index, etc.). Use monthly frequency for statistical significance. Output the beta coefficients and R^2. For example, if the energy overweight strategy has a +0.5 beta to oil prices, that's expected.
7. **Scenario Simulator:** Implement a function simulate_scenario(portfolio, scenario):
   - For each asset in the portfolio, assign a sensitivity to each scenario variable (could be as simple as delta = 1 for itself, or something we derive).
   - For example, scenario: "Inflation +1% surprise, Fed hikes +50bps, Oil +20%". We estimate SPY might drop X%, TLT might drop Y%, GLD up Z%, etc. These estimates could be based on historical correlation or guess. Initially, we can hard-code plausible moves or use linear regression on past shock events.
   - Apply those moves to current positions to estimate P&L. Produce a table of scenario impact.
   - Also generate *opposite* scenario (like inflation -1%) to see asymmetric effects.

     This is somewhat subjective, but even a rough scenario table is useful to challenge the portfolio.

8. **Risk Alerts Logic:** Implement checks:
   - If any single position > some fraction (say 30%) of total portfolio, flag concentration.
   - If portfolio 1M VaR (we can derive by taking 95th percentile of 22-day return distribution of portfolio returns) is high relative to a threshold, flag it.
   - If two positions have >0.8 correlation and both large, flag "lack of diversification".
   - If a strategy consistently lost money historically in similar regimes to now, flag that (requires regime logic).

     These can be simple if-else checks after computing relevant stats.

Pitfalls & Mitigations:

- *Complexity overreach:* We must be careful not to over-engineer sophisticated models that the team can't maintain. For example, building a full Monte Carlo VaR engine or an AI regime classifier might be too much now. We stick to simpler methods (historical VaR, threshold regimes) that are easier to implement and understand.
- *Backtesting bias:* Ensure we avoid look-ahead bias in our backtests (e.g. if using future data accidentally). Using Backtrader should handle that, but we must be cautious when we feed indicators (if we use Pandas for custom logic, align data properly).
- *Performance:* Backtests with daily data over ~15 years * number of strategies are fine. But if we try a massive parameter sweep or very long series, Backtrader could slow down. If needed, we'll limit date ranges or later switch to VectorBT for those specific tasks. VectorBT can vectorize and even run thousands of parameter combos quickly – but we only do that if absolutely needed for optimization tasks.
- *Integration:* At this stage, the quant engine is somewhat standalone (not yet integrated with LLM or UI). We should still design it to be called by higher layers later. For example, the results could be stored in the DB or as JSON so the LLM can access them (or the UI can query them). We plan to store summary metrics in the DB's "scenario" or "trade" tables for reference.

Success Criteria: By end of Phase 3, we have a working quant sandbox where:

- We can run a backtest for a given thesis and get performance metrics printed out.
- We can calculate current portfolio risk stats given a set of positions.
- We can produce a scenario impact report for a hypothetical event.
- These outputs should make sense and be trusted by the PM (we might validate by comparing with known outcomes: e.g. does our backtest show gold did well in 2020 when real yields fell? If yes, good).

The quant engine doesn't need to be *fully polished* by this point, but it should be functional enough to support the next phases (where LLM and intuition engine will query it or use its outputs). We will likely revisit and refine the quant engine in Phase 8 (Evaluation), but Phase 3 establishes the core capabilities.

Phase 4: Thesis & Observation Schema Design (Month 4)

Purpose & Goals: Now that data and quant capabilities are in place, we shift focus to the **Human Intuition Engine**. First step: define structured schemas for the qualitative inputs/outputs – specifically the **Thesis JSON schema** (for representing a macro thesis in structured form) and the **Observation (intuition log) JSON schema**. We'll also define schemas for trades and

scenarios for completeness. This phase is about data modeling and database adjustments: how do we store a thesis, what fields does it have, and how do we link observations (journal entries) to theses and to outcomes.

Key Design Decisions:

- **Thesis Schema:** A **Thesis** represents a top-down macro idea the PM has. We decide to store each thesis as a JSON or as columns. Considering we want to use LLMs to fill or critique parts of it, a JSON structure is convenient (and can be stored in a JSONB column in Postgres for flexibility). We enumerate fields:
    - id: a unique identifier or slug.
    - title: short name of thesis (e.g. "Long Gold on Falling Real Yields").
    - hypothesis: a full description of the thesis – essentially the PM's reasoning in prose.
    - drivers: an array of key **drivers** that should propel the trade if thesis is right (e.g. "Real interest rates decline").
    - disconfirmers: an array of **conditions that would invalidate** the thesis (e.g. "Inflation expectations rise significantly" could disconfirm a bullish gold thesis).
    - start_date (and maybe end_date or review date): when the thesis was initiated (and optionally when it's closed or to be re-evaluated).
    - asset or expression: how the thesis is expressed in trading – could be a list of positions, like expression: [{asset: "GLD", position: "long", size: 10% portfolio}]. In v1, the PM might not allocate sizes precisely, but we capture at least which instruments and direction. (Size or conviction can be a field too, e.g. low/medium/high conviction ranking).
    - status: active, closed, or watchlist.
    - tags: category tags (e.g. "inflation", "Fed").
    - notes: any free-form notes.

  We also link a thesis to data drivers. Perhaps an explicit field monitor_indices that lists which data series correlate to this thesis outcome (for automation to monitor). For example, in the gold thesis, monitor_indices could include "10yr TIPS yield" or "real_rate = 10yr yield - breakeven". This way, the system knows what to watch for disconfirmation (if real_rate jumps instead of falls, that's against the thesis).

  The schema might look like:

```
{
  "id": "THESIS001",
  "title": "Long Gold on Falling Real Yields",
  "hypothesis": "If real interest rates decline due to higher inflation or
lower nominal rates, gold will appreciate.",
  "drivers": ["Declining real yields", "Erosion of policy credibility",
"Rising inflation expectations"],
  "disconfirmers": ["Real yields rising", "Fed unexpectedly hiking
aggressively", "USD strengthening significantly"],
```

```
  "expression": [{"asset": "GLD", "direction": "long", "size_pct": 15}],
  "start_date": "2025-01-15",
  "status": "active",
  "tags": ["inflation", "gold", "rates"],
  "monitor_indices": ["US10Y_TIPS_Yield", "DXY"],
  "notes": "Gold often performs well when real returns on bonds are low or
negative."
}
```

- We will formalize this and consider using **Pydantic** models to validate it in code (ensuring types and required fields).
- **Observation (Intuition) Schema:** An **Observation** is a single journal entry or note from the PM, capturing subjective intuition or reflections. We want to store these in a structured way as well. However, observations are free-form text by nature. The LLM will help structure them. We decide on fields like:
    - id: unique ID.
    - timestamp: when the observation was recorded.
    - text: the raw text of the entry (we may keep the original narrative for reference).
    - thesis_ref: a reference to which thesis(es) this observation relates to (if mentioned). Could be a list of thesis IDs or null if general market thought.
    - sentiment: the PM's sentiment or emotion, if detectable (e.g. "confident", "anxious", "confused"). The LLM can classify this from the text.
    - categories: topics present (e.g. mentions of "Fed policy" or "recession fears").
    - actionable: a boolean or enum indicating if the observation implies an action or just a thought.
    - embedding: we will likely not store the full embedding vector in JSON (since pgvector column handles it), but perhaps store a reference or store it in a separate column in the observations table of type VECTOR(1536). Alternatively, we make a separate table observations and another observations_embedding linking id to vector, but pgvector allows vector in main table, so we might do embedding vector(1536) column.

  Example observation entry:

```
{
  "id": "OBS2025-03-01-1",
  "timestamp": "2025-03-01T18:30:00Z",
  "text": "Noticed that despite Fed's hawkish tone, long-term yields fell
today. I feel more confident in my bull steepener idea. However, I'm a bit
anxious that equity markets are too quiet.",
  "thesis_ref": ["THESIS003"],
  "sentiment": "confident with some anxiety",
  "categories": ["yield_curve", "Fed", "market_volatility"],
  "actionable": "monitoring (no immediate action)",
  "embedding": "[…vector data…]"
}
```

- The sentiment and categories here could be generated by an LLM classifier. We might define a fixed set of sentiment labels (e.g. Very Bullish, Bullish, Neutral, Bearish, Very

Bearish, or emotional states like Calm, Excited, Anxious). For now, storing a text label is fine.

Behavioral metadata: We also want to incorporate "psychological RAG scoring". We interpret this as a Red/Amber/Green status of the PM's mindset – e.g. **Red** if the PM is acting on tilt or under fear/greed extremes, **Green** if balanced. We can derive this from observations (LLM could classify each entry into these categories). We might add a field mood_score or bias_flag in observations. E.g. bias_flag: "overconfidence (Red)" or mood: "Anxious (Amber)". This will help later analytics to warn the PM ("Your recent notes show potential overconfidence bias – take a step back").

- **Trade Schema:** Although execution comes later, define a simple trade log structure:
  - trade_id, datetime, asset, action (Buy/Sell), quantity, price, type (Simulated/Real), thesis_ref (if this trade is tied to a particular thesis), and maybe notes.

    This is straightforward and can be stored as a SQL table (each field a column).

    Example:

```
{
  "trade_id": "TRD0001",
  "timestamp": "2025-03-02T15:45:00Z",
  "asset": "GLD",
  "action": "BUY",
  "quantity": 100,
  "price": 180.50,
  "type": "SIMULATED",
  "thesis_ref": "THESIS001",
  "notes": "Initial entry for gold thesis."
}
```

- We'll ensure the trade schema aligns with what the ExecutionAdapter will use.
- **Scenario Schema:** Define how to represent a scenario (for scenario analysis results or definitions):

  We can have:

  - scenario_id, name,
  - assumptions: a dictionary of key variables and shock magnitudes (e.g. {"Inflation": "+1%","10Y Yield": "+50bps","Oil": "+20%"}),
  - portfolio_impact: results such as expected P&L or asset class moves (e.g. a dict of asset => % change or a narrative "Equities down ~5%, Treasuries down ~2%").
  - description: text explaining scenario narrative.

