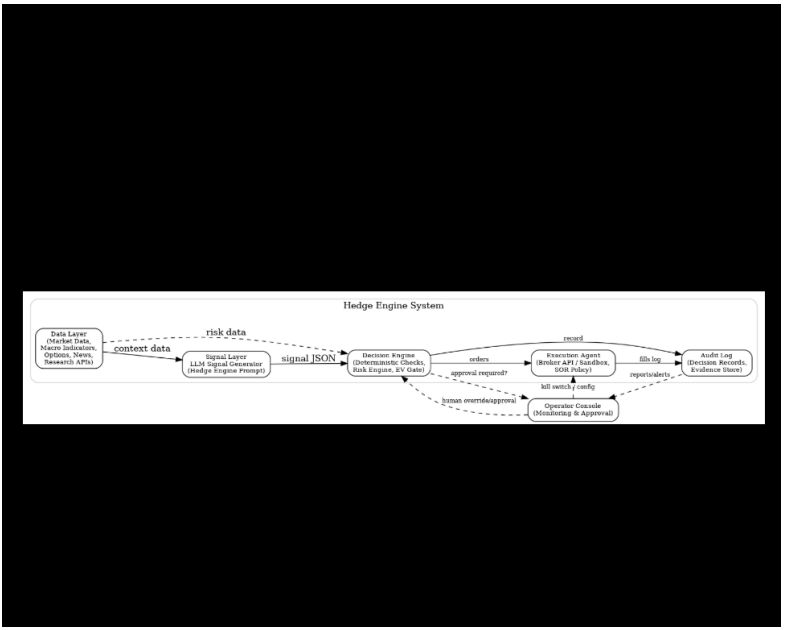
Hedge Engine Implementation Manual

Hedge Engine: Implementation Addendum
Version 1.0 – December 2025



Manifest and System Architecture Overview

Figure: High-level architecture of the Hedge Engine system, showing data inputs, the LLM-driven signal layer, decision engine with risk controls, execution agent, audit log, and operator console flows.

The Hedge Engine is a modular trading system that combines large-language-model (LLM) generated signals with rigorous, coded risk controls and auditability. This manual serves as a comprehensive implementation guide for deploying the Hedge Engine strategy. It is organized for both developers and compliance officers, detailing each system component and the exact

implementation steps to stand up a safe, auditable pilot instance of the Hedge Engine.

System Manifest: The Hedge Engine comprises the following core components:

- Data Layer: Ingests and aggregates all required data (market prices, macro indicators, options metrics, news, research documents, etc.) and provides structured snapshots as context for the strategy. This layer ensures data is up-to-date, clean, and delivered in a deterministic format for the LLM and risk modules.

- Signal Layer (LLM Engine): The large language model component that generates trading signals based on the Hedge Engine strategy blueprint. It uses a structured prompt (the e-book's methodology as a meta-prompt) plus current data to output a proposed trade decision in a JSON schema, including probabilities, suggested instruments, and cited evidence.

- Decision Engine: A deterministic logic module that validates and refines the LLM's signal. It performs quantitative checks such as expected value (EV) calculation, leveraged ETF decay simulation, and risk limit enforcement. The decision engine gates any trade recommendations behind hard rules – only if all viability criteria pass (or a human override is given) will a trade proceed. It also sets flags for human review when policy thresholds are triggered.

- Risk Management Subsystems: Part of the decision engine, the risk modules implement volatility targeting, Value-at-Risk estimation, and stop-loss management. These subsystems adjust position sizing to meet volatility targets, estimate potential losses, and impose protective measures (like scaling down exposure or halting trades on extreme events) to keep the strategy within defined risk parameters.

- Execution Agent: The component responsible for converting an approved decision into actual trade orders. In a pilot deployment this is a sandbox broker stub that simulates order execution (or connects to a paper trading API). It implements a Smart Order Routing (SOR) policy, e.g. slicing orders to limit market impact (participation rate) and using appropriate order types (limit, TWAP/VWAP). The execution agent reports fills or simulated fills back to the system.

- Audit Log & Decision Records: An immutable logging system that records every decision, data snapshot, model prompt/version, output, and action. Each Decision Record is stored as a JSON object with all

details needed to reconstruct the decision. Audit records are write-once and retained (e.g. for 7+ years) in a searchable repository. This supports compliance reviews, replay of decisions, and tracking of model behavior over time.

- Operator Console: A user interface or console for the human operator and compliance officer. It provides real-time monitoring of the engine's decisions and system status, approval workflows for any trades requiring human sign-off, and controls such as a kill switch and configuration of risk policy thresholds. The console also surfaces audit reports, evidence used by the model, and alerts (e.g. if the model's performance drifts or if any "hallucination" is detected in LLM outputs).

Each of these components will be detailed in the sections below, including architecture, implementation steps, sample code or pseudo-code, and configuration. The tone of this manual is pragmatic and implementation-focused: our goal is to enable a small quantitative team to assemble this system using open-source tools, LLM APIs, and market data, with an emphasis on safety and auditability throughout.

1. Data Layer Implementation

The Data Layer is responsible for gathering all inputs the Hedge Engine needs each cycle. It provides the real-time context for both the LLM prompt and the risk engine. A robust data layer is critical for deterministic behavior – the data snapshot must be consistent and timestamped so that any decision can later be audited or replayed with the same inputs.

Key functions of the Data Layer include:

- Market Data Ingestion: Fetching current market prices, indices, and technical indicators for relevant assets (especially the ETFs in the strategy's universe). For example, retrieving the latest price and volume for each ETF, and possibly recent returns or volatility metrics. This could be done via an API (Yahoo Finance, Alpha Vantage, etc.) or a direct data feed. The data layer should assemble a `market_snapshot` JSON containing prices, returns, and liquidity metrics for use in the prompt.

- Macro Indicators: Gathering macroeconomic data points (e.g. interest rates, inflation, employment data, yield curve values, etc.) that the strategy uses to determine the macro regime or to react to events. These might be fetched from an economics API or internal database.

They are provided as a `macro_indicators` object in context.

- Options Skew Data: If the strategy uses options market signals (for example, implied volatility or put/call open interest skew to gauge sentiment), the data layer should fetch relevant options chain metrics. For instance, using an options API to retrieve implied volatilities or risk reversal metrics for key indices. This is packaged as `options_skew` data for the prompt. In practice, one might supply aggregated values like “1M ATM IV = X%, put-call skew = Y” as context.

- News and Research Feeds: The strategy may require unstructured data like news headlines or research documents. Hedge Engine specifically mentions evidence sources such as news articles, CRS reports (Congressional Research Service documents), legislative dockets, and PTR filings (Periodic Transaction Reports of politicians). The data layer should provide connectors or access to these text sources. This could mean maintaining a local database or index of relevant documents (with IDs and text) that the LLM can cite, or calling external APIs (e.g. news feed API) for the latest headlines. For deterministic operation, local caching of any external text is recommended – e.g. store the relevant snippet or reference by ID so that the exact same text can be retrieved during an audit.

- Internal Model Store: The data layer may also interface with any internal models or stored data required by the strategy. For example, previous Decision Records (if the LLM or logic needs historical context of what the portfolio currently holds), or any learned parameters. In our implementation, we ensure that any such data used is captured in the decision record’s inputs for transparency.

Data Snapshot Structure: The output of the data layer is a deterministic snapshot that will be passed into the LLM prompt (and also logged). A recommended format is a dictionary with keys such as:

```
{

  "market_snapshot": {

    "timestamp": "2025-12-15T18:10:00Z",
```

```
  "prices": {
    "SPY": 447.25,
    "IEF": 101.10,
    "...": "..."
  },
  "volumes": { "SPY": 58900000, ... },
  "volatility": { "SPY_30d_realized": 0.12, ... }
},
"macro_indicators": {
  "GDP_growth_yoy": 0.023,
  "CPI_latest": 0.031,
  "Unemployment_rate": 0.045,
  "YieldCurve_10y2y": -0.50,
  "...": "..."
```

```
},
"options_skew": {
  "SPY_put_call_ratio": 1.4,
  "VIX": 18.2,
  "skew_index": 130.5,
  "...": "..."
},
"ptr_swarm": {
  "recent_trades": [
    {"ticker": "XLF", "net_buyers": 5, "net_sellers": 1},
    {"ticker": "XLK", "net_buyers": 2, "net_sellers": 0}
  ],
  "notable_filings": "Senator A bought SPY on 2025-11-30",
  "...": "..."
```

```
  },
  "event_calendar": {
    "next_FOMC_date": "2026-01-25",
    "econ_release_today": "NFP at 08:30 ET",
    "...": "..."
  }
}
```

This JSON (or a similar structured payload) is injected into the LLM prompt under the Context section. Note that timestamps are included wherever applicable (e.g. when data was fetched) to ensure auditability. The data layer should freeze this snapshot once collected – it becomes read-only input to the decision, and the `timestamp_utc` of the snapshot will be recorded in the Decision Record.

Implementation notes:

- Use a deterministic wrapper to fetch data. For each cycle, the data layer script should fetch all needed data in one go and assemble the context. This can be done with a Python script using APIs. Ensure to handle failures (e.g. retry or use last known data if a feed is unavailable) but log any such occurrences.
- Connector configuration: The system should be configurable via a YAML/JSON to specify API keys, endpoints, and any caching. For example, a `data_config.yaml` might list the services to call and the

instruments of interest.

- Unit tests for the data layer should verify that the data snapshot is correctly formed and contains all required fields. Also test edge cases like missing data (simulate an API giving null and see that defaults or errors are handled).

- Example: A simple implementation might use the yfinance library for market prices and pandas_datareader for macro series in a Jupyter notebook environment. For production, one might use an official feed or CSV uploads for repeatability. Always record the source of each data point if possible (e.g. attach source name in the JSON), for later reference.

The output of the data layer is then passed to the signal generation stage. Next, we describe how the Hedge Engine uses an LLM to generate a candidate trade signal from this context.

2. Signal Layer (LLM Signal Generation)

The Signal Layer is where the Large Language Model (LLM) is used to generate trading signals based on the Hedge Engine strategy blueprint. This layer takes the structured data from the Data Layer and the embedded strategy logic (the e-book content) to produce a proposed decision in machine-readable format.

2.1 Prompt Design: We deploy a structured prompt to guide the LLM. The entire Hedge Engine e-book (or a distilled set of rules from it) serves as context, effectively functioning as the “program” the LLM will execute. The LLM is instructed to follow the strategy’s rules precisely and to output a JSON object with specified schema. Key aspects of the prompt design include:

- System Message: For example:

  “You are an expert macro portfolio AI following the Hedge Engine strategy described in the provided document. Your task is to analyze the current data and propose a portfolio action strictly according to the strategy rules. You MUST output a JSON following the specified schema and include evidence for your conclusions.”

  This system prompt encapsulates the role of the LLM and sets the expectation for output format and behavior (no digressions, no

omissions of evidence, etc.).

- Context Injection: We include the data snapshot from the Data Layer in the prompt (perhaps after the system message). The prompt might say: “Context: { *JSON of market_snapshot, macro_indicators, etc.* }”. By providing this context JSON, the LLM has access to all the current state it needs to evaluate signals (e.g., the business cycle indicators, recent market moves, any political trades, etc.).

Task Instruction: We use a user message to explicitly request the signal generation. For example:

```
[USER] SIGNAL_GENERATION
```

Using the context data provided, determine the current recommended portfolio adjustment per the Hedge Engine methodology. Return your analysis and decision in JSON format according to the schema. Cite at least 2 distinct evidence sources if your p_success ≥ 0.7.

- The special token `SIGNAL_GENERATION` acts as a clear indicator in the conversation, and the instructions emphasize JSON output and evidence requirements.

Few-Shot Examples: To help the model adhere to format, we include a few examples of inputs and outputs in the prompt. For instance, one example of a bullish early-cycle scenario, one of a Fed event hedge, and one showing a low-confidence output that triggers human review. These examples (kept concise) are given in JSON form as if they were previous outputs, to demonstrate the correct schema usage. The prompt might have a section like:

```
"Example1": {

  "p_success": 0.9,

  "p_confidence": 0.95,
```

"horizon_days": 5,

"expected_delta": {"fav": 0.02, "neutral": 0.0, "unfav": -0.01},

"suggested_instrument": {"type": "ETF", "tactic": "core", "symbol": "XLK"},

"evidence": [{ "source_id": "CRS2025-01", "type": "CRS", ... }],

"rationale": "Example early-cycle tilt with tech overweight",

"flags": {"requires_human_review": false, "low_evidence": false, "out_of_universe": false}

}

- (The actual examples would be formatted as part of the prompt, not as literal JSON on output, but to guide the model's style.)
- Output Schema Enforcement: The model is instructed to output strictly in JSON according to the schema. We will validate the LLM's output and reject any response that fails to parse or match the schema. This is critical: no free-form text should be accepted from the model, to avoid hallucinations or ambiguous instructions from being acted on. The schema is given to the model in the prompt (either described in text or via a `system` function message if using OpenAI function calling). The schema the LLM follows is a simplified version of the Decision Record fields relevant to the model's output:

2.2 Signal JSON Schema: The expected JSON structure from the LLM includes fields like:

{

```
"p_success": 0.0,            // Probability of favorable outcome (LLM's estimate)

"p_confidence": 0.0,         // Confidence level in its assessment

"horizon_days": 0,           // Suggested holding period or horizon for the trade

"expected_delta": {          // Expected move magnitudes in scenarios

    "fav": 0.0,              // Favorable scenario return (fraction or %)

    "neutral": 0.0,          // Neutral scenario return

    "unfav": 0.0             // Unfavorable scenario return

},

"suggested_instrument": {

    "type": "ETF",           // Could be "ETF", "LETF", "Futures", etc.

    "tactic": "core",        // "core" or "tactical" position

    "symbol": "SPY"          // Specific instrument ticker or identifier

},

"rationale": "string",       // Brief explanation of the decision (optional short bullets)
```

```
  "evidence": [
    {
      "source_id": "string",   // Identifier of source (e.g., doc ID or URL)
      "type": "CRS|docket|news|options_skew|ptr",
      "filecite": "ref_id|L10-L20", // Reference (like filename and line range, if applicable)
      "excerpt": "string"      // Short excerpt supporting the claim
    }
  ],
  "flags": {
    "requires_human_review": false,
    "low_evidence": false,
    "out_of_universe": false
  }
```

}

The above is an illustrative schema. In practice, the actual JSON returned by the model will be embedded in a larger Decision Record (which the Decision Engine constructs), but this portion is the LLM's responsibility.

Field semantics:

- `p_success` – the model's estimated probability that the suggested trade or scenario will succeed (e.g. yield a positive return or achieve its hedge objective). This is a key quantitative output from the model that will be used in EV calculations and calibration checks.

- `p_confidence` – a measure of the model's confidence in its recommendation. This might reflect the consistency or strength of evidence. The model is prompted to provide a lower confidence (closer to 0.5 or 0.6) if evidence is weak or conflicting, and high confidence (0.9+) if evidence is overwhelming. This field is used to trigger oversight: e.g. if `p_confidence` is below 0.7, we'll require human approval.

- `horizon_days` – the number of days the model expects the trade to be held or the prediction to play out. For example, a tactical hedge around an event might have horizon of 2 days, whereas a macro regime shift position might be 30 days.

- `expected_delta.fav/neutral/unfav` – the expected outcome (perhaps in percentage return) under three scenarios: favorable, neutral, unfavorable. The LLM will derive these from its analysis (e.g., "if the Fed cuts rates (fav scenario), S&P500 +5%, if neutral (no surprise) +0%, if unfavorable (hike) -3%"). This structured breakdown is useful for calculating expected value and risk.

- `suggested_instrument` – the instrument the LLM proposes to use. It has a type (e.g., a plain ETF vs a Leveraged ETF or futures contract) and a tactic classification (core = strategic long-term position, tactical = short-term hedge or overlay). The model should also specify the actual symbol or contract. The allowed universe of symbols will be controlled

(and anything outside triggers a flag).

- `rationale` – a short explanation of the reasoning (for human readability). The LLM might produce a few bullet points here, but this is primarily for transparency; the execution logic will not parse this field.

- `evidence` – an array of evidence items. For every probabilistic claim or important assumption, the LLM must provide evidence. Each evidence item includes a source type and an excerpt or reference. For example, if the model says "p_success = 0.8 because a Congressional research report and options skew suggest high odds of outcome", it should cite the specific report paragraph (type: CRS) and the options data (type: options_skew). At least two distinct sources are required for high-conviction trades (p_success ≥ 0.7). If the model output lacks sufficient evidence entries or they are not credible, it will be flagged (and likely a second step of evidence augmentation occurs).

- `flags` – the model can pre-flag certain conditions:

    - `requires_human_review`: set to true if the LLM itself is unsure or thinks a human should double-check (this might be triggered by internal logic if, say, data is very uncertain).

    - `low_evidence`: true if the model knows it didn't find strong evidence for its claims.

    - `out_of_universe`: true if the suggested instrument is outside the allowed list (the model can be made aware of the universe; if it still recommends something else, it raises this flag).

The LLM is instructed to set these flags as needed (and the Decision Engine will also enforce setting them in certain cases, see Safety Gates below).

2.3 Latent Research and Evidence Augmentation: One innovative aspect is the two-step prompt process to ensure evidence completeness. If the initial `SIGNAL_GENERATION` response from the LLM has high conviction but not enough evidence citations, we trigger an Evidence Augmentation step. This is done by issuing another user prompt, e.g.:

[USER] EVIDENCE_AUGMENT

Provide 3 source ids that support your top claim, with a one-sentence excerpt from each.

As defined in the strategy blueprint, this prompt tells the LLM to output additional evidence items. The LLM might then return a list of source identifiers and quotes (for instance, it might pull in an ID of a news article that backs its thesis, a line from the Fed minutes, etc.). These can then be merged into the `evidence` field of the Decision Record. The augmentation step can use a truncated context focusing on the model's primary claim (e.g., if the model said "80% chance of rate cut – evidence: …", we ask it to find more sources supporting the rate cut expectation).

Note: The design expects the LLM to have access to the knowledge base of sources (via its training data or via retrieval augmentation if implemented). If we have a vector database of documents, one could intercept the EVIDENCE_AUGMENT request and do a similarity search to provide the model with candidate texts to choose from. For the pilot, a simpler approach is to rely on the LLM's internal knowledge for widely known info, and maintain a curated set of documents that can be provided in prompt if needed (e.g. supply the text of a CRS report or Fed minutes as context if expecting the model to cite them).

2.4 Model Selection and Configuration: We will typically use a GPT-4 level model or similar for this signal generation, given the complexity of tasks (reading macro data, recalling strategy rules, and writing JSON). The model should be configured with:

- Temperature = 0 (or very low, like 0.1) to ensure deterministic output and avoid random variation. Consistency is crucial for auditability.
- Max tokens high enough to include all context (the strategy blueprint + data) and the output.
- Function calling or structured output preferences: If using OpenAI API, define a function schema for the output JSON to strictly enforce structure. Otherwise, rely on prompt instructions and validate afterwards.

2.5 Handling LLM Output: Once the LLM produces a candidate decision JSON, the Signal Layer passes this to the Decision Engine. But first, we perform validation: parse the JSON, ensure all required fields are present, and that values are within expected types/ranges. If parsing fails (e.g., the LLM returned text or incomplete JSON), the runtime should log the failure and either retry (perhaps with a stricter prompt) or fail-safe (no trade). As policy, *any response that fails schema is rejected*. In a controlled environment, we might iterate with the LLM until it gets it right, but in production trading, a safer route is to default to “no action” if the AI’s output is non-compliant.

2.6 Example LLM Output: Below is a hypothetical (simplified) example of the LLM’s JSON output for an illustrative scenario:


```
{
  "p_success": 0.78,
  "p_confidence": 0.80,
  "horizon_days": 5,
  "expected_delta": {
    "fav": 0.025,
    "neutral": 0.0,
    "unfav": -0.015
  },
```

```
  "suggested_instrument": {

    "type": "LETF",

    "tactic": "tactical",

    "symbol": "SPXU"

  },

  "evidence": [

    {

      "source_id": "CRS2024-88",

      "type": "CRS",

      "filecite": "CRS_FedOutlook.pdf|L10-L18",

      "excerpt": "The CRS report indicates increasing recession odds in the next
quarter..."

    },

    {
```

```
      "source_id": "NewsFT_2025-12-14",

      "type": "news",

      "filecite": "FT:2025-12-14",

      "excerpt": "Financial Times: 'Markets brace for Fed policy reversal as data softens...'"

    }

  ],

  "flags": {

    "requires_human_review": false,

    "low_evidence": false,

    "out_of_universe": false

  },

  "rationale": "Macro indicators suggest late-cycle; hedging via SPXU (3x inverse S&P) for 1-week horizon."

}
```

This indicates the model thinks there's a 78% chance the trade (short S&P via a 3x leveraged ETF) will be profitable over ~5 days, with a favorable scenario of +2.5% gain, unfavorable -1.5% loss. It cites a CRS report and a news article supporting the decision. Because `p_confidence` is 0.80 (in the 0.7–0.85 moderate range), this likely would trigger a manual review per policy (discussed later).

With the signal JSON in hand, the process moves to the Decision Engine for deterministic checks.

3. Decision Engine and Pre-Trade Checks

The Decision Engine is the rule-based core that takes the LLM's suggested signal and decides whether to execute it, modify it, or block it. It implements pre-trade gates that ensure only trades meeting defined criteria (expected value positive, risk within limits, compliance constraints satisfied) can go through without manual intervention. This component provides the "glass box" determinism to complement the LLM's "black box" suggestions.

Key responsibilities of the Decision Engine:

- Expected Value (EV) Calculation: Compute the expected net benefit of the proposed trade using the probabilities and outcomes given by the LLM, adjusted for costs like LETF decay or transaction costs.

- LETF Decay Simulation: If the suggested instrument is a Leveraged ETF (LETF) and the horizon is more than one day, simulate or estimate the potential decay due to daily rebalancing and volatility, and factor this into the EV.

- Viability Gate: Determine if the trade's expected value after adjustments is above a minimum threshold (and not negative). If not, mark the trade as not viable (and thus it will not execute unless manually overridden).

- Risk Checks: Apply risk management rules, such as ensuring the new trade will not violate volatility targets, VaR limits, position limits, or stop-loss constraints. This includes adjusting position size if needed (volatility targeting) and flagging any scenario that exceeds risk tolerance.

- Policy & Compliance Gates: Check if any conditions require human approval: e.g., low model confidence, use of certain instruments, or large notional trades beyond preset limits. Set flags for human review accordingly.

- Decision Record Assembly: Compile all information (inputs, LLM output, results of checks, execution plan) into the structured Decision Record JSON for logging.

We break down these functions below.

3.1 Expected Value (EV) Calculation

Gross Expected Value (EV_gross): Given the model's probabilities and scenario outcomes, the decision engine calculates the expected return of the proposed trade. For example, using the fields `p_success` and the `expected_delta` values:

- We interpret `p_success` as the probability of the favorable scenario (`fav`) occurring.

- We might assume a certain probability for the unfavorable scenario; if only two outcomes (fav or unfav) are considered, `p_unfav = 1 - p_success` (assuming neutral is just no significant change). If a neutral scenario is explicitly considered, we may have to assign probabilities to each scenario. A simple approach: if the model didn't provide explicit probabilities for neutral, assume neutral is the complement of the other two combined probability. However, often `neutral` might be an ~0 outcome that can be grouped with either.

- For safety, we can treat `neutral` outcome as occurring if the event doesn't strongly go in either direction. One way: assume `p_neutral` is small unless specified. To keep it straightforward: EV = p_success * fav_delta + (1 - p_success) * unfav_delta (ignoring neutral as 0 delta).

For example, if `p_success = 0.78`, fav = +2.5%, unfav = -1.5%:

EV_gross = 0.78*(+2.5%) + 0.22*(-1.5%) = 1.95% - 0.33% = +1.62%

So about +1.62% expected gain over the horizon. This would be recorded as `ev_gross` in the record.

Costs and adjustments: If there are known transaction costs or fees, subtract them. For instance, if trading ETFs, the bid-ask spread/slippage might be negligible for small trades, but if using options or futures, incorporate an estimate of entry/exit cost. In a pilot, these can be approximated or set to zero to focus on strategy logic.

3.2 Leveraged ETF Decay Simulator

If the recommended instrument is a Leveraged ETF (LETF) and the `horizon_days` is more than one day, the engine must estimate the impact of volatility decay on the instrument's performance. Leveraged ETFs reset daily, which causes their returns over multiple days to deviate from the simple multiple of the index's return due to path dependency (volatility drag). We implement a deterministic decay simulation to quantify this and adjust the expected value accordingly.

Approach: We can use a mathematical approximation or a Monte Carlo simulation with a fixed random seed (to remain deterministic). A widely used formula for expected return of a daily-reset leveraged fund is:

rx≈(1+μ)x×exp((x−x2)σ2T2)−1,r_x \approx (1 + \mu)^x \times \exp\Big(\frac{(x - x^2)\sigma^2 T}{2}\Big) - 1,rx≈(1+μ)x×exp(2(x−x2)σ2T)−1,

where xxx is the leverage factor (e.g. 3 for a 3x ETF, or -1 for an inverse 1x), μ\muμ is the underlying index's expected return over period TTT, and σ2\sigma^2σ2 is the variance of the underlyingquant.stackexchange.comquant.stackexchange.com. The term (x−x2)σ22\frac{(x-x^2)\sigma^2}{2}2(x−x2)σ2 effectively captures the drag: for x=3x=3x=3, x−x2=3−9=−6x-x^2 = 3-9 = -6x−x2=3−9=−6 (negative, so it subtracts from return). For an inverse x=−1x=-1x=−1, x−x2=−1−1=−2x - x^2 = -1 - 1 = -2x−x2=−1−1=−2. Higher volatility (σ\sigmaσ) or longer horizon TTT increases the decay.

However, to keep things intuitive, we implement a small Monte Carlo with fixed seed:

- Determine volatility and drift: Use recent historical volatility of the underlying index (or an implied vol if available) as σ\sigmaσ. Use an expected drift μ\muμ (which could be 0 for short-term or derived from yield).

- Simulate paths: Simulate a large number (e.g. 10,000) of daily return paths for the underlying over `horizon_days`. For each path, calculate the cumulative return of the LETF vs the underlying.

- Measure decay: The "decay" can be defined as the difference between the LETF's actual average outcome and the ideal outcome if it scaled linearly. For instance, if underlying is flat over period (0% change) but a 3x ETF ends on average at -5%, that 5% is purely decay.

- Worst-case scenario: Also consider a scenario of oscillating returns which maximize decay (like up-down-up-down swings). One extreme test: if the underlying moves +a, -a, +a, -a ... alternating, the ETF will lose value while underlying ends near unchanged. We can simulate with a fixed pattern for an upper bound on decay.

Deterministic Implementation: For speed, an analytical approach is fine: We can estimate decay as approximately decay≈x(x−1)σ2T2\text{decay} \approx \frac{x(x-1)\sigma^2 T}{2}decay≈2x(x−1)σ2T (taking the negative of the exponent part of formula above, for x>1 or x<0)[quant.stackexchange.com](). For example, for x=3, decay≈3∗2∗σ2T2=3σ2T\text{decay} \approx \frac{3*2*\sigma^2 T}{2} = 3\sigma^2 Tdecay≈23∗2∗σ2T=3σ2T. If σ=20%\sigma = 20\%σ=20% (0.2) and T=5 days≈0.02 yearsT = 5 \text{ days} \approx 0.02 \text{ years}T=5 days≈0.02 years, decay ≈ 3∗0.04∗0.02=0.0024=0.243 * 0.04 * 0.02 = 0.0024 = 0.24%3∗0.04∗0.02=0.0024=0.24. That seems small for 5 days at 20% vol. For larger T or vol, it grows. We will implement a more precise calc.

Code Example – LETF Decay (illustrative Python-like pseudocode):

```
import math

def estimate_letf_decay(leverage, underlying_vol, horizon_days):
```

```
    """
    Estimate the expected decay (fractional) for a leveraged ETF over the given horizon.

    Using continuous approximation from Cheng & Madhavan (2009).

    """

    T = horizon_days / 252  # convert days to years (assuming 252 trading days/year)

    x = leverage

    sigma = underlying_vol

    # Expected factor relative to underlying (without drift)

    factor = math.exp((x - x**2) * (sigma**2) * T / 2.0)

    # If x positive, factor < 1 indicates decay; if x is negative (inverse), also typically < 1.

    decay_fraction = 1 - factor

    return max(0.0, decay_fraction)
```

Example: 3x ETF, 5-day horizon, 20% vol

print(estimate_letf_decay(3, 0.20, 5)) # e.g., returns ~0.0024 (0.24%)

In practice, for each decision:

- Identify if `suggested_instrument.type == "LETF"`.
- Determine `leverage_factor` (we might have a map of symbols to leverage, e.g. SPXU is -3, meaning 3x inverse).
- Get an estimate of `underlying_vol` for the horizon. Perhaps use the volatility of the underlying index from `options_skew` or recent history. If unknown, conservatively assume a high vol (to not underestimate decay).
- Compute decay = `decay_fraction * |expected_index_return|` or simply use decay_fraction of the underlying's magnitude. Actually, a simpler integration is: we adjust the expected favorable/unfavorable outcomes. For a leveraged bull ETF x=2: If underlying expected +5% in fav, -5% in unfav, the ETF ideally +10%/-10%, but with decay it will be slightly less. We can multiply the fav/unfav magnitudes by `factor` or subtract a bit from EV.