  Or separate scenario definition vs scenario result objects. Simpler: one JSON with assumptions and expected outcomes (the outcome could be filled after running simulation).

Example:

```
{
  "scenario_id": "SCEN_STAGFLATION",
  "name": "Stagflation Shock",
  "assumptions": {
      "GDP_growth": "-2%",
      "Inflation": "+3% (surprise)",
      "FedFunds": "+1.00%",
      "Oil": "+30%"
  },
  "expected_impact": {
      "SPY": "-15%",
      "IEF": "+5%",
      "GLD": "+10%",
      "USD": "-5%"
  },
  "description": "Recession with high inflation causes equities to sell off,
bonds rally (as Fed eventually eases), gold jumps, dollar weakens on policy
uncertainty."
}
```

- We'll store scenarios either in a JSON column or as separate tables (scenario table and scenario_results table). JSON is fine for flexibility.

Implementation in DB: We will likely create tables:

- theses (with at least id TEXT PRIMARY KEY, data JSONB or columns for key fields plus JSONB for details).
- observations (id, thesis_id (nullable), timestamp, text, sentiment, etc., embedding VECTOR).
- trades (structured columns as described).
- scenarios (JSONB for scenario definition, plus maybe separate column for name).

We might also create linking tables if many-to-many relations (e.g. one observation can relate to multiple theses, in which case an association table or we allow an array of thesis IDs in the JSON). Simpler: allow array in JSON for thesis_ref as above.

Subtasks:

1. **Design JSON Schemas:** Write down the expected JSON structure for Thesis and Observation. Use examples (like above) to refine fields. Get feedback from the PM (the user) to ensure it captures what they normally write down. Possibly iterate a bit – maybe

the PM wants an "objective" field (what the goal is) or a "time horizon" field. We include anything relevant.

2. **Implement Pydantic Models:** In Python code, create Pydantic classes Thesis and Observation reflecting the schema. This will help when the LLM output is received – we can parse the JSON into these models to validate types.
3. **Database DDL Update:** Write migrations or SQL commands to create the new tables:
    - CREATE TABLE thesis (id TEXT PRIMARY KEY, title TEXT, status TEXT, data JSONB);

      We might store most fields in data JSONB for flexibility. But for querying ease, we could also duplicate some fields as columns (like status, start_date) if we want to easily query active theses. We'll do at least status and start_date as columns.

    - CREATE TABLE observation (id TEXT PRIMARY KEY, thesis_ids TEXT[], timestamp TIMESTAMP, sentiment TEXT, categories TEXT[], raw_text TEXT, embedding VECTOR(1536));

      Here we store embedding in the table with pgvector, which allows similarity searches via an index (CREATE INDEX idx_obs_embedding ON observation USING ivfflat (embedding vector_cosine_ops);). We include an array of thesis_ids if multiple related, or could have one and assume one primary thesis per observation (but journal entries can cover multiple points, so array is better).

    - CREATE TABLE trade (trade_id SERIAL PRIMARY KEY, timestamp TIMESTAMP, asset TEXT, action TEXT, quantity FLOAT, price FLOAT, type TEXT, thesis_id TEXT, notes TEXT);
    - CREATE TABLE scenario (scenario_id TEXT PRIMARY KEY, data JSONB);

      This scenario table can hold both assumptions and results inside data JSON.

      (Alternatively, separate scenario_def and scenario_result, but not needed for now.)

4. **Populate Sample Data:** After creating, insert one or two sample theses and observations (maybe from historical memory or dummy) to ensure the schema works. E.g., create a thesis JSON like above and insert it. Query it back to check JSON structure is intact.
5. **LLM Prompt Templates for Schema Filling:** Start drafting how we'll get the LLM to produce these JSONs. For example, when the PM writes a freeform thesis description, we want an LLM to extract drivers and disconfirmers. Or when they write a journal entry in plain text, we want LLM to output an Observation JSON. We outline these prompt formats now (though actual integration is next phase, it's good to consider what fields might be hard for LLM to infer).
    - For thesis: We can prompt GPT-4 with: *"Parse the following thesis description into a JSON with fields: hypothesis, list of drivers, list of disconfirmers, etc."* The LLM likely can do it, but we have to ensure the PM actually provides that info. Alternatively, we might have the user enter drivers/disconfirmers via UI form. In

early stages, it might be easier to have the PM list them and then just store directly. We decide that *for initial version, the user will fill out a template for a thesis (like a form or just enumerating in text), and we use the LLM mainly to verify or refine it.* We will however use LLM to interrogate (ask questions) rather than rely on it to generate the thesis itself.

- For observation: definitely use LLM to structure it, because user will journal in free text. So we note that we'll create a prompt like: *"You are an assistant that labels journal entries for a trader. Extract sentiment (one of: {list}), list any mentioned thesis by title if present, and main topics. Output a JSON."*

We don't implement this now, but having schema defined means we can craft these prompts.

Pitfalls:

- *Schema Creep:* It's easy to overcomplicate the schema (too many fields). We aim for a **minimal schema that still captures the necessary context**. For example, we considered adding a "confidence level" numeric field for a thesis. The PM might informally have that, but if it's not naturally provided, we skip it for now (or derive from how strong the language is using LLM later). We focus on what will be used: drivers and disconfirmers are key for automation, expression is key for linking to trades, and dates for tracking.
- *Flexibility vs Rigor:* Using JSONB gives flexibility (we can add fields later without migration), but also means less relational integrity. We'll rely on application logic (Pydantic validation) to keep things consistent.
- *Mapping observations to theses:* If an observation mentions a thesis indirectly (e.g. referencing "gold" but not explicitly the thesis title), the LLM might or might not link it. We should not expect perfect linking. To mitigate, maybe whenever the user writes a note, they tag which thesis it's about (we can encourage that in UI by letting them select relevant thesis). We'll plan for both: allow manual linking, and have LLM guess links for additional context.

Success Criteria: By the end of Phase 4, we have:

- Clearly defined JSON schemas for all key entities (thesis, observation, trade, scenario).
- The database tables created to store them.
- The system (via a simple script or admin UI) can accept a new thesis entry or observation and save it correctly.
- We should be able to run a sample query like: "get all active theses and their drivers" or "list last 5 observations with sentiment". This ensures our structure is queryable.
- No actual LLM processing is done yet, but the stage is set for it – meaning we know exactly what we want the LLM to output. This structure is crucial for the next phase where we harness GPT to fill these in.

Phase 5: Intuition Logging & Embedding Integration (Month 5–6)

Purpose & Goals: Implement the **Human Intuition Engine**: capture the PM's qualitative insights and store them with vector embeddings for later retrieval and analysis. In Phase 5, we connect the OpenAI API to process raw text observations into structured JSON (as per our schema) and generate embeddings for semantic search. We also establish the workflow for updating the rolling "memory" of observations (embedding new ones, possibly re-embedding older ones if needed) and basic analysis of these embeddings (clustering, similarity).

Key Steps:

- **Observation Ingestion via LLM:** When the PM writes a journal entry (for example, in a plain text form or even in a markdown file), we will use the OpenAI **GPT-4** to convert it into our Observation JSON structure. We'll create a function process_new_observation(text) that does the following:
    1. Takes the raw text and perhaps some context (like date, maybe the list of current thesis titles to help it recognize references).
    2. Crafts a prompt for GPT-4: e.g.

```
You are an assistant for a portfolio manager's journal. Read the entry below
and extract:
- sentiment tone (one of: very bullish, bullish, neutral, bearish, very
bearish, or an emotional state),
- any specific thesis mentioned (match by known titles if possible),
- main topics or categories (single words),
- whether the PM suggests an action.
Provide output in JSON with keys: "sentiment", "thesis_ref", "categories",
"actionable". The "thesis_ref" should be an array of thesis IDs if any thesis
is discussed; "categories" an array of key topics.
Entry: \"\"\"[journal text]\"\"\"
```

    2. and instruct to only output JSON. Temperature = 0 for consistency.
    3. GPT-4 responds with JSON (we'll have to ensure we catch errors if it goes off-script, but GPT-4 with a clear prompt and temp0 usually works).
    4. Parse the JSON (via Pydantic) to an Observation object.
    5. Save the observation in the DB (assign a new id like using current timestamp or sequence).

We will likely refine the prompt as needed to get good output. Perhaps we also include "mood" vs "market sentiment" explicitly. But we can iterate after some testing with sample texts.

- **Embedding Generation:** After structuring the observation, generate its vector embedding using OpenAI's embedding model. We'll use openai.Embedding.create(model="text-embedding-ada-002", input=observation_text) which returns a 1536-d vector. We then store this vector in the observation.embedding field (pgvector column). We also maintain an in-memory vector index if needed, but since pgvector can handle similarity search, we'll rely on Postgres for queries (with an index as noted).

  We design this such that it's done for each new observation on the fly (the cost of one embedding call per note is fine). For older notes (if any exist pre-v1, the user might input some retrospective notes to build initial memory), we can batch embed them once.

- **Embedding Index and Search:** Ensure the pgvector index is in place. Implement a function search_observations(query_text, k) that:

  - Takes a query (which could be a question or a thought), gets its embedding via OpenAI, then does a vector similarity search in the observations table:

```
SELECT id, text, sentiment, thesis_ids
FROM observation
ORDER BY embedding <-> query_embedding
LIMIT k;
```

  -
  - using <-> operator for cosine distance (we stored normalized or use appropriate opclass).
  - Returns the top-k similar past observations. This is essentially the **Recall** part of RAG memory: find relevant past intuitions. We will use this later for context when the LLM is summarizing or detecting patterns.