For simplicity, we can calculate an expected decay loss in percentage points and subtract it from the EV. For example, if EV_gross was +1.62% and we estimate 0.3% decay over 5 days for the chosen LETF, then `ev_net = ev_gross - 0.3% = +1.32%`. This `ev_net` goes into the decision record.

Edge cases: If horizon is very long (weeks), or underlying_vol is extreme, the decay could wipe out gains or even make expected return negative. Our policy might define a maximum horizon for using LETFs safely (e.g., T_max = 7 days). Indeed, the blueprint suggests if `horizon_days > T_max` and the instrument is LETF, that should trigger a low_evidence or human approval flag. We will enforce: if `horizon_days` beyond a week and a LETF is suggested, set

`flags.low_evidence = true` or require an override because model is going out of its intended use.

We also set the `letf_decay` field in the record as the numeric value of decay adjustment (for audit transparency).

3.3 Viability and Pre-Trade Gate

After computing `ev_net` (net expected value after decay and costs), the Decision Engine decides if the trade is viable to execute:

- Define a threshold `safety_margin` (could be 0 or a small positive number like 0.5% expected return) as minimum EV to bother trading. In the JSON, `viability_pass` is a boolean.
- If `ev_net` is positive and meets any additional criteria (like probability not entirely borderline), set `viability_pass = true`. Otherwise, `false`.
- The blueprint mandates: *"Execution is forbidden unless viability_pass is true or a documented human override occurs."*. So this gate absolutely must stop trades with negative or negligible expected value.

For example, if `ev_net` came out to -0.5%, clearly viability_pass = false -> the trade should be blocked (or at least not auto-executed).

We incorporate viability into the overall logic as:

```
if ev_net > 0 and some_minimum_condition:
    decision_record["quant_checks"]["viability_pass"] = True
else:
    decision_record["quant_checks"]["viability_pass"] = False
```

(where `quant_checks` in the record contains ev_gross, letf_decay, ev_net, viability_pass).

Additionally, we implement any custom logic to refine viability. For instance:

- If `p_confidence` is extremely low (<0.5), even a positive EV might be too unreliable; we could mark not viable and require human input.
- If evidence count is below required (e.g., less than 2 sources for high p_success), one could mark not viable until evidence is augmented (though more appropriately we use a flag to get human review in that case, as evidence can be subjective).

But generally, EV and decay are numeric, so viability_pass is mostly a numeric gate.

3.4 Risk Engine: Volatility Targeting and Position Sizing

Before finalizing the decision, we ensure the proposed trade aligns with portfolio risk targets. The Hedge Engine strategy uses volatility targeting to scale exposure. Implementation steps:

- Compute current portfolio volatility: Based on recent returns (e.g., last 20 days) of the current portfolio, annualized (this could be in `recent_vol`).
- Compare to `target_vol` (for example, 10% annual).
- Compute `scale_factor = target_vol / recent_vol`.
- Clamp the `scale_factor` between some bounds (e.g., 0.5 and 2.0) to avoid over-leveraging or dropping exposure too low.
- If `scale_factor` is significantly different from 1, adjust the size of the new trade accordingly. For example, if the model suggested buying $100k of SPY but recent_vol is half of target (so scale_factor = 2), you might double the trade size (subject to leverage limits and instrument availability like using a 2x ETF or margin). Conversely, if scale_factor is

<1, you reduce the trade size or partially hedge with cash.

In practice, since the LLM's suggestion might already incorporate some vol logic (it read the blueprint), we treat our implementation as a safety net. We can post-adjust the order quantities in the Execution Plan to meet the volatility target. For example:

```
proposed_qty = base_qty * scale_factor
```

proposed_qty = min(proposed_qty, max_position_limit) # also enforce absolute limits

-
- The adjustment (if any) should be recorded (perhaps in the rationale or as a note in the decision record).

Volatility targeting ensures the portfolio's overall risk is steady. If the model suggests a major position increase during a high-vol period, this mechanism will scale it down.

Value-at-Risk (VaR) Check: As an additional risk measure, compute the portfolio's VaR if this trade is added:

- We can use a simple historical VaR: take historical daily returns of similar positions or the index, scale to the position size, and find the 95th percentile loss. Or use a parametric VaR: VaR95≈1.65σportfolio×1 day\text{VaR}_{95} \approx 1.65 \sigma_{\text{portfolio}} \times \sqrt{1 \text{ day}}VaR95≈1.65σportfolio×1 day value.

- Check that this VaR is within limits (e.g., does not exceed X% of portfolio NAV).

- If adding the trade pushes VaR too high, flag for review or trim the position size.

Stop-Loss / Drawdown Management: If the strategy design includes stop-loss rules (e.g., "if trade loses 3% from entry, exit early"), we incorporate that as part of

the execution rules rather than immediate decision. However, the Decision Engine can pre-plan for it by:

- Setting a stop-loss level for the trade (e.g., calculate price that corresponds to -3% on the position).
- Storing this in the execution plan or internal state so that if market hits that level, the Execution Agent or next cycle can act to exit.

We might not execute the stop-loss at decision time, but it's part of risk management configuration.

Leverage and Concentration Limits: The engine should also ensure:

- Gross leverage: If the portfolio uses margin or LETFs, total effective exposure vs NAV is monitored. The policy might limit gross leverage to, say, 1.5x NAV. If adding this trade (especially with LETF) would exceed that, auto-approval should be disabled and require human sign-off.
- Single-position limit: e.g., no single trade > 5% of NAV (just an example). If the model tried to allocate too much to one ETF, scale it down or mark for approval.
- Instrument eligibility: Check that `suggested_instrument.symbol` is in the allowed list (the “Definitive ETF Universe”). If not, flag `out_of_universe=true` and set `requires_human_review=true`. In our compliance config, we will maintain a list of approved tickers the system can trade.

After these risk adjustments and checks, we have either a modified trade proposal that is safe or we determine that the trade should not auto-execute.

3.5 Putting It Together – Pseudocode

Below is a simplified pseudocode of the Decision Engine logic incorporating the above elements:

```
decision = llm_output  # JSON from LLM, already parsed
```

```
record = {}  # initialize Decision Record structure



# 1. Populate LLM output fields in record

record["llm_output"] = decision  # copy under llm_output key

record["inputs"] = data_snapshot  # attach input snapshot used



# 2. Compute EV and apply decay

p = decision["p_success"]

fav = decision["expected_delta"]["fav"]

unfav = decision["expected_delta"]["unfav"]

ev_gross = p * fav + (1 - p) * unfav

decay_adj = 0.0

if decision["suggested_instrument"]["type"] == "LETF" and
decision["horizon_days"] > 1:
```

```
    leverage = infer_leverage_factor(decision["suggested_instrument"]["symbol"])

    underlying_vol =
data_snapshot["market_snapshot"]["volatility"].get(base_index_for(symbol),
default_vol)

    decay_frac = estimate_letf_decay(leverage, underlying_vol,
decision["horizon_days"])

    decay_adj = decay_frac * abs(fav if fav < 0 else unfav)  # assume worst-case
decay proportional to move

    ev_net = ev_gross - decay_adj

else:

    ev_net = ev_gross



record["quant_checks"] = {

    "ev_gross": round(ev_gross, 5),

    "letf_decay": round(decay_adj, 5),

    "ev_net": round(ev_net, 5),

    "viability_pass": ev_net > safety_margin
```

```
}



# 3. Basic viability gate

if not record["quant_checks"]["viability_pass"]:

    # Block auto-execution

    record["flags"] = {"blocked": True}

    # (The trade will not proceed unless human override)



# 4. Risk adjustments

scale_factor = compute_vol_scale_factor(current_portfolio, target_vol=0.1)

if scale_factor != 1.0:

    adjust_execution_plan_for_volatility(scale_factor,
proposed_trade=decision["suggested_instrument"])

# Also compute VaR and check limits
```

```
var_est = estimate_portfolio_var(current_portfolio, proposed_trade=decision)

if var_est > VAR_LIMIT or would_exceed_gross_leverage(proposed_trade):

    require_human = True




# 5. Policy checks for human approval

require_human = False

policy = current_policy_profile  # thresholds set by operator

if decision["p_confidence"] < policy.confidence_min or decision["p_confidence"] <
0.7:

    require_human = True

if decision["p_confidence"] < 0.7 and ev_net > safety_margin:

    # scenario: low confidence but seemingly high EV

    # Could either block or require human review, we choose human review

    require_human = True
```

```
if decision["p_confidence"] >= 0.7 and decision["p_confidence"] < 0.85:

    # moderate confidence range

    require_human = True

if decision["suggested_instrument"]["type"] == "LETF" and
decision["horizon_days"] > 7:

    require_human = True  # over decay horizon

if is_leverage_above_threshold(decision, portfolio_NAV):

    require_human = True




record["human_review"] = {

    "required": require_human,

    "reviewer_id": None,

    "approval": None,

    "notes": ""
```

}

The above pseudocode encapsulates the logic:

- It calculates `ev_net` and sets viability.
- It adjusts for volatility targeting and checks limits.
- It sets `require_human` flags based on policy. (We will detail policy profiles in the next section.)

One can see references to specific policy values (like confidence thresholds and horizon days); those come from the Operator's configuration.

Now, assuming the decision passes viability (ev_net positive) and perhaps does not strictly require a block, the Decision Engine prepares an Execution Plan if auto-execution is allowed. Otherwise, it will halt and mark for human approval.

4. Execution Agent and Order Management

The Execution Agent takes a validated decision and turns it into actionable trade orders, either in a sandbox simulation or live broker API. For a pilot deployment, we implement this as a sandbox broker stub – meaning no real money moves, but we simulate the process closely to test end-to-end.

4.1 Execution Plan: In the Decision Record JSON, there is an `execution_plan` section which includes the orders to be executed and any special instructions (like SOR policy). For example:

"execution_plan": {

"orders": [

{

```
    "instrument": "SPY",
    "side": "BUY",
    "qty": 123,
    "type": "LIMIT",
    "limit_price": 447.00,
    "timestamp_planned": "2025-12-15T18:30:00Z"
   }
 ],
 "sor_policy": "percent_of_adv=10"
}
```


This indicates a plan to buy 123 shares of SPY via a limit order at $447, scheduled around 18:30Z, with a Smart Order Routing policy to not exceed 10% of average daily volume in execution.

4.2 Smart Order Routing (SOR) Policy: We implement a simple SOR logic appropriate for a small trader:

- Participation Rate: We configure a maximum percentage of the asset's average daily volume (ADV) that our orders can be. The blueprint

suggests 1% to 10% as a configurable range by liquidity. For liquid ETFs like SPY, we might cap at 5% ADV; for less liquid, maybe 1%. This prevents our orders from being too large relative to market.

- Order Type and Timing: For small orders, a single limit or market order might suffice. For larger orders, use algorithmic execution like TWAP (time-weighted average price) or VWAP (volume-weighted) to spread the trade. In practice, our stub can simulate a TWAP by splitting into multiple child orders over a timeframe.

- Slippage guard: We include a check on slippage – for example, if the price moves more than X basis points away from our intended price while executing, we abort or alert. The blueprint suggests aborting if estimated slippage > threshold.

- Preferred Venues: In a real SOR, one would route to multiple exchanges or dark pools. In our simulation, we can assume we always fill at the best market price available.

4.3 Sandbox Broker Stub: We create a class or module representing a broker interface. Its API might have methods like `send_order(order)` and `check_fills()` but instead of sending to an exchange, it will:

- Log the order (maybe to a in-memory list or file).

- Immediately (or with a configurable delay) simulate a fill. For simplicity, we could fill at the current market price from the data layer (or the next tick in a backtest scenario). If `type` is LIMIT, ensure the market has traded that price; if `type` is MARKET, just take the next available price.

- Apply some randomness or slippage if we want realism: e.g., fill at 0.1% worse than last price to simulate impact.

- Mark the order as filled and return a fill confirmation.

All of this occurs in sandbox mode, so no actual brokerage calls. If we wanted to connect to a live paper trading API (like Interactive Brokers paper, or Alpaca's paper trading), we could integrate that by swapping out this stub with actual API calls.

4.4 Execution Modes: The system should have modes:

- Shadow/Sandbox Mode: All orders are sent to the stub, and no live trades occur. This is the default for testing and model updates.

- Live Mode: Orders are sent to a real broker API. In live mode, the kill switch and human approvals become critical safety checks.

- Canary Mode: A special live mode where only very small notional trades are executed to test a new strategy update. For example, if normally one would trade $50k positions, in canary mode we trade $5k or less, while closely monitoring. Our execution agent can implement this by scaling down all order quantities by a “canary factor” when this mode is active.

4.5 Order Generation: Based on the decision, the execution plan is created by the Decision Engine just before handing to Execution Agent. For instance, if the model said “Buy SPXU (3x inverse S&P) as a tactical hedge”:

- Determine quantity: Suppose portfolio NAV is $1,000,000 and they want a 5% allocation to the hedge. That’s $50,000. If SPXU is $15 per share, that’s ~3333 shares. Apply vol target scaling if needed. Also, ensure not exceeding any position limits (say we cap any single trade at $50k in this conservative profile).

- Choose order type: If market is liquid and it’s near trading time, perhaps a market order for simplicity. But per policy, prefer limit orders or TWAP for large trades. We might break 3333 shares into 3 chunks of 1111 each to trade over a few minutes.

- Set `sor_policy`: e.g. `"sor_policy": "participation_rate=5%"` or `"TWAP_30min"`.

- Populate the `orders` list in the plan accordingly.

4.6 Execution Confirmation and Logging: After sending orders via the stub, the Execution Agent collects the results:

- If filled, great – capture fill price and time.

- If partially filled or not filled (maybe price moved away for a limit), the agent can decide to adjust or cancel after some time. This logic can be made simple: in sandbox, we can assume fill unless price truly never hit

(which in a backtest we know from data).

- All these outcomes should be appended to the Decision Record's audit log. For example, we might extend `execution_plan.orders` entries with a `"status": "filled"` and `"fill_price": X`.

4.7 Example Execution (Sandbox): Let's say the plan was to buy 3333 shares of SPXU at market:

- Our stub sees market price from data (e.g., $15.00) and "fills" the order at $15.00 for all shares.
- It returns a fill confirmation object.

We log:

```
"execution_plan": {
 "orders": [
  {
    "instrument": "SPXU",
    "side": "BUY",
    "qty": 3333,
    "type": "MKT",
    "timestamp_planned": "2025-12-15T18:31:00Z",
```

```
    "fill_timestamp": "2025-12-15T18:31:00Z",
    "fill_price": 15.00,
    "status": "filled"
   }
 ],
 "sor_policy": "participation_rate=5%"
}
```


-
- The Audit Log module will note this and also keep a higher-level log of trade executed.

This simplicity is acceptable for pilot. In a more advanced setup, we would integrate something like Interactive Brokers API with an `IBKRExecutionAgent` implementing similar interface.

4.8 Safety Checks in Execution: The Execution Agent must also check the Global Kill Switch state before sending any order. If the kill switch is engaged, it should not send orders (or if in hardware form, it might physically prevent it). The kill switch is controlled by the Operator Console (more in section 6). Implementation: the agent simply queries a flag (perhaps an environment variable or a file that the operator toggles) each cycle. If `KILL_SWITCH=true`, then skip order sending and log “Skipped due to kill switch”.

Additionally, in live trading, the Execution Agent might need to handle errors (broker unavailable, network issues). In sandbox, we simulate success, but we should still code for these possibilities and ensure an error doesn’t result in an

uncontrolled state (e.g., if an order fails to send, we either retry or mark the decision as not executed and alert operator).

At this point, if a trade was executed (even in sim), the system will have taken on a position. We should ideally update the portfolio state (positions holdings) in our internal memory so that next cycle's decisions account for current holdings.

Finally, every decision (whether executed or not) gets formally logged via the Audit system.

5. Audit and Decision Record System

Auditability is a cornerstone of Hedge Engine. Every cycle, whether a trade is made or not, a Decision Record is created capturing exactly what happened and why. This record is the atomic artifact for compliance review. In implementation terms, it's a JSON document written to persistent storage (and optionally, to an audit database or ledger for immutability).

5.1 Decision Record Schema: The structure of the Decision Record JSON was touched on earlier. To summarize the fields (some of which we have filled in previous steps):

```
{

  "decision_id": "UUID",              # unique ID for this decision instance

  "timestamp_utc": "2025-12-15T18:31:00Z",  # time of decision

  "model_version": "gpt-4-0613",      # identifier of LLM model used

  "prompt_hash": "abc123...sha256",  # hash of the prompt text or ID of prompt version

  "prompt_text": "[optional] full prompt string or reference",
```

```
"inputs": { ... },                # the data snapshot input (market, macro, etc.)

"llm_output": { ... },            # the LLM signal JSON (as described in section 2)

"quant_checks": {

  "ev_gross": 0.0162,

  "letf_decay": 0.0030,

  "ev_net": 0.0132,

  "viability_pass": true

},

"human_review": {

  "required": true,

  "reviewer_id": "jane.doe@example.com",

  "approval": "approved",

  "notes": "Approved for execution given Fed meeting tomorrow."

},
```

```
  "execution_plan": {
    "orders": [ {...} ],
    "sor_policy": "participation_rate=5%"
  },
  "audit_hash": "def456...sha256",
  "signature": "operator|JaneDoe"
}
```

Fields explanation:

- decision_id: A unique identifier (UUIDv4 or similar) for this decision. Used for cross-referencing in logs.
- timestamp_utc: When the decision was finalized.
- model_version: Precisely which model was used (including version number or epoch if relevant). E.g., `"openai:gpt-4-0613"` or a local model hash. This ensures traceability if model updates happen later.
- prompt_hash: A SHA-256 hash of the prompt (system + user messages) given to the LLM. This allows verification that the exact same prompt text was used. We also version-control prompts, so this hash corresponds to a particular prompt version file (see Prompt Versioning below).
- prompt_text: Optionally store the full prompt or a reference to it (since it might be long, one could store it externally identified by the hash). For

full internal audit, storing it is ideal.

- inputs: The full data snapshot used. This can be large; we include it to allow reconstruction. In cases of storage concern, one might store a reference to an archived data file instead, but direct inclusion ensures self-containment of the record.

- evidence: As part of `llm_output` in our earlier example, but we could also elevate it for convenience. The blueprint shows evidence array outside llm_output in some listing. In our structure above, evidence is within llm_output. Either way, it's present.

- quant_checks: Our deterministic calculations (EV, decay, etc.) and viability boolean. This shows auditors what the numbers were and that they matched the coded logic.

- human_review: Records any human-in-the-loop action. Initially, `required` is set by the decision engine. If a human actually reviews, the `reviewer_id` (email or name) is filled, `approval` is set to "approved" or "rejected" (or "escalated" if passed to higher authority), and any notes (maybe the rationale for override, etc.). If no review was needed, `required=false` and the rest can be null/empty.

- execution_plan: The orders and SOR instructions, with any fill info as available.

- audit_hash: A SHA-256 hash of the entire record except the signature. This is for integrity checking – any alteration of the record will change the hash. Optionally, one could chain these hashes (each record stores hash of previous record to form a tamper-evident ledger).

- signature: A digital signature field. This could be the system signing the record (with a key) and/or a human operator co-signing if they approved a trade. For simplicity, we might just mark "system" if fully automated or the operator's name if they approved. In a full implementation, an actual cryptographic signature could be stored here to ensure non-repudiation.

5.2 Logging and Storage: Each Decision Record is written to an append-only log. Options include:

- Writing as a line of JSON to a text log file (e.g., `decisions_2025.jsonl` for all decisions, one per line). This is simple

and easy to back up.

- Inserting into a database (SQL or NoSQL). A document database (like MongoDB) could store JSON as is; a relational DB could be used but may be overkill.

- Storing in a blockchain or immutable ledger if desired for maximum integrity, though that's beyond our scope for now.

We ensure logs are time-stamped (we have timestamp in record) and kept for at least the regulatory minimum (often 7 years). As part of deployment, set up a secure storage or at least nightly backups of these logs to prevent loss.

5.3 Replay and Deterministic Reconstructability: A core principle: *"Every decision must be reconstructible. Given the record, one must be able to replay the inputs and arrive at the same numeric signal and recommendation."*.

To support this:

- The stored `prompt_text` and `model_version` allow us to query the same model with the same prompt. However, note: Many LLMs are nondeterministic even with same prompt unless we fix a random seed or use temperature 0. We use temperature 0 to mitigate randomness. In a local model, one could set the RNG seed. For an external API, we rely on the model's inherent determinism at temp=0 (OpenAI's GPT at temp=0 is usually deterministic).

- Alternatively, we do not strictly require rerunning the LLM if the output is stored. The numeric decision can be reconstructed by feeding the stored LLM output back into the decision engine code. Essentially, our replay function would do: take a past Decision Record, feed its `llm_output` and `inputs` into the decision engine logic (EV calc, etc.) and confirm that the same quant_checks and outcome result. This verifies our code was consistent.

- We can write a replay tool (e.g., a script or notebook) that takes a Decision Record JSON, and prints out the step-by-step reconstruction: use the input snapshot and possibly re-run model if desired, or skip to using stored model output, then recalc EV, etc., and compare to record. This helps auditors trust that no tampering occurred.

5.4 Evidence Archive: For each evidence cited (like a document ID and lines), ensure those sources are archived. If it's an external link, download and store the content around those lines at the time. If it's an internal doc (like a CRS PDF we have on disk), keep that file and version. An auditor should be able to retrieve the exact snippet that was quoted. Our evidence entries have `filecite` fields for that purpose.

We might maintain an `evidence/` folder where we save any external content the first time it's cited, indexed by `source_id`. For example, if `source_id="CME.2025-12-01"` was an implied probability from CME on a date, we store that number or page in the log.

5.5 Audit Reports and Monitoring: On top of raw records, we will likely produce summary reports:

- Daily email or print-out of decisions made, including any that required human intervention.
- A monthly compliance report summarizing key stats (number of trades, P&L, any policy breaches or overrides).
- These can be generated by scanning the logs and collating info.

We also implement alerts for anomalies as described:

- If an output is missing evidence (hallucination flag) more than a threshold (say >5% of decisions), alert.
- Drift detection: Over time, compare the model's predicted p_success vs actual outcomes. We can maintain a rolling calibration metric: e.g., in past 20 trades where p_success ~0.8, did roughly 16 of them succeed? If we find the model is overestimating (only 10 succeeded out of 20, so actual 50% vs predicted 80%), that's a drift. The system could then raise an alert and possibly automatically switch to sandbox until model is recalibrated.
- If evidence completeness falls (e.g., required evidence not provided in X cases), alert or auto-trigger evidence augmentation step.

These monitoring functions would typically be implemented as separate scripts or Jupyter notebooks run periodically, reading from the audit log.

5.6 Prompt & Model Versioning: We emphasize again that any change to the LLM prompt or model must be tracked. Implementation:

- Keep the prompt(s) in a version-controlled repository (e.g., a `prompts/` directory in Git).
- Each prompt file (for instance `prompt_v1.0.txt`) has a header with version number and changelog of what was modified.
- The Decision Record's `prompt_hash` ensures we can verify which prompt file was used for that decision. If a new prompt version is rolled out, that will produce a new hash in subsequent records.
- Immutable versions: Once a prompt version is used in live trading, do not edit that file; create a new one for changes. Also, log model version changes similarly (if switching from one model provider to another or updating model weights).

This discipline is crucial for compliance: auditors can see exactly what instructions the AI was following for each trade.

With the audit system in place, we can demonstrate the required evidence and control to regulators or stakeholders. Next, we cover how to test the system thoroughly before live deployment, and the tools used for that.

6. Backtesting and Sandbox Testing with Jupyter

Before risking any capital (even in a pilot), one must backtest the Hedge Engine strategy and perform sandbox run-throughs. We provide Jupyter notebooks as part of the implementation to facilitate this. These notebooks serve both as a development sandbox and a way to validate the strategy logic with historical data.

6.1 Backtesting Framework: We create a notebook (e.g., `backtest_hedge_engine.ipynb`) which will:

- Load historical market data for a relevant period (e.g., past 10 years or specific crisis scenarios).

- Simulate stepping through time (either daily or event-driven) and at each step:
    - Provide the data to the LLM (or a stub in lieu of the actual LLM to save on API calls).
    - Get a signal (if using the real LLM in backtest, one might limit to key dates or use a cheaper/distilled model for speed; alternatively, have a simplified decision logic that approximates the strategy for backtest).
    - Run the Decision Engine on the signal, get a decision (and log it).
    - If a trade is executed, update the portfolio holdings and track P&L.
- Repeat for each time step, accumulating performance metrics.

We will likely not use the actual GPT-4 for thousands of backtest steps due to cost. Instead, possible approaches:

- Use a proxy model or heuristic: For backtest purposes, we might encode some of the strategy rules directly (e.g., if interest rates rising and yield curve inverting, do X) to generate signals. These rules can be derived from the e-book's logic. This bypasses the LLM but tests the rest of the system.
- Or focus on specific historical events (a handful) and use the LLM to simulate how it would respond, rather than a continuous daily simulation.

In either case, the notebooks can help verify:

- The EV gate logic works (we can simulate scenarios where EV is negative and see it block).
- The risk limits catch extreme cases.
- The audit logs populate correctly.

6.2 Example Backtest Scenario: We might run a backtest for 2020 (including the COVID crash):

- Start with an initial portfolio allocation (say 60% equities, 40% bonds).
- The data layer loads macro and market data day by day.
- As February/March 2020 approaches, the model (or our proxy logic) should trigger hedges (like increase cash or buy inverses) as volatility spikes and recession indicators flash.
- We verify that the Hedge Engine would have reduced drawdown compared to a 60/40 static portfolio (as claimed by the strategy).
- Check that after the crisis, it re-risks appropriately.
- The notebook will output performance charts – e.g., equity curve of Hedge Engine vs S&P 500, drawdown curves, etc., to validate the strategy's efficacy (these were hinted in the e-book and should be reproduced for verification).

6.3 Sandbox Live Testing: Another notebook or script (`sandbox_run.ipynb`) can be used to run the system in near-real-time but with the sandbox execution:

- It might connect to a live data feed (or a simulated real-time feed from historical data) and run the engine loop continuously (e.g., check every hour for signals, but only actually act if there's a significant event).
- All orders go to the stub, so no money moves.
- This is essentially a dry run of deployment in a controlled environment. It helps to test end-to-end integration: data fetching, LLM API calls, decision gating, human approval flow (we can simulate an operator by manually approving in the notebook if needed), and execution logging.

We can simulate, for example, a Fed interest rate announcement day in sandbox:

- The data layer feeds in an interest rate surprise at 14:00.

- The LLM might generate a signal to adjust portfolio (say, reduce equities, increase bonds).
- The decision engine sees `p_confidence` moderate, flags for approval. In the notebook, we then mimic the operator clicking "Approve".
- The execution stub "trades".
- We verify the Decision Record logged contains the evidence (maybe citing Fed announcement text and market reaction).

6.4 Notebook Structure and Outputs: The notebooks should be well-documented, with sections corresponding to:

- Setup (import libraries, load config, define stub classes).
- Define or load the strategy logic (the prompt, or a simplified decision model).
- Run through data timeline.
- Collect performance metrics.
- Plot results (using matplotlib or similar).
- Print a table of key risk metrics: CAGR, volatility, max drawdown, Sharpe ratio, etc., comparing Hedge Engine vs benchmark. The e-book references such comparison to illustrate the strategy's benefit.

These notebooks not only validate performance but also serve as a tutorial for users to understand how the system behaves. We include them in the repository and mention how to run them in the README.

6.5 Unit Testing vs Backtest: We differentiate that unit tests (discussed in next section) will test individual components for correctness, whereas backtest notebooks test the *strategy as a whole* for effectiveness. Both are important: passing unit tests ensures our code does what we intend; a successful backtest gives confidence the strategy achieves the desired outcome.

After backtesting and addressing any discovered issues (for example, maybe adjusting thresholds if we find the model too often triggers manual review unnecessarily, etc.), we proceed to deployment considerations.

7. Deployment: Docker-Compose Setup

To make the system easy to run, we provide a Docker Compose configuration that containerizes the Hedge Engine and its supporting services. This ensures a consistent environment for all users (developers, testers, even production deployment on a server) and encapsulates dependencies.

7.1 Container Architecture: We use a multi-container setup:

- hedge-engine-app: The main application container running the strategy (data layer, LLM call, decision engine, execution stub, audit writing). This could be a Python application.
- hedge-engine-ui: (Optional) a small web app for the Operator Console. Could be a Flask or Streamlit app that reads the audit log and provides buttons for approvals and kill switch.
- database: (Optional) if using a DB for logs or config. Possibly a small PostgreSQL or MongoDB if needed. If logs are just file-based, we might skip this.
- monitoring: (Optional) we could have a container for a Jupyter notebook server to run notebooks or a Grafana/Prometheus stack for monitoring, but not necessary for pilot.

However, in a minimal pilot, we might combine the UI and app into one container (for simplicity).