  Test this by adding a few dummy observations and querying. For example, if we have an observation about "Fed hiking" and we query "Fed policy", it should retrieve that note, confirming embeddings work.

- **Clustering & Pattern Recognition:** Use unsupervised learning on the embeddings to find recurring themes:
  - We can run a simple KMeans on all observation vectors (maybe the number of clusters can be decided, say 5 clusters) to see if they group by topic (e.g. all "inflation fear" notes together, etc.). Or use hierarchical clustering to see a dendrogram of thoughts. This doesn't have to be in real-time; it could be an analysis we run monthly.

- We also consider using dimensionality reduction (UMAP or PCA) to project the intuition space for visualization later (perhaps not now, but as a tool to see how the user's thoughts evolve).
- The result of clustering can feed into a "themes summary". For example, if one cluster is mostly observations with "anxiety" sentiment and mention of "volatility", that might indicate a recurring worry. We could label it (the LLM can help label clusters by providing it some example texts from each cluster and asking for a theme name).

Implement a function analyze_intuition_history() that:

- Groups last N months of observations into clusters.
- Uses GPT-4 to generate a short summary of each cluster (like "Cluster 1: Frequent concerns about Fed tightening and market volatility").
- Detects if the latest observation falls into a particular cluster that had negative outcomes in the past (if we have performance data tied, though that might be advanced).
- Compute simple stats: e.g. out of last 20 observations, X were bullish, Y bearish – to quantify if the PM is overly bullish lately.

These analytics will be used to produce feedback to the PM on their own behavior (like a "meta" analysis of their intuition – think of it as a behavioral coach). We may not fully implement the feedback in this phase, but lay groundwork.

- **Rolling Window Management:** Since we plan a rolling 12–24 month window for intuition, we need a strategy for older notes:
  - We won't delete old observations (they can be valuable to revisit), but maybe our active analysis focuses on last 2 years. If the dataset grows very large (hundreds of notes), the embedding search might slow a bit but still fine (pgvector can handle a few hundred or thousand easily). If it grows to many thousands over years, we might consider archiving older than 2 years or summarizing them.
  - A method could be **summary consolidation**: e.g., every quarter, take all observations from >2 years ago and have the LLM summarize them into one or a few "memory summary" entries, embed those, and remove the originals from the vector index (or mark them archived). This is analogous to how some memory systems summarize long conversations into chunks.
  - We won't implement archiving in v1 unless needed, but we note the plan for it. For now, assume number of observations is manageable.
- **Psychological RAG (Red/Amber/Green) Scoring:** Using the sentiment and possibly comparing observation content to outcomes, assign a color score reflecting the PM's current psychological state:
  - Red could be triggered by signs of overconfidence (many bullish statements after big gains) or panic (many fearful notes after losses). Amber for moderate concern, Green for balanced. This is somewhat subjective. We can use LLM to classify directly: e.g., prompt GPT-4 with the last week's worth of observations: "Given these journal entries, does the PM exhibit any of: overconfidence, fear, or neutral

mindset? Answer with one: Red (overconfident or reckless), Amber (cautious or slightly emotional), Green (rational/collected)."
    - This "state" can then be stored or displayed. It's a soft indicator just for self-awareness. Implementation-wise, we might add a field in the DB for the latest psychological state or store as part of daily summary.

Technologies:

- **OpenAI GPT-4 and Ada Embeddings:** We rely on OpenAI's API heavily here. Ensure to handle API errors (network issues, rate limits). We might implement exponential backoff or retries for embedding calls if they fail. Observations aren't so time-critical that a slight delay matters.
- **OpenAI Function Calling:** Instead of prompt engineering JSON, we could use the new function calling feature. E.g., define a function schema for Observation and let GPT fill it. This might yield more reliable structured output. We should try this if available with our model (GPT-4 supports it). It would require using the OpenAI Python SDK's function calling interface. This could eliminate JSON parse errors. We will attempt function calling for extraction tasks.
- **Scikit-learn:** Use scikit-learn's clustering (KMeans) for the embedding cluster analysis. Also use numpy for distance computations if needed outside DB.
- **NLTK or Sentiment library:** We might incorporate a simple sentiment analysis library for cross-check against LLM (like VADER sentiment to gauge tone as bullish/bearish). But GPT's understanding should suffice for now, especially as we want a more nuanced "trader mood" not just positive/negative.

Subtasks:

1. **LLM API Integration:** Set up OpenAI API keys and test a simple request from our environment (e.g. prompt "Hello" to GPT-4) to ensure connectivity.
2. **Function: process_new_observation:** Implement as described. Test it on a few example texts (we can write some mock journal entries and see what JSON GPT returns, adjusting prompt until it's correct). For example, test with: "I'm worried about the yield curve inversion deepening, but stocks keep rallying. Maybe I'm wrong about an imminent recession." The function should output sentiment ("bearish" or "anxious"), thesis_ref (if any known thesis matches "recession thesis"), categories ("yield_curve", "equity_market"), etc. Iterate to get good results.
3. **Store Observation & Embed:** After getting JSON, insert into observation table and call embedding API for the text. Store that vector. (This could be combined with step 2: we might embed the raw text or we might embed a processed version. Likely best to embed the raw original text plus maybe key extracted info, since the raw captures nuance. We could also embed the combination of text + extracted categories to enrich it. But initially, just embed the raw note text itself).

4. **Similarity Search Test:** Add multiple observations and use our search_observations("some query") to verify we get sensible nearest neighbors. For example, if we query for "yield curve", it should bring up notes about yield curve inversions, etc. This confirms our embeddings are capturing semantics.
5. **Clustering & Summaries:** If we have enough sample data (maybe we can simulate ~10-20 observations by writing dummy ones covering different topics), run KMeans and see if grouping makes sense. Then feed representative texts of each cluster to GPT-4 to label them. This is more of a back-end analysis – we might not need to surface cluster labels explicitly to the user yet, but it's useful internally (and can be part of a periodic behavioral report).
6. **Behavioral Stats:** Implement a routine to compute counts of sentiment categories over the last N observations. E.g. out of last 10 notes: 7 bearish, 3 bullish – that could indicate a pessimistic bias (maybe after a downturn). We can decide what to do with that info (could just present it to the user or have LLM comment on it).
7. **Memory Decay (optional):** If time, implement a time-decay in similarity search. One way is to adjust similarity score by a decay factor based on age (recent notes weighted higher). This can be done in application code by multiplying distances or via a custom distance function. Since pgvector doesn't natively know about time, we might do this manually: filter to last 2 years before searching, or use a hybrid approach (search, then post-filter or re-rank results by recency if needed). We note this but might leave tuning for later if search results are satisfactory.

Pitfalls:

- *LLM misinterpretation:* Sometimes GPT might hallucinate categories or link to wrong thesis. We mitigate by providing it the list of valid thesis IDs/titles in the prompt context ("Theses: [THESIS001: Title1, THESIS002: Title2,…]. If the entry mentions any of these, include their IDs in thesis_ref."). This ensures it doesn't make up thesis names.
- *Privacy/Cost:* All journal content is sent to OpenAI – since this is personal and presumably the user is fine with that under API terms. The cost per entry (one GPT-4 call + one embedding call) is small given likely a few sentences. Still, we'll keep an eye on usage (embedding is ~$0.0004 per entry for 1536 dims, negligible; GPT-4 maybe $0.03 per prompt if short). Under $3k/year this is fine unless user writes a novel every day.
- *Vector storage issues:* Need to ensure the vector dimensionality matches what we declare (1536). Also, Postgres's ivfflat index requires a certain number of vectors before it's efficient – we might not reach that threshold early. That's okay; even sequential search over hundreds of vectors is super fast.
- *Overfitting behavioral analysis:* We should avoid drawing strong conclusions from small data. The RAG color is just an indicator, not an absolute truth. We'll treat these outputs as suggestive.

Success Criteria: By end of Phase 5:

- We can input a free-form observation and the system stores a structured version and embedding without manual intervention.
- We can query similar past observations given a topic and get reasonable matches.
- The PM's intuition logs are now accessible for the LLM to use in conversations (like a knowledge base of the PM's thoughts).
- We have initial means to summarize or categorize the intuition history.

Essentially, the "memory" is up and running: the last 1-2 years of the PM's reflections are stored in a vector database and can be retrieved for context. This sets the stage for Phase 6, where we use the LLM *with* this memory and the quant engine to actually provide insights and critiques.

Phase 6: LLM Analytical Toolkit Integration (Month 6–7)

Purpose & Goals: Leverage the LLM (GPT-4/5) to enhance analysis: have the AI **interrogate theses**, ask "Socratic" questions, catch inconsistencies, and synthesize information from both the quant engine and the intuition memory. Essentially, we build a suite of LLM-based agent capabilities that act as the PM's intelligent assistant. This includes: (a) an LLM-driven **Thesis Reviewer** that reads a thesis and generates constructive critiques or probing questions, (b) an **Inconsistency Detector** that looks across all active theses for contradictions or overlaps, (c) a **Thematic Summarizer** that summarizes clusters of observations or answers queries about the PM's past notes, and (d) a **Narrative Generator** that can produce a daily or weekly commentary combining quant results and the PM's intuition.