7.2 Dockerfile for hedge-engine-app: We base it on a Python image, install required libraries (requests for API calls, pandas for data handling, openai API client for LLM calls, etc.). Also include any CLI scripts (like a runner script that initiates the main loop).

7.3 Docker Compose Configuration: An example `docker-compose.yml` snippet:

version: '3.8'

```
services:

  hedgeengine:

    build: ./hedge-engine-app

    container_name: hedge_engine_app

    environment:

      - OPENAI_API_KEY=${OPENAI_API_KEY}        # example of passing
secrets

      - MODE=sandbox                            # config for running mode

      - CONFIG_FILE=/app/config/config.yaml

    volumes:

      - ./data/:/app/data/                      # mount data or logs

      - ./logs/:/app/logs/

    command: ["python", "-u", "main.py"]        # run the main loop

  hedgeui:
```

build: ./hedge-engine-ui

container_name: hedge_engine_ui

ports:

- "8080:8080"

volumes:

- ./logs/:/app/logs/ # UI needs access to logs to display

depends_on:

- hedgeengine

We also include a `docker-compose.override.yml` for local development if needed (to mount source code for live edit).

7.4 Configuration Management: The `config.yaml` might include:

- API keys (or environment can supply those).
- List of allowed instruments (universe).
- Risk limits (like max_notional_trade, adv_limit, etc.).
- Policy profile (confidence thresholds, etc. can be defined in config or separate policy file).

- Mode flags (live/sandbox, etc.).

This config can be baked into the image or mounted at runtime for flexibility.

7.5 Running Locally: With Docker Compose, the user would:

- Install Docker.
- Set environment variable for any API keys (if using external LLM or data).
- Run `docker-compose up --build`.
- The Hedge Engine app should start, possibly do an initial data fetch and then wait/schedule next actions.
- The UI (if included) would be accessible at `localhost:8080` showing status and awaiting any approvals.

7.6 Scheduling and Orchestration: We might run the Hedge Engine app in two modes:

- As a loop with a sleep. E.g., it wakes up every hour, checks if any scheduled event (like market open or some calendar event) and runs a cycle. This can be controlled via config (like cron expressions).
- Or triggered by an external scheduler (could use cron on the host to hit an endpoint that triggers the strategy). For simplicity, embedding a schedule in code (using `schedule` library or a simple sleep loop) is fine.

7.7 Logging in Docker: The container can simply output logs to stdout/stderr (captured by Docker) and also write detailed logs to files in the mounted volume (`./logs/decision_log.jsonl`, etc.). This allows operators to tail logs or inspect after the fact.

7.8 Dependencies: Ensure the docker image contains all needed system libs. For instance, if using any plotting (for notebooks) or TA-Lib for technical analysis,

include those. Since this is a self-contained environment, everything needed for a full run or replay should be inside.

7.9 Example of Docker Compose Up/Down: This manual should include instructions, e.g.:

- To start in sandbox mode: `docker-compose -f docker-compose.yml up -d`.
- To shut down: `docker-compose down`.
- If moving to production or cloud, similar instructions apply (maybe using Docker Swarm or Kubernetes, but that's beyond scope; Compose is enough for a small scale).

7.10 Security: Since this might involve an API key for the LLM, the compose file uses an environment variable rather than hard-coding it. Advise the user not to commit keys to repo. For compliance, document how keys are managed (perhaps in an `.env` file that is not checked in).

At this stage, we assume the system is deployed locally (or on a server via Docker) and ready to run. We next ensure that the codebase is well-tested and continuously integrated.

8. Testing and Continuous Integration

A robust test suite and CI pipeline ensure the Hedge Engine is reliable and changes do not introduce regressions or new risks. We outline the unit test layout and how to set up GitHub Actions CI for the project.

8.1 Unit and Integration Tests:

We create a `tests/` directory in the repository with tests covering each module:

- `test_data_layer.py`: Tests the data fetching and formatting. For example:
    - Simulate API responses (use dummy data or saved JSON) and ensure the data snapshot produced matches expected structure.

    - Test that missing fields or bad data are handled (e.g., if an indicator is missing, we still form JSON with maybe a null or default).

    - Test timestamping and that the snapshot is consistent (no changing between function calls if same input).

- `test_llm_prompt_format.py`: Tests functions related to preparing the prompt or parsing the LLM output.

    - For instance, if we have a function `format_prompt(context)` that inserts JSON, test that it properly escapes/serializes and the string is as expected.

    - If using function calling, test the function schema registration.

- `test_llm_output_validation.py`: Feed some sample outputs (both valid and invalid) to the validation logic.

    - If the model output is missing a field, ensure our validator catches it.

    - If the output is not valid JSON, ensure we handle the error (maybe by raising an exception or returning False).

- `test_ev_calculation.py`: Specifically test the EV computation and decay.

    - Example: If p_success=0.5, fav=0.1, unfav=-0.1, ensure EV = 0 (with proper formula).

    - Test LETF decay: e.g., leverage=2, sigma=0.2, 1 day horizon -> decay should be ~0 (or one can test a known scenario: if alt daily moves up/down).

    - Perhaps include a test with a small Monte Carlo to see that our decay estimate is reasonable (not too far off).

- `test_decision_engine.py`: This can test the overall logic on a few crafted scenarios.

    - Create a fake `llm_output` for a straightforward case (high confidence, good EV) and ensure the decision engine sets

viability_pass true and no human review required.

    - Another with borderline EV and low confidence to see it flags for human.

    - Use known input and expected output (maybe use the examples from the blueprint as reference outcomes).

- `test_risk_engine.py`:

    - Test vol targeting math (if recent_vol == target_vol, scale_factor =1; if double, scale=0.5; test the clamp at bounds).

    - Test VaR calc on a simple case (maybe known distribution).

    - Test that risk flags (like leverage limit) trigger correctly by simulating a portfolio and a trade that would breach limit.

- `test_execution_agent.py`: Using the sandbox stub:

    - Test that given an order, the stub sets status to filled and logs price correctly.

    - If we simulate a limit price above current market, ensure stub does not fill (if that's how we design it) until price reaches.

    - Test that kill switch flag prevents the stub from "filling" (we might implement stub such that it checks a global or config).

- `test_audit_log.py`:

    - Test that a Decision Record can be serialized to JSON (no non-serializable types).

    - Test the `audit_hash`: compute on a dummy record, flip a value, recompute, ensure it changes.

    - If implementing chain-hash, test integrity linking.

    - Test the replay function: create a dummy record, run replay (which should recompute EV, etc.) and assert it matches the

record's stored values.

Many of these tests will use small dummy objects or monkeypatching (e.g., monkeypatch the LLM call to return a predetermined output so we can test the rest deterministically).

8.2 Running Tests: The project can use `pytest` as the framework. Running `pytest` should execute all tests. Aim for good coverage especially on critical math and logic (expected value, gating conditions).

8.3 Continuous Integration (CI): We set up GitHub Actions (assuming the repo is on GitHub) to run tests on each push and pull request:

Create a workflow YAML, e.g., `.github/workflows/ci.yml`:

```
name: HedgeEngine CI

on: [push, pull_request]

jobs:

  build-and-test:

    runs-on: ubuntu-latest

    steps:

      - uses: actions/checkout@v3

      - uses: actions/setup-python@v4
```

```
      with:
        python-version: '3.10'
    - name: Install dependencies
      run: pip install -r requirements.txt
    - name: Run tests
      run: pytest --exitfirst --disable-warnings -q
```

- This will automatically install and run tests. We might exclude integration tests that call external APIs (or better, mock them in tests).
- We will also add possibly a Lint job (using flake8 or black) to ensure code style consistency and catch simple errors.
- If we have Docker build, we could add a job to build the Docker image to catch any Dockerfile issues (maybe not push it, just build).
- Optionally, a job to run the notebooks (there are ways to execute notebooks in CI to ensure they run without error – though they might be slow, so perhaps skip heavy ones).

8.4 Test Data and Determinism: We should include some static test data files (like a small CSV of prices, or JSON of an example macro input) for tests to use, rather than calling live APIs. This ensures tests run offline and deterministically. Those can reside in `tests/test_data/`.

8.5 Continuous Delivery (CD): While not explicitly asked, as a forward-looking note: one might set up a workflow to build and publish a Docker image (to Docker Hub or a registry) on release. Also, for documentation, using GitHub Pages to host the e-book or a summary of the manual could be considered. But for now, focus on CI for code reliability.

By maintaining rigorous tests and CI, any code changes (like updating the prompt template or modifying risk thresholds) can be automatically validated. This is especially important if multiple team members collaborate – it catches issues before deployment. Compliance will appreciate that any change goes through a tested pipeline with results logged (we could even require code review approvals via PRs for changes to critical files like the prompt or risk logic, as part of process).

Now that we have built and tested the system, we prepare guidelines for the operator to actually run and manage it.

9. Operator Playbook

The Operator Playbook provides runbooks and procedures for those overseeing the Hedge Engine in practice. Even though much is automated, human oversight is crucial especially during a pilot. This section addresses deployment, monitoring, incident handling, and shutdown procedures.

9.1 Launch and Deployment Checklist:

- Pre-launch checks: Ensure all configuration is correct and secrets are in place. Verify the allowed instruments list (Definitive ETF Universe) is up-to-date and approved by compliance. Double-check that `MODE` is set to `sandbox` or limited risk for initial run.

- Start system: Run `docker-compose up` (or equivalent) to start the engine. Confirm the data feeds connect (watch logs for any errors fetching data).

- Initial output: The engine might produce a decision on startup (for example, if it runs at market open). The operator should verify that the decision record looks reasonable (open the latest log entry). This first dry run can be done with `human_review` forced on everything (you can configure policy so that `required=true` for all trades initially). That way, nothing executes without manual approval until trust is built.

- Set policy profile: Operators can configure a conservative policy profile at first:

    - e.g. Only allow auto-exec if trade size < $50k, portfolio leverage change < 0.05, `p_confidence ≥ 0.85` and `ev_net > 1.5%`. This ensures only the most clear-cut trades

go through.

    - All others queue for approval.

    - This policy can be in a YAML (like `policy.yaml`) and loaded by the engine at runtime.

- Communication: Ensure the team (and possibly any broker or tech support contacts) know the system is running and how to reach the operator if issues.

9.2 Monitoring Routine:

- The operator should regularly monitor the Operator Console UI or logs. Key things to watch:

    - Heartbeat: Is the system running on schedule? If it's supposed to run daily at 9am, confirm it did. We can implement a heartbeat log or even an alert if no decision made in 24h (system might have silently stopped).

    - Decisions and Pending Approvals: If the console shows a trade waiting for approval, the operator should review it promptly. The UI will display the Decision Record details including rationale, evidence excerpts, EV calc, etc., to inform the decision. For example:

        - *Evidence:* Fed minutes (filecite:R.12|L12-L26), CME futures implied rate cut 70%.

        - *EV net:* +3.8%, viability pass: true.

        - *Proposed trade:* Buy 200 SHY (Treasuries ETF) and sell 100 SPY.

        - The operator would verify if this aligns with their expectation and whether evidence is indeed convincing.

    - Alerts: If any anomaly triggers (drift, missing evidence, etc.), the console or logs should flag it. E.g., "ALERT: Model

hallucination rate exceeded 5%, check data/prompt quality."

    - Performance Dashboard: Over days, monitor how the portfolio is performing vs benchmarks. Large deviations might signal an issue or an opportunity to adjust risk.

- The operator should maintain a log book (even informal) of notable events: e.g. "2025-12-20: Approved trade to short XLF ahead of Fed meeting – reasoning: aligned with our macro view." This is helpful for retrospective and for additional compliance context.

9.3 Human Approval Process: When a decision is flagged for manual approval:

- The system should pause execution for that decision, logging `mark_for_approval`.
- The operator reviews via console or CLI. They should check:
    - Are the evidence sources trustworthy and relevant?
    - Are the assumptions reasonable? (Does p_success seem overstated?)
    - Does the trade fit within overall portfolio strategy (not over-concentrated, etc.)?
- If satisfied, the operator triggers an Approve action (perhaps clicking a button or running a command). This will update the `human_review` fields in the record (the system logs the reviewer's ID and time) and then allow the Execution Agent to proceed with orders.
- If not satisfied, they can Reject the trade. The system will log it as such and will not execute the plan. The portfolio remains unchanged. The operator might then possibly adjust something (e.g., manually hedge if they disagree with the AI but still see risk).
- If unsure, the operator can escalate (`approval = escalated`) to seek a second opinion (if team structure allows). In the meantime, no action is taken by the system on that decision until resolved.

9.4 Incident Response:
Despite precautions, issues can occur. Some scenarios and responses:

- Data Feed Failure: If a data source fails (say an API doesn't return), the engine might not have the context to make a decision. By design, it should halt and not trade if critical data is missing. Operator action: investigate feed, possibly switch to backup data source (we should have ability to plug in a backup or use last available data). Document the incident and resolution.

- Model Output Anomaly: If the LLM returns gibberish or highly unusual suggestion (e.g., suggesting to buy an outlandish asset or the JSON is malformed repeatedly):

    - The system likely will catch malformed JSON and not act. The operator should use the kill switch or set system to shadow mode until the cause is found (maybe the model is trending or was updated unexpectedly). Possibly roll back to a previous model version if needed.

    - If output is valid JSON but a bizarre trade (like extremely high leverage position without good reason), operator should reject it and probably switch to manual oversight mode. This might indicate the model misinterpreted something – retrain or recheck prompt after.

- Execution Failure: If a live order fails (e.g., broker rejects order, or partial fill with remainder stuck):

    - The execution stub in sandbox will simulate success, but in real, such cases can happen. The operator might need to manually intervene (cancel orders from broker interface if hung, etc.).

    - The system should ideally detect no fill confirmation and alert after X minutes, then give operator option to cancel or re-send.

- Loss Exceeding Stop: If a position is losing beyond the stop-loss and the system hasn't exited (maybe because LLM hasn't run yet or didn't signal exit):

    - Operator should manually trigger an override trade to close the position (via the console or directly in broker). We could

implement a “close position” button on console that instructs Execution Agent to immediately trade out of a given symbol.

    - Then investigate why the stop logic didn’t trigger (perhaps add a more real-time stop watcher in future).

It’s wise to have a dry-run of incident response before going live: e.g., simulate a needed kill switch activation.

9.5 Kill Switch Procedure:

- The Hedge Engine has a Global Kill Switch configuration that can be toggled by the operator (likely a button in UI or a command-line flag, or a hardware switch for ultimate safety).
- Two-person rule: It’s recommended that re-activating live trading after a kill requires two authorized people (to prevent one rogue actor from re-enabling a dangerous system). Implementation can be manual (two people agree and sign off) or technical (two separate keys needed).
- In our system, kill switch might simply be an environment variable `KILL_SWITCH=true` that the engine checks. Setting it true can be done by an operator command. To enforce two-person for reactivation, one approach: require two separate config changes or approvals (not implemented in code, but operationally enforced by policy).
- When kill switch is engaged, the app should immediately stop sending orders and preferably stop asking the LLM for new decisions (to avoid confusion). It can still log that kill is active and maybe still record what it *would* have done in shadow (this is “shadow mode” – the agent continues to simulate but not trade).
- Testing: The operator should test the kill switch at least 3 times in simulation as per guidelines to be comfortable with it.

9.6 Routine Maintenance:

- End-of-day checks: Ensure logs are saved and backed up. Verify that the number of Decision Records equals number of intended cycles (if any missing, find out why).

- Model updates: If a new LLM model or prompt version is to be deployed, do so during off hours in sandbox mode first. For example, if OpenAI releases a new model, test it on recent data in sandbox for a week before switching.

- Periodic retraining/calibration: Over time, if drift detection shows the model's probabilities aren't calibrated, you may want to fine-tune the model or adjust the prompting. This should go through a proper change management (update prompt version, document, test in sandbox, then deploy).

- Scaling up: If the pilot is successful, to scale to real money or more AUM:

    - Increase trade size gradually (the blueprint suggests launching with small AUM and expanding after 30 days of stability).

    - Possibly run parallel for a period (one instance in shadow mode vs one with small live orders) to compare performance and ensure no divergence or technical issues.

9.7 Operator's Golden Rules:

- *Always know the current portfolio exposure.* Understand what the AI has done so far so you aren't surprised by a position.

- *Trust but verify the AI's reasoning.* The evidence and rationale are there to help you trust the decision. If anything is unclear, err on the side of caution (reject or ask for more info).

- *Document overrides.* If you override the AI (either approving something against default rules or canceling a suggested trade), document why. This helps improve the system (maybe the prompt could be adjusted to handle that scenario).

- *Safety first.* This means adhering to kill switch, sandbox testing for changes, and not succumbing to pressure to execute trades you don't fully understand.

By following this playbook, the operator acts as a critical layer of safety and accountability on top of the automated system.

10. Compliance and Governance

Deploying an AI-driven trading strategy requires careful compliance oversight. This section outlines how to fulfill compliance requirements: disclosures, checklists, and procedures from pilot to production.

10.1 Disclosure Templates: If this strategy is managing external capital or even just communicating to stakeholders, provide appropriate disclosures:

- Clearly state that an AI (LLM) is involved in decision-making. For example: *"This strategy uses an AI model (GPT-based) to generate trading signals. While the system has built-in safeguards and human oversight, AI outputs can be unpredictable. Investors should be aware of the experimental nature of this approach."*

- Risk disclosures: *"Investing involves risk of loss. Leveraged ETFs (if used) carry additional risks including volatility decay*[*quant.stackexchange.com*](quant.stackexchange.com)*. There is no guarantee the strategy will achieve its objectives of loss mitigation."* Mention specific known risks (model could misread data, etc.).

- Performance is hypothetical in backtests and not indicative of future results (until actual track record is established).

- Any compliance statements required by law (e.g., if you are a registered investment advisor, ensure all standard disclosures about not guaranteeing accuracy of AI, etc.).

We likely prepare a short document or appendix that can be shared with compliance departments or clients outlining how the AI is used and controlled.

10.2 Internal Compliance Checklist: Before going live, compliance officers (or whoever is in charge of oversight) should verify:

- All trades instruments are approved and within mandate (for instance, ensure the "Definitive ETF Universe" is documented and the system cannot trade outside it unless specifically allowed).

- The model and data sources do not use any non-public or insider information (e.g., using politicians' public PTR filings is okay because they are public, but ensure no misuse of material non-public info).

- Data privacy: If any personal data was used (likely not, mostly financial data), ensure compliance with privacy laws. Not relevant here except maybe if we have user credentials or something (which we handle securely).

- Audit trail sufficiency: Confirm that the Decision Records capture what compliance needs – rationale, evidence, timestamps, who approved what. We expect auditors to actually inspect a few Decision Records; our format was built for that.

- Sign-offs: The compliance officer should sign off on the Implementation Addendum (this manual) to indicate they understand the controls and approve the system's operation. Essentially, treat this manual as a living policy document – any change to the strategy or model means compliance should re-approve the updated manual.

10.3 Pilot-to-Live Transition Procedures:

- Start with Paper Trading Phase: run the system in sandbox for a set period (e.g., one month) with no real trades. Analyze the decisions it would have made and their outcomes. If issues are found, adjust before live.

- Limited Live Phase (Canary): Trade with minimal capital or position sizes (like 10% of intended full size) for another period (e.g., 1-3 months). During this, keep a very tight watch (perhaps require human approval on almost all trades still). Evaluate performance and any operational hiccups.

- Only after these phases, consider Full Deployment: Even then, ramp up gradually. As the blueprint said, launch with small AUM and expand after ~30 days of stable ops.

- At each phase gate, hold a review meeting: go over all trades and decisions, see if anything unexpected happened, ensure everyone is comfortable to proceed.

10.4 Ongoing Compliance Oversight:

- Schedule regular audits (e.g., quarterly) where an independent party (could be internal compliance or external auditor) reviews a sample of Decision Records. They would check:
    - Did the system follow its rules? (Compare decisions to policy thresholds at the time.)
    - Were all required approvals actually obtained and logged?
    - Are the evidence citations legit and not hallucinated?
    - Was the prompt or model changed without proper version update and sign-off?
- Maintain a compliance log of changes: any prompt updates, model swaps, parameter changes should be logged with who approved it. We already require a signed changelog for prompt edits.
- If the strategy is offered to clients, ensure marketing materials align with what's actually implemented (no overpromising, use actual backtest results for reference, clearly labeled).

10.5 Regulatory Considerations:

- Depending on jurisdiction, using AI in trading might attract questions from regulators. Be prepared to explain:
    - The purpose of the AI (it's not an unsupervised black box; it's a codified strategy turned into rules for the AI to follow).
    - The controls in place (human oversight, thresholds, audit logs, etc.).
    - How you prevent the AI from doing something rogue (we have kill switch, restricted universe, and deterministic constraints).
    - Data used by AI (all public or permissioned data).
    - That the AI does not learn on the fly in an uncontrolled manner (no self-modifying without human deployment of new

prompt/model).

Document these answers internally. Possibly add them as an FAQ in compliance docs.

10.6 Contingency for Model Unavailability:

- If the external LLM API is down or becomes too expensive, what's the plan? It might be to pause trading (system already should fail safe by not trading if no signal).

- Consider having a fallback strategy (like a basic rule-based strategy) that can take over if AI is unavailable, to avoid having no hedge at all in place. But that fallback should also be documented and tested.

10.7 End-of-Pilot Decisions:

- If the pilot underperforms or shows the concept is not viable, be ready to decommission or revise the approach. Compliance will want a controlled shutdown plan too (basically unwind positions and turn off).

- If successful, scale up but also consider additional controls that a larger scale might require (e.g., if managing external money, you may need real-time risk monitoring, disaster recovery setup, etc.).

Finally, treat this Implementation Manual as a living document. It was noted that any change to the system requires an update to this addendum and a new signed revision. That means the manual should always reflect the current implementation, and any deviation needs to be reconciled. This practice ensures transparency and discipline as the Hedge Engine evolves.

Conclusion: By following this comprehensive implementation guide, a professionalizing retail investor or small quant team can confidently deploy the Hedge Engine strategy in a controlled environment. We combined cutting-edge LLM-driven signal generation with rigorous risk management and audit controls to create a system that is both innovative and responsibly managed. The direct, no-nonsense approach in this manual is intended to make each step clear. With

this, you should be able to stand up a safe, auditable pilot instance of the Hedge Engine using open source tools and readily available market data, proceeding from concept to reality with oversight at every stage.

Implementation Manual — Hedge Engine

Rolling out an LLM-prompted Macro ETF Strategy (Practical Developer + Compliance Guide)

> Short version: Put everything the engineers and compliance people need in **one Implementation Addendum** inside the e-book, and place the runnable artifacts in a companion GitHub repository. The Addendum is the authoritative roadmap; the repo is the toolbox. This manual is that Addendum — a full, prescriptive, runnable blueprint for a Python + GitHub implementation that a professionalizing retail investor or a small quant team can operate and audit.

Who this manual is for

This is written for the devs, quants and compliance folks who will actually build, test and operate **Hedge Engine**. You are not a pure academic and you don't want 400 pages of probability theory — you want code examples, operational checklists, test recipes, and governance that regulators and auditors can read without falling asleep.

If you're a technically-capable retail investor looking to professionalize: this manual gives you the exact architecture, the minimal runnable code to get started, the test harness and the compliance checklist. You'll still need server infra, brokerage access, and a lawyer, but this gets you to a real sandbox and a safe pilot.

Tone: pragmatic, plainspoken, and unapologetically practical. No fluff. We include runnable Python snippets, Docker guidance, CI examples, and operator playbooks.

Table of contents (jump to anything)

---

1. Goals & deliverables

Primary goal: Deliver a single, auditable, reproducible implementation Addendum and companion repo so a professionalizing retail investor or small quant team can run Hedge Engine in a sandbox and progressively move to live with governance.

Minimum deliverables (what this manual must guarantee):

- A precise Implementation Addendum (this manual).
- A companion GitHub repo structure with:
    - LETF decay simulator (`decay_sim.py`) + unit tests.
    - EV calculator / pre-trade gates (`ev_calc.py`) + tests.
    - Risk engine (`risk_engine.py`) including vol target and VaR.
    - Execution stub with sandbox broker (`execution_stub.py`).
    - Audit/Decision Record glue (`audit.py`).
    - Example Jupyter notebooks: full backtest + sandbox run.
    - Small sandbox dataset (ETFs, LETF NAVs, options snapshots, event calendar).
    - `docker-compose.yml` for full sandbox.
    - CI (pytest + notebook validation).
- An LLM prompt bank as JSON + examples and system messages.
- Operator playbook & compliance checklist.

This manual is written so you can drop it in the e-book as the Implementation Addendum and point users to the repo for runnable artifacts.

---

2. Implementation Addendum: single document vs repo

Short answer: Yes — put all the specification, architecture, rules, and runbook into a single Implementation Addendum inside the e-book. But *don't* embed the runnable artifacts in the PDF. Instead, **link** the Addendum to a companion GitHub repository that contains the runnable code, datasets, and CI. The Addendum is the canonical spec; the repo is the implementation.

Why this split?

- PDFs are version-controlled badly. Repos are the natural home for code, data and unit tests.
- Auditors and compliance like an immutable policy in the document and a reproducible codebase.
- It's easier to fix a bug and tag a release on GitHub than to update a 200-page PDF every time.

Implementation Addendum structure (what goes in the PDF):

- Manifest/architecture diagram + mapping to repo filenames.
- Decision Record and schema (immutable canonical JSON).
- Precise operator policies and human-in-the-loop gates.
- Risk & disclosure language (legal text).
- Runbook: commands to run notebooks, tests and simulation, with expected outputs and checksums.
- Developer checklist to sign off before live.

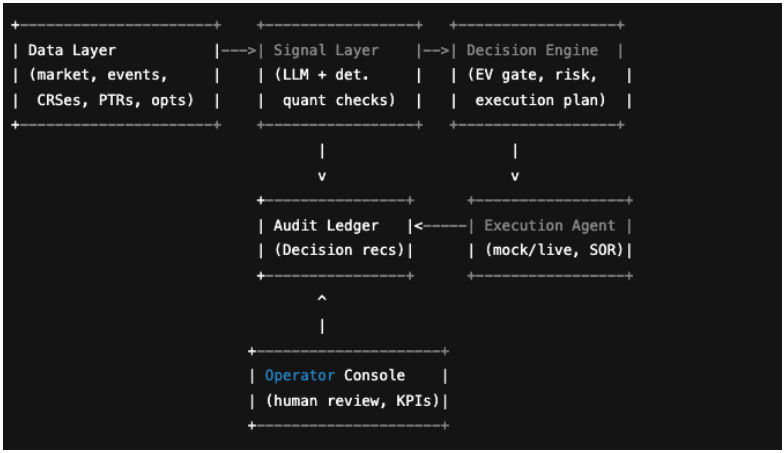
Companion repo:

- All runnable modules, tests, sample data and Docker composition.
- Release tags, CI pipeline and signed artefacts.

3. Architecture overview (manifest)

Below is the canonical architecture. Copy this exact diagram into the Addendum and the repo README.

High-level components:



Data Layer details

- Macro feeds (FRED, BEA, BLS) — used for cycle detection. Use free APIs for initial implementation.
- Market data (ETF daily OHLCV, LETF NAV series, options chains, futures) — best via broker API (IB) or Alpha Vantage / IEX for sandbox.
- Event calendar (Fed meetings, court dates, legislative schedules, PTR files) — curated CSV.

Signal Layer

- Primary LLM agent for discovery (signals, probability, rationale) but **must** output structured JSON only.
- Deterministic quant checks: options skew, momentum, liquidity filter, PTR swarm aggregator.

Decision Engine

- EV calculation, LETF decay sim, viability gate.
- Risk engine (vol targeting, VaR, stops).
- Execution plan generator (orders + SOR policy).

Execution Agent

- Sandbox broker for simulation (Dockerized mock).
- Live broker connector (Interactive Brokers recommended) with SOR/TWAP wrappers.

Audit

- Append-only Decision Records with prompt hash, model version, evidence, quant checks, human approvals, orders.

Operator Console

- UI or simple CLI to review flagged decisions and approve or reject.

---

4. Data & sandbox datasets (formats & sample)

To make anything reproducible you need a small curated sandbox. Ship these files in `/data`:

1) `sandbox_etf_prices.parquet`
Columns: `date, ticker, open, high, low, close, adj_close, volume`
Tickers included: `SPY, XLK, XLV, XLY, TLT, IEF, GLD, SHY, SMH, SOXX, SSO` (a small but representative universe).

2) `sandbox_letf_navs.parquet`
Columns: `date, letf_ticker, nav`
Purpose: letf NAVs are required to simulate real LETF daily reset returns. Include 3× and 2× LETFs for SPY and SMH for 2 years.

3) `sandbox_options_chains.parquet` (optional small sample)
Columns: `date, ticker, expiry, strike, call_put, bid, ask, iv, underlying_price`
Include a handful of snapshots for NVDA-like underlying around an earnings event.

4) `event_calendar.csv`
Columns: `date, event_type, name, metadata` where `event_type` ∈ {FOMC, EARNINGS, COURT_RULING, PTR_RELEASE}.

5) `ptr_sample.csv`
A small PTR feed with `date, member, ticker, side, amount_range`.

Format and provenance notes

- All datasets should include a `source` and a `sha256` checksum in a `dataset_manifest.json`.
- Use parquet for performance. Provide small CSV equivalents for initial users who prefer that.