Key Features to Implement:

- **Thesis Critique Q&A:** For each active thesis, we want the LLM to play devil's advocate and also sanity-check it. We'll implement a function like review_thesis(thesis_id) that:
    - Fetches the thesis details (title, hypothesis, drivers, disconfirmers, expression, etc.).
    - Possibly also fetches recent related data (like values of key drivers, performance of the expression so far).
    - Prompts GPT-4 to analyze it. For example:

```
You are an expert macro economist and risk manager. A portfolio manager has
the following investment thesis:
Title: Long Gold on Falling Real Yields
Hypothesis: "...(text)..."
Drivers: ["Declining real yields", "Policy credibility issues"]
Disconfirmers: ["Real yields rising", "Fed hawkish surprise"]
```

```
Current data: Real 10Y yield = 0.5% (down from 1% a month ago); Gold price =
$1800 (up 5% in 1 month).
Task: Critique this thesis. Ask 2-3 Socratic questions challenging the
assumptions or highlighting risks. Also identify if any driver/disconfirmer
is missing or contradictory.
Format: Output a JSON with "questions": list of questions, and "notes": any
additional inconsistencies or suggestions.
```

-
-
    - The LLM would output questions like:
        - "What if real yields fall because inflation expectations are unanchored – could that actually hurt gold if policy responds?"
        - "How will gold perform if the USD strengthens significantly (an omitted factor)?"

          and notes maybe: "You listed Fed hawkish surprise as a disconfirmer, yet you are long gold which typically correlates with USD; consider FX impact."

    - We parse that JSON and possibly store it or present directly to the user. This fulfills the idea of the LLM interrogating the ideas.
    - We ensure the LLM does **not** make numerical predictions itself beyond using given data (in prompt we only give it needed data, not ask it to forecast).

  We might incorporate results from quant engine here: e.g., if backtest for this thesis yielded a max drawdown of X, we can mention that in prompt like "Historical backtest max drawdown 20%". The LLM can then question if the PM can stomach that, etc.

- **Cross-Thesis Consistency Checker:** If multiple theses are active, ask the LLM to examine them for conflicting positions or assumptions:
    - For example, one thesis is long gold (implying inflation fear or dollar down) and another is long USD vs EUR (implying strong USD expectation). These two could conflict (strong USD often pressures gold down). We want the LLM to spot that and raise a flag.
    - We implement check_thesis_consistency(all_theses):
        - Provide GPT-4 a summary of each thesis (title + one-liner).
        - Prompt: "Review these theses together. Identify any contradictions or overlapping bets. If two theses would both lose/win in the same scenario, note that correlation. If they assume opposite macro outcomes, point that out."
        - The LLM might output: "Thesis A (long gold) and Thesis B (long USD) appear contradictory: a rising USD often coincides with higher real yields, which would hurt gold. Be cautious of these offsetting bets."
        - We deliver that to the PM so they are aware.
    - This ensures the PM sees the portfolio-level picture, not just thesis silos.
- **Knowledge Base Q&A:** Now that we have a vector store of observations, we can enable the PM to query it in plain English. This is like a personalized chatbot that knows the

PM's past thoughts. We integrate a simple **RAG (Retrieval-Augmented Generation) pipeline**:

- When the user asks something like "Have I ever been this concerned about inflation before?" or "What was I thinking the last time the Fed paused rate hikes?", we:
  - Convert the query to embedding, retrieve top relevant observations (from Phase 5).
  - Feed those excerpts plus the question into GPT-4, asking it to formulate an answer or summary.
  - For instance:

```
User question: "What were my thoughts during the 2024 inflation spike?"
Context (retrieved notes):
- [2024-06-10]: "Inflation hit 5%. I'm worried the Fed is behind the curve.
Considering adding to TIPs..."
- [2024-09-15]: "Inflation seems to be peaking, I'm less anxious now, maybe
rotate back to stocks..."
Assistant: using the above journal entries, answer the question.
```

-
  -
  -
  - GPT-4 then answers: "In mid-2024 you were very concerned about high inflation and doubted the Fed's response, even considering inflation-protected bonds. By September 2024, you noted signs that inflation was peaking and your anxiety had decreased, shifting focus back to equities."
  - This is extremely useful for reflection – the PM can query their own memory quickly.
- We will implement a chat-like interface or at least a function for this Q&A. This involves building a prompt template that includes the retrieved snippets and the question, instructing the LLM to use them and not hallucinate beyond them (classic RAG approach). If needed, we use few-shot examples to encourage it to quote from entries.
- We also ensure that if no relevant notes are found, the system says "I don't recall any entries on that topic" rather than making something up.

- **Daily/Weekly Brief Generator:** Automate a brief that combines **quant data + intuition**:
  - Possibly a **Daily Dashboard Summary**: Each day after market close, compile key info:
    - Portfolio performance: e.g. "Portfolio +0.3% today, driven by XLE +2%, GLD -1%."
    - Any driver/disconfirmer triggered: e.g. "CPI came in high, which challenges Thesis Y's premise."
    - New observation sentiment: e.g. "You noted feeling anxious about valuations."
    - Then have GPT-4 summarize this in a few sentences, in a tone the PM prefers.
  - This requires feeding the LLM structured daily data. We can gather:
    - P&L from the quant engine (if we mark-to-market the portfolio daily).

        - A list of any alerts (from risk rails or disconfirmers).
        - Maybe the last observation text or sentiment.
        - And prompt:

```
Summarize today's update for the portfolio manager:
- Performance: ...
- Notable events: ...
- Sentiment: ...
Give a brief commentary (2-3 sentences) combining these.
```

        -
            -
            -
            - LLM might output: "Your portfolio rose 0.3% today, thanks largely to energy gains offsetting a dip in gold. However, today's higher CPI print contradicts your 'disinflation' thesis – an issue to monitor. You noted some anxiety about valuations, which is understandable given the market's rally."
        - This daily note can be delivered via UI or even email. It helps ensure nothing falls through cracks and connects the qualitative and quantitative dots.
- **LLM Boundaries & Safety:** We strictly ensure the LLM **does not replace the PM's decision**. It should not say "I think you should sell X" on its own – it can raise questions or highlight risks. We will incorporate in prompts a system message like: *"You are not to give explicit trade recommendations or make up data. Focus on analysis of given information."*

  Also, avoid LLM doing math – any numeric analysis we feed it from our calculations. If we want it to compute something simple (like sum positions), we'll do it in Python and give it the result rather than rely on the LLM's math.

Agent Patterns: In implementing these, we are effectively building an LLM **agent** that can use tools (the tools being: vector DB retrieval, database queries, running backtests). We might consider using the **ReAct** pattern for complex queries (Reason+Act). For example, if the PM asks a very broad question ("What's my biggest risk right now?"), an agent might decide: okay, tool1 = get risk metrics, tool2 = search observations for recent biases, then combine. Given the time, we might not build a full multi-step agent now, but design the prompts such that they incorporate relevant info via our functions. We should keep it modular so in future we could integrate something like LangChain or our own loop for multi-step reasoning if needed.

Subtasks:

1. **Thesis Review Prompt & Function:** Implement review_thesis(thesis) as described. Test it on one or two theses to see the quality of questions. Adjust prompt for clarity (e.g. ensure it knows to be polite but challenging).
2. **Multi-thesis Checker:** Implement cross_check_theses(theses_list). Possibly simpler: make a combined prompt listing each thesis in a bullet form and ask: "Do you see any contradictions or redundancies? Answer in bullet points." Test with some dummy contradictory theses to ensure it catches them.
3. **Observation Q&A (RAG)**: Implement a query_intuition_history(question):
    - Embed question, retrieve top 3-5 notes from Phase 5's search.
    - Form prompt with those notes (maybe as quoted material or summarized if very long).
    - Ask question to GPT with instruction to use those notes.
    - Test with some typical questions.

    We should be careful to not feed too many tokens; if notes are long, maybe truncate or summarize them first (the Pinecone conversational memory approach suggests summarizing long interactions to memory – similarly, if an observation is a page long, we might have to truncate to the relevant part).

4. **Automated Summary Gen:** Implement generate_daily_summary():
    - Query the quant engine for today's P&L (we can simulate or use the difference in yesterday vs today's price for each position).
    - Check any triggered alerts (e.g. in our risk monitoring table, see if any bool flags turned true today).
    - Fetch last observation's sentiment.
    - Put into prompt and call GPT-4 for summary.
    - Similarly, for weekly or monthly, we could accumulate changes (this can be an extension).
    - Set this up to run via scheduler (to be integrated next phase).
    - Test by feeding sample data and see if output is coherent.
5. **Interface Consideration:** We have these LLM-driven insights; plan how user will interact. Possibly:
    - The UI might have a "Review" button next to each thesis to get the questions.
    - A section "Assistant's Notes" where daily summary and any alerts are displayed.
    - A chat box where user can ask questions (wired to query_intuition_history and possibly extended to other tools).

    We will design UI in Phase 8, but keep an eye on how to present these results clearly (maybe use collapsible sections for context).

6. **Testing and Tuning:** Thoroughly test each function with realistic content:
    - Use historical scenarios: e.g. simulate that it's March 2020 and see if daily summary would catch "market crash event – does it mention risk?". This might be artificial but useful.
    - Ensure that if LLM output is too verbose or off-tone, adjust instructions (maybe we want a concise style).

- Also test that the LLM doesn't hallucinate facts we didn't provide. This means always giving it enough context if we ask something factual. In our usage, it's mostly analyzing user-provided data, so it should be okay.

Pitfalls:

- *Information Overload:* We must keep prompts focused. If we dump a huge thesis and many stats, GPT-4 might produce a very long critique that's hard to digest. We might explicitly ask for brevity or a certain format.
- *Token Limits:* GPT-4 has large context, but we should avoid hitting it. We'll generally be within a few pages of text, which is fine. But if in future we try to feed all theses and lots of notes at once, careful with limits.
- *Cost:* Each usage of GPT-4 in these analytical tasks has a cost. We mitigate by doing them on-demand or on schedule rather than continuously. E.g. thesis review only when PM requests or updates a thesis; consistency check only when a new thesis is added or on a weekly schedule. Daily summary is one call per day, fine.
- *Wrong Answers:* GPT might sometimes give a confidently incorrect analysis. By grounding it with data (and it doesn't have to recall external facts from training, just analyze what we give), we reduce hallucination. We also keep a human-in-the-loop: the PM will see the questions or suggestions and can judge them. If something seems off, they can disregard. Over time, we could refine prompts or fine-tune a model on our domain to improve reliability, but for now monitoring output is key.
- *Security:* Since this is personal, not multi-user, prompt injection risk is low (only the PM inputs queries). Still, we should sanitize any user question before sticking it into a system prompt context. But the user presumably won't try to break their own system with malicious input.