---

5. Core runnable modules (what goes in repo)

Each module below must be accompanied by docstrings, type hints, and tests.

`src/decay_sim.py` — LETF Decay Simulator

- API: `simulate_lef_decay(underlying_returns: np.array, leverage: float, trials: int=10000, window: int) -> dict`
- Outputs distribution stats: mean, median, pctiles, worst-case.

- Implement daily compounding: LETF_daily_return = (1 + leverage * r_i). Simulate bootstrapped windows and compute cumulative returns.

`src/ev_calc.py` — EV & Viability Gate

- API: `compute_ev(signal, scenarios, fees, decay, slippage) -> dict`
- Steps:
    1. Use LLM-produced `p_success` and `expected_delta` to compute EV_gross.
    2. Subtract LETF_decay (from `decay_sim`) and transaction costs.
    3. Return `ev_net`, `viability_pass` boolean.

src/pretrade_checks.py

- Liquidity filters (ADV thresholds, bid-ask cap), instrument whitelist, leverage horizon rules (T_max per LETF).

src/risk_engine.py

- Vol-targeting module (`compute_scale_factor(target_vol, recent_vol, limits)`), VaR estimator (parametric/variance-covariance as a baseline), stop-loss manager, drawdown trigger.

src/execution_stub.py

- `execute_order(execution_plan, mode='sandbox')` with sandbox mock broker. Implement TWAP and percent-of-ADV participation and simulate fills with slippage model.

src/audit.py

- Decision Record creation, hashing, sign/verify placeholder, replay function.

notebooks/01_backtest_hedge_engine.ipynb

- End-to-end reproducible backtest using sandbox data.

notebooks/02_sandbox_run.ipynb

- Live sandbox run: LLM in shadow mode, EV gate, execution stub, audit log.

docker-compose.yml

- Compose with Python service, mock broker (HTTP), a simple Redis for telemetry, and a local Postgres for audit logs.

---

6. LETF decay simulator — full implementation & tests

This is the single most important runnable piece. We supply a complete example here.

> **Key idea:** LETF returns are path-dependent. You cannot multiply the underlying return by leverage over multiple days. You must simulate daily compounding.

Example implementation (Python)

```python
# src/decay_sim.py
import numpy as np
import pandas as pd
from typing import Dict, Any

def simulate_lef_decay(underlying_returns: np.ndarray,
                       leverage: float,
                       window: int,
                       trials: int = 20000,
                       seed: int = 42) -> Dict[str, Any]:
    """
    Monte Carlo bootstrap simulation of LETF T-day returns.
    - underlying_returns: historical daily returns (as decimals)
    - leverage: e.g. 3.0 for 3x
    - window: holding days
    - trials: simulation trials
    """
    rng = np.random.default_rng(seed)
    n = len(underlying_returns)
    # bootstrap sample starting indices
    starts = rng.integers(low=0, high=n-window, size=trials)
    results = np.empty(trials)
    for i, s in enumerate(starts):
        slice_returns = underlying_returns[s:s+window]
        # daily LETF factor
        letf_factors = 1.0 + leverage * slice_returns
        # cumulative factor; if any daily factor negative (extreme), cap
        cumulative = np.prod(letf_factors)
        results[i] = cumulative - 1.0
    stats = {
        "mean": float(np.mean(results)),
        "median": float(np.median(results)),
        "p10": float(np.percentile(results, 10)),
        "p90": float(np.percentile(results, 90)),
        "worst": float(np.min(results)),
        "best": float(np.max(results)),
        "trials": trials,
        "results": results  # optional large object
    }
    return stats
```

Unit test (pytest):

```python
# tests/test_decay_sim.py
import numpy as np
from src.decay_sim import simulate_lef_decay

def test_decay_sim_basic():
    rng = np.random.default_rng(123)
    # create synthetic underlying returns ~ daily vol 1%
    underlying = rng.normal(0, 0.01, 1000)
    stats = simulate_lef_decay(underlying, leverage=3.0, window=5, trials=1000)
    assert 'mean' in stats
    assert stats['trials'] == 1000
    # mean should be finite
    assert np.isfinite(stats['mean'])
```

Interpretation and usage:

- Run simulation for the ETF index underlying each LETF, for the candidate `horizon_days`. Use results to compute `letf_decay = -stats['mean']` (if negative drag) and worst-case metrics for risk sizing.
- Store simulation results in Decision Record `quant_checks.letf_decay`.

Why this matters: If `EV_net = P_success * expected_delta - letf_decay - fees` is negative, you *cannot* justify using the LETF for that horizon. You must switch to futures/options or shorten the horizon.

---

7. EV gate & pre-trade checks — implementation

The EV gate is the deterministic guardrail. We implement it in `ev_calc.py`.

`ev_calc.py` skeleton (Python)

```python
# src/ev_calc.py
from typing import Dict

def compute_ev(signal: Dict, decay_stats: Dict, trading_costs: float, slippage: float) ->
    """
    signal: {
        'p_success': 0.8,
        'expected_delta': {'fav': 0.12, 'neutral': 0.0, 'unfav': -0.05},
        'horizon_days': 10,
        'suggested_instrument': {'type': 'LETF', 'ticker': 'SSO', 'leverage': 2}
    }
    decay_stats: output from simulate_lef_decay (mean etc)
    trading_costs: expressed fraction
    slippage: expressed fraction
    """
    p = signal['p_success']
    delta = signal['expected_delta']['fav']  # assume favour case for EV calc
    ev_gross = p * delta + (1-p) * signal['expected_delta']['unfav']
    letf_decay = - decay_stats['mean'] if decay_stats['mean'] < 0 else 0.0
    ev_net = ev_gross - letf_decay - trading_costs - slippage
    viability_pass = ev_net > 0.01  # safety margin 1% default
    return {
        'ev_gross': ev_gross,
        'letf_decay': letf_decay,
        'ev_net': ev_net,
        'viability_pass': viability_pass
    }
```

Pre-trade checklist (deterministic):

- Instrument allowed (in approved universe).
- Horizon ≤ T_max for LETF (T_max is a function of underlying vol and acceptable decay).
- Avg daily volume and bid-ask spread thresholds.
- Portfolio concentration limits checked.
- Human approval required if `p_confidence < 0.7` or `horizon_days > T_max` for LETF.

Note on `T_max`: Determine `T_max` empirically per LETF and underlying vol. A safe default: for a 3× LETF with daily vol > 25%, `T_max` may be <= 5 trading days. Implement `T_max` function:

```python
def compute_t_max(leverage, est_vol_annual):
    # rough heuristic
    if leverage >= 3.0:
        return max(1, int(5 * (0.20 / est_vol_annual)))  # 5 days at 20% vol baseline
    if leverage >= 2.0:
        return max(1, int(10 * (0.20 / est_vol_annual)))
    return 30
```

Calibrate `T_max` with real simulations and record values in the Decision Record.

---

8. Execution stubs + sandbox broker + SOR policy

Execution policy summary

- Orders default to **limit**; only use market orders for small notional (< 0.1% NAV) or emergency.
- **Participation cap**: not more than 10% of 10-minute ADV. Default: 5% for medium liquidity ETFs.
- For block trades use TWAP with dynamic step size and slippage cap.
- Slippage guard: abort if projected slippage > allowed bps.

`execution_stub.py` (simplified)

```python
# src/execution_stub.py
import time
from typing import Dict

def execute_order(execution_plan: Dict, mode='sandbox'):
    """
    execution_plan:
      {
         'orders': [{'ticker':'SPY','side':'BUY','qty':100,'type':'LIMIT','limit':...}],
         'sor_policy': 'percent_of_adv=0.05',
         'max_slippage_bps': 5
      }
    Mode sandbox: calls a mock broker that returns fills with simulated slippage.
    """
    if mode == 'sandbox':
        return _mock_execute(execution_plan)
    else:
        # integrate with broker SDK (e.g., IB)
        return _live_execute_via_ib(execution_plan)

def _mock_execute(plan):
    results = []
    for o in plan['orders']:
        # simulate partial fill and slippage
        slippage = 0.0002  # 2 bps
        fill_price = o.get('limit', 100.0) * (1 + slippage if o['side']=='BUY' else 1 - s
        results.append({'ticker': o['ticker'], 'qty': o['qty'], 'fill_price': fill_price,
        time.sleep(0.1)
    return {'status': 'ok', 'fills': results}
```

Sandbox broker: Use a minimal HTTP server that accepts orders and returns simulated fills. This can be a tiny Flask app in Docker.

Live broker: Provide an example wrapper for Interactive Brokers (ib_insync) in `src/brokers/ib_wrapper.py` — but require user config and two-factor verification. Keep the live wrapper gated by operator approval.

---

9. Risk engine — vol targeting, VaR, stops

Vol targeting (implementation details)

```python
# src/risk_engine.py
import numpy as np

def compute_scale_factor(target_annual_vol: float, returns: np.ndarray,
                         max_scale: float=2.0, min_scale: float=0.3):
    recent_vol = np.std(returns) * np.sqrt(252)
    scale_factor = target_annual_vol / (recent_vol + 1e-9)
    scale_factor = min(max(scale_factor, min_scale), max_scale)
    return scale_factor, recent_vol
```

VaR module (parametric)

- Compute 1-day 99% parametric VaR using portfolio exposures and covariance.
- Provide an alternative historical VaR using bootstrapped returns.

Stop rules & emergency drawdown

- `emergency_drawdown_trigger = -0.10` (10% peak-to-trough) default. On trigger: cut exposure to `min_exposure_floor = 0.3` of prior exposure and immediately notify operators.

Telemetry

- Publish: gross leverage, net exposure, VaR 1-day 99%, realized vol, drawdown, viability_pass_rate. Use Redis or a small time-series DB for telemetry.

---

10. Audit, Decision Record & replay

Decision Record (canonical JSON) — required for every signal.

Example (trimmed):

```
{
  "decision_id":"uuid",
  "timestamp_utc":"2025-12-01T14:05:00Z",
  "model_version":"gpt-fin-2025-11-v3",
  "prompt_hash":"abcd1234",
  "inputs": {
    "market_snapshot": {...},
    "macro_inputs": {...}
  },
  "evidence":[
    {"source_id":"CRS.R47525","type":"CRS","filecite":"R.12|L12-L26","excerpt":"..."}
  ],
  "llm_output":{
    "p_success":0.8,
    "p_confidence":0.87,
    "horizon_days":5,
    "expected_delta":{"fav":0.12,"neutral":0.00,"unfav":-0.05},
    "suggested_instrument":{"type":"LETF","ticker":"SSO","leverage":2}
  },
  "quant_checks":{
    "ev_gross":0.08,
    "letf_decay":0.002,
    "ev_net":0.05,
    "viability_pass":true
  },
  "human_review":{
    "required":false
  },
  "execution_plan":{...},
  "audit_hash":"sha256"
}
```


Audit requirements (enforced):

- Every decision: has `prompt_hash` and `model_version`.
- Evidence array must include at least two distinct sources for `p_confidence` ≥ 0.7.
- If `viability_pass` is false, do not execute automatically.
- All Decision Records written to append-only DB (Postgres with `wal` + object store or an immutable S3 prefix) and hashed with `sha256` stored in separate ledger.

Replay

- Provide `src/audit_replay.py decision.json` which:
    - Loads `decision.json`.
    - Replays deterministic `ev_calc`, `decay_sim` and compares outputs.
    - Returns `mismatch` if outputs differ in quant values.

Why this matters: Regulators and auditors will ask: *how did you get here?* This JSON is the answer.

---

11. LLM meta-prompts & "deep research → writing" function

Your e-book described a “deep research to writing” function: research runs in the background (latent) and is used to gather evidence, check sources, and generate visualizations, but research content must **not** be exposed directly to the final manuscript — only the polished product.

Below is a practical pattern that’s auditable and safe.

Design principles

1. **Research kept latent:** LLM may fetch source excerpts, but **the output of research is only evidence items** (source ids, short excerpt) stored in Decision Record, never raw scraped text.
2. **Structured evidence:** Each evidence item includes `source_id`, `type`, `filecite` (location), and 1-sentence excerpt (<= 200 chars). No raw docs are output to the manuscript.
3. **Visualization generation:** LLM suggests which chart to create; deterministic code pulls the numeric arrays and generates the chart (matplotlib or plotly), saved as PNG in repo/artifacts. The LLM *does not* output the raw data.
4. **LLM as assistant only:** LLM produces structured JSON signals only. Deterministic code interprets it and does heavy numeric work and

charting. The final manuscript text is assembled from templated sections plus vetted LLM explanations that are sanitized and validated.

System message (required)

Embed an unambiguous system message in the custom GPT:

SYSTEM: You are HEDGE ENGINE RESEARCH AGENT. Output ONLY structured JSON per schema. For 'evidence' provide at most 3 items with source IDs and a one-sentence excerpt. NEVER output full text chunks. For visualizations, request 'chart_spec' and the deterministic engine will create the chart with numeric data. For manuscript copy: output short narrative blocks (<= 300 words) and a list of associated evidence citations. DO NOT output raw research documents.

Example JSON schema for signal generation (LLM)

```
{
 "decision_id": "uuid",
 "llm_signal": {
    "p_success": 0.78,
    "p_confidence": 0.82,
    "horizon_days": 7,
    "expected_delta": {"fav":0.12, "neutral":0.02, "unfav":
-0.06},
    "trade_strength": 0.45
 },
 "evidence": [

{"source_id":"CRS.R4725","type":"CRS","filecite":"R.12|L12-L
26","excerpt":"CRS concluded..."},

{"source_id":"PTR.202401","type":"PTR","filecite":"PTR.51|L1
-L3","excerpt":"Swarm of 5 MoCs bought chips..."}
 ],
 "chart_spec": {
    "type":"line",
    "x":"date",
    "y":"smh_nav",
```

```
    "annotation":"CHIPS Act passage date"
 },
 "manuscript_snippet": "Short paragraph arguing the policy
persistence..."
}
```

Deterministic engine responsibilities

- Validate `evidence[]` completeness and fetch excerpts from canonical local store for replay. Do not accept an LLM `evidence` item that lacks a `filecite`.
- Run `decay_sim` for the suggested horizon and produce `letf_decay`.
- Render `chart_spec` using deterministic code and save PNG; include PNG path in Decision Record.
- Assemble manuscript blocks: deterministic template merges sanitized LLM prose + chart image references + evidence list. This output is the only textual product written into the final manuscript.

Example "deep research → writing" flow

1. **LLM research call** (shadow): LLM reads the strategy doc + current data, produces `llm_signal`, `evidence`, `chart_spec`, and `manuscript_snippet`. It MUST include `evidence` items.
2. **Deterministic validation**: `ev_calc` runs and appends `quant_checks`. Chart renderer produces images.
3. **Human review (if required)**: If `p_confidence < 0.7` or `ev_net` borderline, route to human.
4. **Manuscript assembly**: deterministic code assembles final copy for the e-book chapter — includes composite narrative, chart PNG, and footnoted citations — but **never** includes raw research text.