Success Criteria: At the end of Phase 6, the system achieves a new level of interactivity and insight:

- The PM can click a button and get thought-provoking questions about any thesis (LLM-driven critique).
- The system flags if the PM's ideas conflict (e.g. "Check overlap: Thesis A vs B").
- The PM can ask the AI "what did I last think about topic X" and get a correct answer sourced from their notes.
- A daily summary or similar report can be generated that ties together data and intuition.

This essentially means we have a working "LLM co-pilot" that uses both the **Quant Engine outputs** and the **Intuition memory** to help the PM reflect and stay disciplined. We will integrate these capabilities into the UI and automation next.

Phase 7: Automation & Continuous Monitoring (Month 7–9)

Purpose & Goals: Achieve **Level 4 automation** by having the system run in the background and continuously keep everything up-to-date without user intervention. In Phase 7, we set up the scheduling of tasks (using APScheduler or cron) for data updates, metric recomputations, monitoring of thesis conditions, and automated report generation. The goal is that Slice becomes a **"self-updating" personal workstation**: every day it pulls new data, refreshes backtest stats if needed, checks if any thesis conditions are breached, updates the embeddings index with new observations, and generates alerts or summaries. The PM should not have to manually run scripts to get latest info – they can just log in each morning and see a dashboard that's already refreshed.

Key Automation Tasks:

- **Data Refresh Jobs:** Integrate the Phase 2 update scripts into a scheduler:
  - Schedule update_daily_prices() to run each weekday evening (or early morning before the PM starts). This ensures by the time user checks, yesterday's closing data is in DB.
  - Schedule update_macro_data() daily or weekly (if macro data is monthly, daily is fine and it will usually do nothing until a new release is available, which is okay).
  - If any data source fails (API down), log it and possibly alert so we know data might be stale.
  - We might also schedule a weekly full refresh (re-fetch last few months of data to catch any revisions – macro data sometimes gets revised).
- **Quant Refresh:** Some metrics may need recalculation periodically:
  - For example, rolling correlations or volatilities that feed into risk alerts. We implement a job compute_risk_metrics() that reads current portfolio or active theses, computes updated vol, correlations, etc., and stores them (maybe in a small table or cache).
  - If using Backtrader, we don't necessarily re-run a full backtest daily unless the strategy depends on latest data (if the thesis logic is ongoing, maybe we do update it). However, since these theses are more like long-term trades, the backtest results won't change day-to-day except P&L. We could integrate with an actual portfolio tracking (i.e., update Mark-to-market P&L daily and see how the hypothetical strategy is doing). Possibly easier: each thesis has an "ongoing P&L" which we update from market moves (since we know positions).
  - In short, we likely schedule something to update each thesis's current performance (which assets are up/down how much since inception). This can be displayed or used by LLM to comment.
- **Monitor Thesis Drivers/Disconfirmers:** This is crucial automation: for each active thesis, check its defined drivers and disconfirmers against actual data:

    - If a disconfirmer condition is met, that means the thesis is at risk or invalid. E.g., if thesis expects declining inflation but latest CPI spiked up beyond a threshold, flag it.
    - Implementation: We have monitor_indices or similar for each thesis. We also have live data in DB. We create rules, e.g., "Real yield 10Y > 1% triggers disconfirmer for Gold thesis". Possibly we store such rules in the thesis JSON or separate config. We can encode a simple DSL or just Python code for now.
    - We implement check_thesis_conditions() job:
        - For each thesis, evaluate: for each driver, is it happening? (maybe not used for alert, but could track a score of how many drivers are in favor vs not).
        - For each disconfirmer, check if triggered. If yes, write an entry to an alerts table or mark the thesis status as "at risk".
        - Possibly also check time horizon: if a thesis was supposed to play out within 6 months and it's now 9 months with no result, that's an alert (maybe "Thesis has gone past its horizon – reconsider").
    - As an example, we might implement a config like: disconfirmer: {"indicator": "CPI YoY", "op": ">", "value": 0.04} to denote "CPI > 4% invalidates this". The code then queries the latest CPI from DB and compares.
    - We store active alerts so the UI can show e.g. a red icon next to that thesis.
    - The LLM daily summary will also mention these (we'll feed it if any disconfirmer tripped).
- **Update Scenario Analysis:** If we maintain scenario tables, maybe update if underlying assumptions shift. Possibly not needed unless scenarios are dynamic. We can skip automatic scenario changes; scenarios are more static definitions. But if we run stress tests regularly, we could schedule a weekly scenario impact run on current portfolio (like recalc what a 1% rate shock would do given latest positions).
    - We could at least refresh scenario outcomes with latest portfolio weights/prices. So if a scenario predicted -$X loss last month and now portfolio size changed, update to current values.
- **Update Behavioral State:** As part of daily tasks, update any derived behavioral metrics:
    - For instance, compute a 30-day moving average of the PM's sentiment (like count of bullish vs bearish notes). Or compute if their sentiment correlates with market moves (there's an interesting metric: e.g., does the PM get more bullish after the market went up – indicating possible chasing? This could be gleaned by correlating sentiment score with S&P returns, but that's advanced).
    - At least, update the RAG status via LLM: e.g., take last week's observations, ask GPT "color code the mood". This can be daily or weekly.
    - Save that status somewhere (maybe just overwrite a single-value table or a field in a "user_state" table).
    - If Red (danger) state, maybe generate an alert like "Caution: Possible overconfidence detected."
- **Alerting Mechanism:** Decide how to notify the PM of important triggers:
    - Within the app's dashboard, we'll have an "Alerts" panel (for disconfirmers, risk limit breaches, etc.). That's fine if the PM checks daily. But for immediate risks, maybe an email or push notification is warranted. Given this is a personal system, email integration can be done easily using SMTP. If budget and time permit, we

could allow the system to send an email or text when, say, a disconfirmer triggers (e.g., "ALERT: Thesis 'Long Gold' disconfirmer triggered (real yields rising)! Check your position."). We outline this, but maybe implement basic email if feasible.
    - Another approach: if the system has a front-end open, just real-time update the UI (less likely as PM might not have it open 24/7). So email is safer for now.
- **APScheduler vs Cron:** We can use **APScheduler** within a FastAPI app to schedule jobs. Or since maybe the app can be run as a background service anyway, APScheduler suffices. We'll configure jobs like:

```
scheduler.add_job(update_daily_prices, trigger='cron', hour=22, minute=0)
scheduler.add_job(generate_daily_summary, trigger='cron', hour=22, minute=30)
scheduler.add_job(check_thesis_conditions, trigger='cron', hour=23)
# etc.
```

- Use UTC or local time carefully.

  We'll also ensure on startup it can catch up if missed (though not crucial if daily).

  Using APScheduler means the app must be running continuously (we plan to deploy on a cloud VM, so yes).

  Alternatively, a simple cron on the server calling scripts could work, but APScheduler integrated is fine and easier to manage in-code.

Subtasks:

1. **Integrate Scheduler:** Add APScheduler to the project. On application startup (or in a dedicated daemon script), initialize the scheduler and add all jobs with their intervals.
2. **Test Jobs Independently:** Run each task function on its own to ensure it does what it should (e.g. simulate a disconfirmer trigger by tweaking threshold, see if alert goes to DB).
3. **Simulated Day Cycle Test:** We may simulate a day's sequence:
    - Add a new price data point, run the monitor, see if any alert triggers, generate summary. Check that summary includes the new info.
    - For example, pretend an inflation number was released by inserting it into DB and run monitor – ensure the system catches it.
4. **Logging & Error Handling:** Ensure each job logs success or error (so we can troubleshoot if something fails silently). For instance, if an API fails or a calculation throws an exception, catch it and log, and perhaps try again next cycle.
    - We might keep an "admin log" table or just use Python logging to a file.
5. **Idempotence:** Design jobs to not produce inconsistent state if run multiple times or if the app restarts. For instance, if scheduler accidentally runs update twice, it should handle duplicates (e.g., our data update should upsert or check last date so it won't insert duplicate price rows).

    - Or if the app was down and missed yesterday's update, when it comes back and runs, it should ideally backfill what it missed (maybe by looking at last date and catching up).
6. **Performance & Resource Check:** Running these tasks is lightweight (few DB queries, API calls). But we should ensure they don't overlap in a problematic way (APScheduler can run jobs in parallel threads by default; we can configure it or ensure heavy tasks are not concurrent).
    - We'll schedule them sequentially (e.g., data update at 22:00, summary at 22:30 etc., which is safe).
7. **Preparation for Phase 8:** These automated updates mean by the time we build the UI, all data is there. We might need an API endpoint for UI to fetch the latest summary or alerts. In this phase, we can implement a simple REST API with FastAPI for critical data (like /api/dashboard that returns JSON of key info: active theses, alerts, PnL summary, etc.). Or we wait until Phase 8 to do that. But good to keep in mind what endpoints or queries UI will need and ensure our data is ready.

Pitfalls:

- *Timing issues:* If using local time vs UTC incorrectly, might run at wrong time. Also ensure the machine's timezone is known. E.g., if we want after US market close, for simplicity we might just schedule at 00:00 UTC (which is 8pm NYT roughly, close enough after close).
- *Concurrent Data Modification:* If a user happens to add a journal entry exactly at the time the summary job runs, maybe minor conflict (like summary might miss it by a minute). That's acceptable; next day it will be included.
- *Resource usage:* The jobs are small, but if not careful (e.g., running backtests for every thesis daily might be heavier), it could strain memory or CPU. We'll monitor and perhaps limit frequency of heavy ops. E.g., full backtest can be weekly instead of daily.
- *Alert fatigue:* If too many trivial alerts, PM might ignore them. We should calibrate thresholds such that alerts are meaningful (maybe only flag big deviations, not every small move).
- *Edge cases:* If a thesis has no monitor indices set, skip it gracefully. If no data in DB yet for something (like just initiated a new macro series with no update yet), handle accordingly.

Success Criteria: After Phase 7:

- If the PM does nothing for a week, the system will still have up-to-date data and metrics at the end of the week.
- Any important change in conditions triggers an alert stored.
- The PM can rely on the daily summary to know what changed without manually compiling that info.

- Essentially, the groundwork for a **"hands-off" automated assistant** is laid.

The system should feel "alive": e.g. if a CPI release happens, by next login the user sees an alert "CPI high – disconfirms thesis X" and maybe the LLM comment on it in the summary, without the PM manually inputting anything. That demonstrates true Level 4 automation of monitoring.

Phase 8: Minimal User Interface & Interaction (Month 9–10)

Purpose & Goals: Develop a **minimal but functional UI** for Slice, enabling the PM to interact with the system easily. Up to now, we have a powerful back-end; Phase 8 focuses on front-end components such as a dashboard to view data, forms to input theses and observations, and possibly a chat interface for the LLM Q&A. We prioritize simplicity and clarity: the UI can be command-line or web-based, but since this may become a product later, we lean towards a web dashboard (accessible via browser, possibly built with Streamlit or a lightweight web framework). The emphasis is to deliver key information (positions, PnL, alerts, etc.) in a **readable format** and allow the PM to input new information with low friction (so they actually use it daily).

Key Design Decisions for UI:

- **Framework:** Given the team's skillset (Python backend), **Streamlit** could be an attractive option for a quick dashboard since it allows easy plotting and is Pythonic. However, Streamlit might be less flexible for custom designs and continuous background processes. Another approach is a small **FastAPI** backend serving a React or simple HTML frontend. Considering time and that this is initially single-user, **Streamlit** (or **Panel**/**Dash**) is likely fastest to implement. It can directly interface with our Python code and even display charts and run LLM calls on button clicks.
    - We will choose **Streamlit** for the initial UI due to its speed of development for data apps. We note that if this were to go multi-user, we'd likely refactor to a proper webapp (FastAPI + JS) for scalability. But for one user on a VM, Streamlit is fine.
- **Layout and Sections:** We structure the UI into a few clear sections (using headings, tabs, or columns):

    1. **Dashboard Home:** Shows an overview – list of active theses with status, current PnL of each (if available), any alerts (flag icons), and maybe an overall portfolio

metric summary. Also the latest daily summary text at the top as a "morning brief".

2. **Thesis Detail Pages:** If the PM clicks a thesis, they see details: hypothesis, drivers, charts of related data (e.g. a small chart of that asset price, or driver vs asset), the LLM's critique questions (generated on demand by clicking "Analyze" or already displayed), and an edit option (to update drivers or status).
3. **Observations/Journal Page:** A chronological list of observations (maybe in an accordion or just a table with date, sentiment icon, snippet of text). Also a text box to add a new observation. Possibly, integrate a voice or quick note mechanism (not required, but maybe later).
4. **Analytics Page (optional):** Show some behavioral analytics charts – e.g. a pie of sentiments, or timeline of the RAG score over months, cluster labels of intuition (if we have something to show).
5. **Q&A Chat:** If feasible, have a text input where the PM can ask a question (like described in Phase 6, the RAG retrieval QA). Streamlit has some limitations for interactive chat (but it's doable using session state). Alternatively, we could skip a real-time chat interface in v1 and instead allow a drop-down of predefined queries or just an "Ask" button that triggers a modal. But a simple chat history in Streamlit is not too hard. We can implement it if time permits because it's a powerful feature for the user to query the system in natural language.

- **Visualization:** Use **Plotly** or **Matplotlib** for charts embedded in the Streamlit app:
  - Plot price series for each asset, maybe performance of portfolio vs benchmark.
  - Plot distribution of returns for risk analysis.
  - Perhaps bar chart of scenario results (scenario vs portfolio impact).
  - Also use icons or colors to highlight statuses (e.g. green dot for thesis on track, red for at risk, using Streamlit's markdown or emoji).
  - These make the dashboard quickly scannable.
- **Interactivity:**
  - Adding a new thesis: We could have a form with fields or a multi-line text input. Perhaps a form where user enters title, hypothesis, etc. Or even just paste a chunk of text and click "Parse Thesis" to have LLM fill out drivers/disconfirmers automatically (that could be neat: the user writes a freeform thesis paragraph, and the system suggests structured bullets for drivers/disconfirmers which the user can accept/edit). This might be a stretch for v1, so a simple manual form might suffice, with ability to call LLM to help if needed.
  - Adding a new observation: Very important this is low friction. Ideally, the PM can open the app on phone or computer and quickly jot down a thought. A plain text area and submit button which triggers the process we built (calls LLM to structure & embed) and then shows it appended to the list.
  - Executing an analysis: e.g. clicking "Run Backtest" for a thesis to update its performance (if we allow that on demand).
  - Answering LLM questions: if not chat, maybe a button "Ask AI" next to each thesis that runs the critique, and a global "Ask AI" that pops up an input for any question.

- **Security & Auth:** Since it's a personal tool likely running on a private server or local, we might not need heavy auth. But if on cloud, at least basic HTTP auth or a secret token to prevent others from accessing it. Since user is comfortable with personal cloud deployment, we can implement a simple login (even if just an environment variable password) for safety.

Subtasks:

1. **Set Up Streamlit App Structure:** Create a app.py using Streamlit. Define navigation (Streamlit lacks native multi-page by default, but we can use st.sidebar for navigation links or the newer multipage support). Alternatively, maintain one page with sections.
    - Maybe use a sidebar with selection: "Overview", "Thesis: [dropdown to select]", "New Thesis", "Journal", "Analytics", "Q&A".
    - The overview page will compile info from DB via queries or our Python functions.
2. **Connect to Back-end Functions:** The UI will call our Python functions or queries:
    - For example, on load, fetch all active theses from DB and display.
    - When user clicks a thesis, load its JSON and parse fields to display nicely.
    - When user submits new observation text, call process_new_observation and then refresh the observations list.
    - When user uses chat, call query_intuition_history (and possibly other tools if needed).
    - Ensure these calls are not blocking too long; GPT-4 can be a couple seconds, which is acceptable in UI (just indicate "thinking…").
3. **Design Display Elements:**
    - Theses: Perhaps use st.write to show markdown for hypothesis and bullet lists for drivers/disconfirmers (which are naturally lists).
    - Could highlight if an alert on that thesis exists (e.g., show a red badge "Disconfirmed!" if that's the case).
    - Observations: we can use expandable markdown or just show "[date] sentiment: text…" truncated. Possibly color-code sentiment (green for bullish, red for bearish).
    - Charts: use st.plotly_chart for interactive or st.line_chart for quick ones. For P&L or prices.
    - Chat: maintain st.session_state for chat history, and on each query, append the question and answer.
4. **Testing UI with Dummy Data:** Before hooking up all live pieces, populate with some fake data to get the layout right. For example, create 1-2 theses and 3 observations in DB, and test the pages.
5. **Refine for Readability:** Ensure Markdown formatting is clear – e.g. bold titles, maybe use emojis (✅ for driver if it's happening, ❌ if disconfirmer triggered, etc.). Streamlit allows basic HTML/CSS hacks if needed for color, or we can just use text/emoji.
    - We want the output to be **scannable** – the user should quickly see what's important:

    - e.g. On overview: "**Long Gold** – P&L: +5% (Active) – 🟢 Drivers mostly intact" or "🔴 Disconfirmer triggered".
    - We'll incorporate those cues.
6. **Error Handling in UI:** If an LLM call fails or DB not reachable, show an error message rather than hanging. Streamlit will show exceptions by default, but we can catch and display friendly messages (like "Failed to get response, please try again").
7. **Performance Consideration:** Streamlit reruns script on every interaction which can be tricky. We'll use st.cache_data or similar for things like loading all observations or running heavy analysis to avoid redoing it too often. But careful: we want latest data, so maybe caching for just session is fine.
8. **User Acceptance Testing:** Have the user (if possible) use the UI for a day or two and gather feedback. Because UI is about usability: maybe they want a feature like exporting data or a dark mode (maybe not crucial but as a personal tool, small tweaks can improve experience).
    - Based on feedback, adjust layout or add minor features (like editing a thesis's fields, or marking a thesis closed via UI).
9. **Documentation:** Write a short user guide: how to add a thesis, how to log a note, how to interpret alerts shown, etc. While the user is the one who asked for it, documenting ensures no confusion about what each indicator means.

Pitfalls:

- *Scope creep in UI:* It's easy to get carried away adding bells and whistles (like draggable charts, etc.). We will stick to essentials due to time. Focus: Display info and input basic data. If time left, a bit of polish (like CSS or nicer formatting).
- *Streamlit limitations:* If we hit ones (for instance, it's not ideal for real-time chat streaming of tokens), we accept them. For chat, we might wait for full answer rather than stream tokens since Streamlit doesn't stream by default (there are hacks, but not needed).
- *Thread safety:* APScheduler running in background alongside Streamlit might cause issues if not careful. We might run scheduler in a separate thread when the app starts. Need to ensure DB access from both UI and scheduler is managed (use connection pooling or just ensure each thread uses its own connection). Python's GIL and our usage likely fine, but keep an eye out for any race conditions (the worst could be UI reading half-updated data during a write; but since our writes are usually single transactions, it should be okay).
- *Mobile view:* Streamlit web pages might not be fully optimized for mobile. If the PM uses it on a phone, we might need to test responsiveness. If it's not great, an alternative is to use Streamlit's sharing or an app wrapper. Possibly not critical, but something to consider.

Success Criteria: By the end of Phase 8:

- The user can access a web UI that shows all key info in organized sections.
- They can add a new thesis or note through the UI without resorting to code or DB edits.
- They can see updated data (from automated updates) in the UI each day, and any AI outputs (questions, summaries) easily.
- The UI should be reasonably intuitive: for example, a new user (maybe one of the other team members) could navigate it and understand the portfolio at a glance.

We essentially aim for a **Minimum Viable Interface** – not pretty, but functional and clear. It sets the stage for possibly more refined UI in stretch goals if needed.

Phase 9: Evaluation, Testing & Refinement (Month 10–11)

Purpose & Goals: Now that all main components are built, Phase 9 is about **rigorous evaluation** and refinement. We need to test the system end-to-end, fix bugs, fine-tune any algorithms or prompts, and ensure reliability and accuracy. Additionally, we will evaluate the system against its intended purpose: does it actually help the PM make better decisions or catch mistakes? The goal is to identify any gaps or misbehavior and address them before calling the project a success. We also measure performance (both computational and how well it meets the user's needs) to ensure it's ready for sustained use.

Key Activities:

- **System Integration Testing:** Continuously run the entire system in as real conditions as possible for a period (perhaps a couple of weeks simulated or actual).
    - We might simulate a series of events (market going up, down, macro surprises, etc.) and see if the system reacts appropriately (alerts triggered, summaries reflecting events).
    - Alternatively, run on live data for a few weeks (if time permits in project timeline) with the PM using it, and gather real feedback.
    - Ensure that each scheduled job triggers and completes as expected (check logs or create a dashboard in UI for "Last run time" of tasks).
    - Test sequences like: adding a new thesis mid-week, then that thesis's data flows through (backtest runs, any disconf monitor active).
- **Prompt/LLM Output Evaluation:** Revisit all LLM outputs and evaluate quality:
    - For thesis critique questions: Are they insightful and relevant? If some are trivial or repetitive, consider adjusting prompt (maybe ask specifically for "very challenging questions").

    - For consistency checks: Did it catch all obvious conflicts? If it missed some or gave a false alarm (maybe two theses that are actually compatible but it thought not), refine instructions to be more precise.
    - For daily summary: Is it focusing on the right things? If the PM says "I'd prefer if the summary highlighted X more", then adjust how we feed info to it. Perhaps weighting or filtering what we include. Maybe the daily summary should mention top gainers/losers in portfolio; if not included, add that data.
    - For Q&A: Check if any hallucination occurs. Possibly create a few known-edge queries and verify the answers only use provided context. If hallucinating, we may need to enforce answers to quote sources or to say "not found" if no context. We can modify the prompt with "If the answer is not in the notes, say you don't recall" to be safe.
- **Performance Tuning:** See if any part is slow or resource-heavy:
    - The biggest overhead is GPT-4 calls. They are reasonably fast for short prompts, but if using many or very large context, it could slow responses to ~10+ seconds which might annoy the user. We check typical response times; if some are too slow, perhaps reduce tokens (like maybe we were giving it too much context that isn't needed).
    - Database performance: ensure indices are used (e.g., vector index working – test by enabling ANALYZE in PG or just noticing search time). If search is slow, consider reducing vector dimensions by using PCA (maybe not necessary at our scale).
    - Memory: If running all in one app, ensure not leaking or storing huge objects in session. The data volumes are small so likely fine.
- **Bug Fixing:** Any bugs encountered (UI crashes, scheduler errors, incorrect calculations) get fixed in this phase.
    - For example, maybe our VaR calc was wrong formula, or an observation got linked to wrong thesis due to an ID parsing issue – find and fix.
    - We also handle edge cases: what if a thesis has no drivers? (Make sure UI/monitor doesn't crash). What if user enters a second observation quickly (system should handle concurrency or sequential requests).
- **User Feedback & UX Improvements:** At this stage, the PM should use the system in real or simulated capacity and provide feedback:
    - Maybe they want an additional feature (like exporting a report to PDF, or a way to toggle assumptions in scenario easily).
    - We prioritize small changes that yield high value and are doable in short time.
    - For instance, if the PM says the interface is too cluttered, we might simplify or re-arrange. Or if they want a certain metric on dashboard (like Sharpe of each thesis), we add that since we already can compute it.
- **Success Criteria Validation:** Revisit the initial objectives (especially the unique ones like capturing intuition, providing insight). Evaluate qualitatively if Slice achieves them:
    - Does the PM feel the system is capturing their intuition better than a simple notebook?
    - Did the system identify a behavioral pattern or risk they hadn't noticed? (Even in testing, maybe it flagged something like "you've become more bullish as market rose" which they find true and helpful).

    - Does having the automated monitoring free the PM from manually tracking as much? Ideally yes – e.g., no need to manually watch CPI releases because the system flags it relative to their positions.
    - If some objectives are not met, decide if adjustments can be done quickly. If not (maybe a very advanced analysis is missing), note it for future improvement but consider v1 scope done if core is solid.
- **Documentation & Maintainability:** Finalize documentation for how to run and maintain the system:
    - Document how to add a new data series or ticker (in case new ETF or macro indicator needs to be included).
    - Document how to update the prompt templates or config if needed (since prompts might need tweaking as the world changes).
    - Ensure the code is organized, with clear modules: data, quant, LLM, UI separated, and add comments for complex parts.
    - This will help if the project is handed to another dev or expanded later.

Pitfalls:

- *Model updates:* If this runs long-term, OpenAI model versions may update (GPT-4 to 4.5 etc). We should pin versions or test when updates come. Possibly out of scope now but a consideration to note.
- *Overfitting to feedback:* If the PM has a certain style, we must be careful not to tailor prompts so specifically that it loses generality. But since this is a personal tool, tailoring to the user's style is actually fine.
- *Scope freeze:* At this point, avoid adding completely new features that aren't in plan (like "hey what if we add news sentiment analysis!" – not now, push to stretch goals).

Success Criteria: By end of Phase 9:

- The system runs smoothly end-to-end with no major errors.
- The PM trusts the outputs (checked for correctness).
- The PM has used it in some live capacity and found it useful – maybe anecdotally, "It reminded me of a risk I'd overlooked" or "It helps me organize my thoughts daily."
- We have baseline metrics: e.g., if multi-month cluster analysis was a goal, we have some output from it to show patterns.
- Essentially, the system is ready for "production" (in the sense of the PM relying on it regularly).

Phase 10: Stretch Goals & Future Enhancements (Month 11–12)

Purpose: In the final phase, we look beyond the MVP to **future enhancements** that were out of initial scope but desirable down the line. We allocate any remaining time to starting on these or at least designing them. These include the real broker integration, multi-user/product considerations, and advanced analytical features. Since the question explicitly asks to consider trade integration later and multi-user, we outline those here as stretch goals.

Stretch Goal A – Real Trade Integration (Schwab API):

- **ExecutionAdapter Implementation:** We already have the interface; now implement SchwabExecutionAdapter. This likely involves using Schwab's API (if available). The user said they have some integration; perhaps Schwab has an OAuth REST API or they use a third-party library. If available, we incorporate it:
    - Possibly use an SDK or just HTTP calls (ensure secure storage of API keys).
    - Implement methods: place_order(asset, quantity, action) that sends an order to Schwab (market order or specified parameters).
    - Also implement get_positions() to retrieve current actual holdings, so our system can know actual portfolio composition.
    - Test the integration with a small order (maybe on a paper account if possible).
    - We will *not* enable auto-trading – the user will always confirm or trigger. For example, in UI we could add a "Execute Trade" button on a thesis page which uses this adapter to send the order.
- **Simulated vs Real Mode:** Add configuration to switch mode. Possibly a global setting or per trade default. Initially, keep default to simulated to avoid accidental live trades. The PM can toggle "live mode" when ready.
- **Backfill of trade logs:** If the PM already has positions or trades executed outside the system, we might import those (maybe via CSV or through API get history). This could be nice to align the system's record with reality. Not urgent though.
- **Risk with Real Execution:** We have to handle errors (order rejects, etc.) gracefully and log them to user. If live trading, maybe add safety like confirmation prompts or "are you sure" dialogues.

Stretch Goal B – Multi-User & Productization Considerations:

- **Modular Architecture:** Evaluate if our architecture is modular enough to onboard a second user (likely yes, since schema is personal now, we'd need to add a user_id to relevant tables and segregate data).

    - We design a plan: each table gets a user_id, and all queries filter by it. The embedding index might be partitioned per user (or include user in vector metadata).
    - The LLM prompts might need to embed user context (like "this user's data").
    - The UI would need a login and user context to filter data.
    - Since this is beyond initial need (only one PM now), we won't implement fully, but planning it ensures our code structure (services functions etc.) can handle it.
- **Scalability and Infra:** With multi-user, we'd need a persistent web service and database (likely we deploy to a cloud VM which is fine for one, but for many maybe move to a managed DB, containerize app, etc.). We consider if our design allows that without major refactor – likely yes if we separate config.
    - Also budget: $3k/year can run a decent VM and some OpenAI usage for one user. For many, costs scale linearly with usage (OpenAI costs etc.). But as a product, one could allocate pricing accordingly.
- **OpenAI Model Switch or Fine-tuning:** For cost or independence, maybe try using GPT-3.5 for some tasks (cheaper). E.g. daily summary might be okay with 3.5 if instructions are clear. We can test quality difference. Possibly set up a cascade: try 3.5, if output not good (maybe some validation), then use 4. This could save cost.
    - Also consider fine-tuning or using embeddings to fine-tune a smaller model for the user's journal style, but that's advanced and might not be needed given small data and high quality needed.
- **Advanced Quant Stuff:** Possibly incorporate more sophisticated quant analysis:
    - Value-at-Risk with Monte Carlo or historical simulation on portfolio (beyond simple calculation).
    - Option to do scenario analysis via full revaluation (if user had options, etc., though not in current scope because focusing on ETFs).
    - Machine learning on market data for regime detection instead of threshold – e.g. train a clustering on macro variables to identify regimes historically.
    - Or use something like "blockchain of intuition" (the prompt mentions persistent mind with hash chains – not sure needed, but if concerned about tampering with past logs, one could hash them).
    - These are extras to improve robustness or depth.
- **Alternate Data Integration:** Another possible enhancement is plugging in news or sentiment data:
    - E.g. connect a news API and have the LLM summarize relevant news for each thesis's theme daily. This is something a human PM does manually (reading news), so automating it would be great. But it's a whole project on its own (requires parsing news, ensuring no hallucination). We mention as future idea but likely skip for now due to complexity and cost.
- **Continuous Learning:** Over time, the system could learn from the PM's decisions and outcomes. For example, track which theses succeeded or failed and how the PM's intuition correlated with that. Possibly use that data to calibrate future suggestions (like if the PM tends to exit too late often, the system might remind about that bias). This is very advanced (needs enough historical data and maybe ML). But as a concept, journaling data could be used to build a personal "bias model" for the PM.

    - For instance, the system could say: “Historically, when you felt ‘very confident’, your positions often ended up losing (overconfidence bias). Currently you are ‘very confident’ – be careful.” Achieving this requires data and careful analysis. We keep it as a north star for what this platform could eventually do.

Subtasks (if time allows in Month 12):

- Implement Schwab adapter basics (if we have API docs).
- Possibly allow trade execution from UI (with a confirm dialogue).
- Document how one would generalize to multiple users (maybe write an architectural note or even implement a quick multi-tenant separation if trivial).
- Maybe do one or two enhancements that are “low-hanging fruit”:
    - e.g. If user really wants a PDF report generator for monthly review, we can use a library like FPDF or just compile markdown and instruct them to print.
    - Or integrate a Slack/Telegram bot that sends alerts (there are libraries for sending messages easily).
    - These are not core requirements, but if main work finished early, a couple can be done as bonus.

Pitfalls:

- Not to break existing functionality while adding new. Will need to retest if any core changes for multi-user or live trades.
- Real trading always has risk – ensure to double-check live orders logic on something low-risk first.
- If multi-user, also consider data privacy and secure multi-tenancy (but that’s future, not needed if it remains personal).

Success Criteria: Phase 10 being stretch, success is measured by how much extra value we add without destabilizing v1:

- If trade execution is set up, a test live trade is successful.
- A clear path for multi-user is documented or prototyped.
- The PM feels confident that the system could be used as a template for others (maybe colleagues) if needed.

By end of this, we conclude the 12-month roadmap. At this point, Slice is fully functional for the PM's personal use (Phases 1–9 accomplished) and we have a vision and possibly initial steps for turning it into a broader product or adding more advanced features (Phase 10).

With all phases described in detail, we now summarize the **execution plan** with sprints and major deliverables:

Execution Plan: Sprints and Milestones (1–2 Week Increments)

To implement the above roadmap in a realistic way for a 1–3 dev team, we break it into sprints (assuming ~2 weeks per sprint) with clear focus:

- **Sprint 0 (Week 0)** – *Project setup and planning*: Finalize tool decisions, set up repo, CI, basic project scaffolding (empty database, etc.). (Outcome: architecture doc, dev environment ready).
- **Sprint 1 (Weeks 1–2)** – *Data Ingestion Pipeline*: Implement database schema for price data and macro data. Write and test backfill scripts for a couple of assets and one macro series. By end, have SPY, GLD, and CPI data in DB as a proof of concept. (Outcome: ability to query historical prices from DB, code for updating data).
- **Sprint 2 (Weeks 3–4)** – *Finish Data Coverage & Basic Risk Calc*: Ingest all priority assets and macro series. Ensure daily update job for prices works (can simulate one update). Also implement a simple risk metric function (e.g. compute vol and correlation matrix of current assets) to use later. (Outcome: DB filled with all series, ready for quant; documentation of data sources and any issues resolved).
- **Sprint 3 (Weeks 5–6)** – *Quant Engine – Backtesting & Metrics*: Integrate Backtrader and run a simple strategy with data from DB (maybe manual data load for now). Develop functions to calculate performance metrics from strategy results. No UI yet, but perhaps output to console/log. (Outcome: given a strategy definition, can produce performance stats and maybe a plot file).
- **Sprint 4 (Weeks 7–8)** – *Quant Engine – Theses Strategies & Risk Rails*: Encode 2–3 example theses as strategies and run backtests. Implement risk checks (VaR, drawdown, etc.) based on these runs. Also set up the scenario analysis function skeleton (even if just a stub that returns fixed impacts). (Outcome: quant engine library with functions like run_thesis_backtest(thesis) and get_portfolio_risk() working; initial results to show).
- **Sprint 5 (Weeks 9–10)** – *Thesis & Observation Schema Implementation*: Create DB tables for theses, observations, trades. Implement code to insert and retrieve them. Possibly build a CLI or small admin UI to input sample theses. (Outcome: can store a thesis with drivers/disconfirmers in the system and retrieve it; prepared to feed LLM).

- **Sprint 6 (Weeks 11–12)** – *LLM Integration – Observation Processing*: Connect to OpenAI API, implement process_new_observation to parse text and store observation + embedding. Test with a few sample texts. Start building similarity search. (Outcome: adding a journal entry via code yields structured output and vector in DB, can find it via semantic search).
- **Sprint 7 (Weeks 13–14)** – *LLM Integration – Thesis Q&A and Summaries*: Implement thesis critique function and multi-thesis consistency check using GPT-4. Also implement query_intuition_history for QA on notes. These can be tested via temporary console scripts or Jupyter. (Outcome: given some input data, the LLM returns meaningful questions/analysis text; refined prompts ready for UI integration).
- **Sprint 8 (Weeks 15–16)** – *Automation Framework Setup*: Integrate APScheduler or cron jobs. Schedule data updates and one or two analysis tasks (e.g. daily summary generation writing to a file or DB). Run through a simulated day or two to ensure jobs execute. (Outcome: background tasks configured; logs/printouts show they ran at intended times and did expected updates).
- **Sprint 9 (Weeks 17–18)** – *Dashboard UI – Basic Overview & Input*: Create the Streamlit app skeleton. Display list of theses, allow adding a new thesis (maybe as raw JSON or basic fields at first). Also display observations list and provide a form to add a new observation (wired to the backend function). (Outcome: developer can run streamlit run app.py and use the interface to add a thesis and note and see them appear).
- **Sprint 10 (Weeks 19–20)** – *Dashboard UI – LLM Insights & Alerts*: Extend UI to show LLM outputs: e.g. a button to "Analyze" a thesis that calls the critique function and displays questions. Show alerts (if any disconfirmers triggered from DB). Show daily summary if available. Essentially, incorporate the intelligence pieces into the UI. (Outcome: the UI now surfaces the AI's commentary and any system-generated warnings, not just raw data).
- **Sprint 11 (Weeks 21–22)** – *End-to-End Testing & Refinement*: Do a test run (maybe simulate a month of usage or walk through various scenarios). Identify bugs or awkward UI flows and fix them. Improve prompt wording or thresholds based on observed outputs. Ensure all components (data update, quant calc, LLM, UI) work in concert when something changes (e.g. if we input an observation that should trigger an alert, does it flow through?). (Outcome: system is stable, accurate, and user-tested; ready for actual daily use.)
- **Sprint 12 (Weeks 23–24)** – *Documentation & Stretch Planning*: Write final documentation (user guide, developer guide). If time remains, start on a stretch goal – likely the Schwab integration skeleton (perhaps get API keys and test a read-only call to get account holdings). Or minor feature that was deferred (like additional chart or an export function). This sprint ensures the project is nicely wrapped up. (Outcome: fully documented system; possibly initial integration code for live trading in place or at least planned.)

Throughout all sprints, we emphasize **Quant engine early, UI late** – indeed, we only started UI in Sprint 9 after getting core engines done by Sprint 7-8. The schedule is tight but feasible with focus and by reusing libraries for heavy lifting (Backtrader, OpenAI API, etc.).

Regular check-ins with the PM (user) should happen at key milestones (end of Sprint 2 for data, Sprint 4 for quant, Sprint 6 for initial LLM, Sprint 9 when UI appears) to course-correct any mismatches with expectations.

Finally, by following this roadmap and execution plan, we expect to deliver a comprehensive **Slice v1.0** in roughly 12 months, giving the PM a powerful personal macro workstation that combines data-driven rigor with structured intuition and AI assistance. The system will be extensible for future enhancements and potentially serve as a prototype for a broader product for other discretionary investors.

References:

- Polak, A. *et al.* (2025). *Battle-Tested Backtesters: Comparing VectorBT, Zipline, and Backtrader – Medium.* (VectorBT performance and limitations)
- Radovanovic, I. (2024). *Backtesting.py – Introductory Guide – IBKR/AlgoTrading101.* (Backtesting.py lacks multi-asset support)
- *Mastering the ReAct Pattern: Smarter AI Agents* (2025) – (ReAct best practices for LLM agents)
- Packman, D. (2024). *Memory for Open-Source LLMs – Pinecone Blog.* (Conversational memory via summarizing interactions into vector DB)
- **(Additional citations provided in-line throughout the design)**.