Why this satisfies the requirement: Research is latent (evidence only), visualizations and calculations are deterministic and reproducible, final manuscript copy is optimized and sanitary.

---

12. Notebooks, backtests & example flows

Notebook 1: `01_backtest_hedge_engine.ipynb`

Contents:

- Load sandbox data.
- Run cycle detection (rule based).
- Run sample LLM (mock) generating signals for a set of events.
- Run `decay_sim` and `ev_calc` for each signal.
- Apply vol targeting and rebalance monthly.
- Track metrics: equity curve, drawdowns, Sharpe, Calmar.
- Save Decision Records and export CSV of top decisions.

Notebook 2: `02_sandbox_run.ipynb`

Contents:

- Start a sandbox broker (or point to Docker service).
- Generate an LLM signal; route through viability checks; if viability pass => execute via sandbox broker.
- Record fills in the audit DB and show telemetry.
- Demonstrate a human approval flow for a flagged decision.

How to run (Addendum copy/paste commands)

```
# clone repo
git clone git@github.com:yourorg/hedge-engine-impl.git
cd hedge-engine-impl
docker-compose up --build -d
pip install -r requirements.txt
# run tests
pytest -q
# open notebook
jupyter lab notebooks/01_backtest_hedge_engine.ipynb
```

Expected outputs: equity curve PNG, Decision Records in `data/decision_records/`, logs, and chart PNGs under `artifacts/`.

---

13. Unit tests, CI and release policy (GitHub Actions example)

Example `github-actions.yml`

```
name: CI
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'
      - name: Install deps
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Run tests
        run: pytest -q
      - name: Validate notebooks
        run: |
          pip install nbval
          pytest --nbval notebooks/01_backtest_hedge_engine.ipynb -q
      - name: Lint
        run: |
          pip install flake8
          flake8 src
```

Test coverage

- `tests/test_decay_sim.py` — decay sim tests extreme cases (zero vol, high vol).
- `tests/test_ev_calc.py` — tests EV logic and boundaries.
- `tests/test_execution_stub.py` — ensures SOR policy honored.
- `tests/test_decision_record_replay.py` — replay an example Decision Record and assert outputs are identical.

Release policy

- Tag releases with `vX.Y.Z`.
- Release assets: wheel, docker image, `artifacts` (baseline outputs), and dataset manifest checksums.
- Notify compliance when promt_hash or model_version changes.

---

14. Operator playbook: deploy, monitor, incidents & kill switch

Daily operations

- Morning check: run `notebooks/02_sandbox_run.ipynb` in shadow mode for 30 minutes. Confirm `viability_pass` rates and hallucination metric < 2%.
- After market open: check human review queue. Approve/reject flagged decisions within 30 minutes.
- End of day: run `audit_replay.py` for all Decision Records and ensure reproducibility.

Incident response (kill switch protocol)

1. **Detect**: telemetry alert: hallucination rate > 5% or slippage > threshold or execution errors.

2. **Kill**: operator triggers global kill switch (Docker env variable or cloud managed flag) that toggles execution agent into `shadow` mode.

3. **Collect**: gather Decision Records, logs, model_version and prompt_hash.

4. **Triage**: root cause analysis (data outage? model change?).

5. **Contain**: if needed, run unwind scripts (pre-approved) to spline off risky positions.

6. **Remediate**: patch code/prompt. Run 7-day shadow replay.

7. **Report**: produce forensic report for compliance.

Two-person custody for kill switch reactivation.

---

15. Compliance & disclosures (exact text for Addendum)

Include the following risk disclosure verbatim in every copy and require a signed acknowledgement before `docker-compose up` can connect to any live broker.

> **RISK DISCLOSURE & LIMITATIONS**
> This Implementation Addendum and related artifacts are for educational and research purposes and do NOT constitute investment, legal or regulatory advice. Past performance is not indicative of future results. The use of leveraged ETFs and derivatives carries risk of rapid and permanent loss of capital. LLMs can hallucinate; all probabilistic outputs must be validated by deterministic gates and human oversight. Any live deployment requires legal and compliance sign-off and registration as required by law. The authors accept no responsibility for trading losses or misuse of the materials.

Operator sign-off checklist (must be accepted before live):

- Legal counsel approval: yes/no

- Compliance policy signed: yes/no
- KYC/AML procedures in place: yes/no
- Insurance / institutional custody: yes/no
- Designated kill-switch custodians (2 persons): names & contacts

---

16. Migration to live: pilot → small live → scale

Phase 0 — Local sandbox (developer)

- Run `01_backtest_hedge_engine.ipynb`. Run `02_sandbox_run.ipynb` to verify decision record creation and replay.

Phase 1 — Shadow live (paper)

- Connect to real-time market feed, keep LLM in shadow and only record Decision Records. Validate calibration `p_success` vs realized frequency for 90 days.

Phase 2 — Small live (low notional)

- Allow auto-execution but cap max notional per trade (e.g., $50k) and max gross leverage < 20% of target. Active ops watching. Human must approve any LETF trade with horizon > 7 days.

Phase 3 — Gradual scale

- Increase AUM stepwise after 90 days of stable metrics and audit sign-off. Monitor concentration risk and add institutional grade execution plumbing.

---

17. Appendix: JSON schemas, templates & sample code

Decision Record schema (canonical)

(We included earlier; add to repo as `schemas/decision_record.json`.)

Prompt templates (place these under `/prompts` in repo)

System prompt (copy into Addendum and repo):

SYSTEM: You are HEDGE ENGINE GPT — a disciplined macro portfolio assistant. Output ONLY valid JSON that matches the "llm_signal" schema. Provide an 'evidence' array with at least 2 sources for p_confidence≥0.7. Do not output raw research documents. Limit manuscript_snippet to <= 300 words.

```
USER: SIGNAL_GENERATION
Context: {market_snapshot, macro_indicators, options_skew, ptr_swarm}
Task: Provide JSON with:
- p_success (0–1),
- p_confidence (0–1),
- horizon_days (int),
- expected_delta {fav, neutral, unfav},
- suggested_instrument {type,ticker,leverage},
- evidence[] {source_id,type,filecite,excerpt},
- chart_spec (optional),
- manuscript_snippet (<= 300 words)
Return only JSON.
```

Example human approval note template

makefile — Copy code

```
Human approval required because p_confidence < 0.7.
Reviewer: {name}
Decision: Approved / Rejected
Notes: {rationale}
Signature: {electronic signature}
```

Final developer checklist (copy/paste)

1. Clone repo and run `docker-compose up`.
2. Run `pytest` and ensure all tests pass.
3. Execute `notebooks/01_backtest_hedge_engine.ipynb` end-to-end using sandbox data. Validate equity curve and Decision

Records.

4. Run `src/decay_sim.py` for target LETFs and freeze `T_max` values in `config/t_max.json`.

5. Start sandbox broker and run `02_sandbox_run.ipynb` to simulate live execution.

6. Confirm audit replay works: `python src/audit_replay.py data/decision_records/example.json` -> exact match.

7. Confirm human approval gates: set `p_confidence=0.65` in a mock signal and ensure it routes to `human_review`.

8. Perform a red-team: simulate a model drift where LLM outputs `p_success` > 0.95 for nonsense evidence. System must flag hallucination rate > threshold.

9. Compliance sign-off and kill switch testing with two custodians.

Code Index:

`README.md` (quickstart)

markdown

```
# Hedge Engine Implementation (Skeleton)

This repository contains a runnable skeleton for Hedge Engine: an LLM-prompted Macro ETF

## Quickstart (local sandbox)

1. Clone:
   ```bash
   git clone <repo-url>
   cd hedge-engine-impl
```

2. Create venv & install:

bash

```
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

3. Generate sandbox data:

bash

```
python data/generate_sandbox_data.py
```

4. Run tests:

bash

```
pytest -q
```

5. Run sandbox runner (mock LLM + mock broker):

bash

```
python scripts/run_sandbox.py
```

6. Run backtest:

bash

```
python scripts/run_backtest.py
```
```

Repo layout

See the repository root for `src/` modules: `decay_sim`, `ev_calc`, `risk_engine`, `execution_stub`, `audit`.

Notes

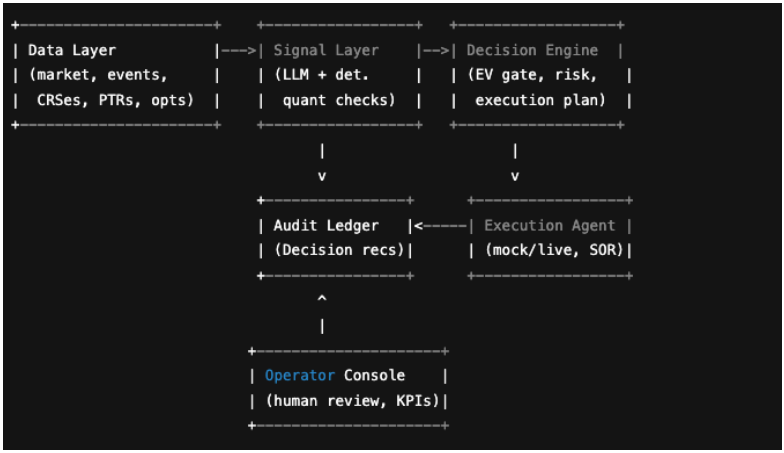
- All execution defaults to `mode='sandbox'`. Live broker integrations are gated behind operator approval and are not enabled by default.
- The `prompts/` folder contains the enforced research→writing prompt and JSON schema used in the Implementation Addendum.

yaml Copy code

```
---

# `requirements.txt`
```

numpy>=1.24
pandas>=1.5
pytest>=7.0
flask>=2.0
pyyaml
nbval
plotly
python-dateutil



```python
# src/decay_sim.py
import numpy as np
import pandas as pd
from typing import Dict, Any

def simulate_lef_decay(underlying_returns: np.ndarray,
                       leverage: float,
                       window: int,
                       trials: int = 20000,
                       seed: int = 42) -> Dict[str, Any]:
    """
    Monte Carlo bootstrap simulation of LETF T-day returns.
    - underlying_returns: historical daily returns (as decimals)
    - leverage: e.g. 3.0 for 3x
    - window: holding days
    - trials: simulation trials
    """
    rng = np.random.default_rng(seed)
    n = len(underlying_returns)
    # bootstrap sample starting indices
    starts = rng.integers(low=0, high=n-window, size=trials)
    results = np.empty(trials)
    for i, s in enumerate(starts):
        slice_returns = underlying_returns[s:s+window]
        # daily LETF factor
        letf_factors = 1.0 + leverage * slice_returns
        # cumulative factor; if any daily factor negative (extreme), cap
        cumulative = np.prod(letf_factors)
        results[i] = cumulative - 1.0
    stats = {
        "mean": float(np.mean(results)),
        "median": float(np.median(results)),
        "p10": float(np.percentile(results, 10)),
        "p90": float(np.percentile(results, 90)),
        "worst": float(np.min(results)),
        "best": float(np.max(results)),
        "trials": trials,
        "results": results  # optional large object
    }
    return stats
```

```python
# tests/test_decay_sim.py
import numpy as np
from src.decay_sim import simulate_lef_decay

def test_decay_sim_basic():
    rng = np.random.default_rng(123)
    # create synthetic underlying returns ~ daily vol 1%
    underlying = rng.normal(0, 0.01, 1000)
    stats = simulate_lef_decay(underlying, leverage=3.0, window=5, trials=1000)
    assert 'mean' in stats
    assert stats['trials'] == 1000
    # mean should be finite
    assert np.isfinite(stats['mean'])
```

```python
# src/ev_calc.py
from typing import Dict

def compute_ev(signal: Dict, decay_stats: Dict, trading_costs: float, slippage: float) ->
    """
    signal: {
        'p_success': 0.8,
        'expected_delta': {'fav': 0.12, 'neutral': 0.0, 'unfav': -0.05},
        'horizon_days': 10,
        'suggested_instrument': {'type': 'LETF', 'ticker': 'SSO', 'leverage': 2}
    }
    decay_stats: output from simulate_lef_decay (mean etc)
    trading_costs: expressed fraction
    slippage: expressed fraction
    """
    p = signal['p_success']
    delta = signal['expected_delta']['fav']  # assume favour case for EV calc
    ev_gross = p * delta + (1-p) * signal['expected_delta']['unfav']
    letf_decay = - decay_stats['mean'] if decay_stats['mean'] < 0 else 0.0
    ev_net = ev_gross - letf_decay - trading_costs - slippage
    viability_pass = ev_net > 0.01  # safety margin 1% default
    return {
        'ev_gross': ev_gross,
        'letf_decay': letf_decay,
        'ev_net': ev_net,
        'viability_pass': viability_pass
    }
```

```python
def compute_t_max(leverage, est_vol_annual):
    # rough heuristic
    if leverage >= 3.0:
        return max(1, int(5 * (0.20 / est_vol_annual)))  # 5 days at 20% vol baseline
    if leverage >= 2.0:
        return max(1, int(10 * (0.20 / est_vol_annual)))
    return 30
```

```python
# src/execution_stub.py
import time
from typing import Dict

def execute_order(execution_plan: Dict, mode='sandbox'):
    """
    execution_plan:
      {
        'orders': [{'ticker':'SPY','side':'BUY','qty':100,'type':'LIMIT','limit':...}],
        'sor_policy': 'percent_of_adv=0.05',
        'max_slippage_bps': 5
      }
    Mode sandbox: calls a mock broker that returns fills with simulated slippage.
    """
    if mode == 'sandbox':
        return _mock_execute(execution_plan)
    else:
        # integrate with broker SDK (e.g., IB)
        return _live_execute_via_ib(execution_plan)

def _mock_execute(plan):
    results = []
    for o in plan['orders']:
        # simulate partial fill and slippage
        slippage = 0.0002  # 2 bps
        fill_price = o.get('limit', 100.0) * (1 + slippage if o['side']=='BUY' else 1 - s
        results.append({'ticker': o['ticker'], 'qty': o['qty'], 'fill_price': fill_price,
        time.sleep(0.1)
    return {'status': 'ok', 'fills': results}
```

```python
# src/risk_engine.py
import numpy as np

def compute_scale_factor(target_annual_vol: float, returns: np.ndarray,
                         max_scale: float=2.0, min_scale: float=0.3):
    recent_vol = np.std(returns) * np.sqrt(252)
    scale_factor = target_annual_vol / (recent_vol + 1e-9)
    scale_factor = min(max(scale_factor, min_scale), max_scale)
    return scale_factor, recent_vol
```

```
{
  "decision_id":"uuid",
  "timestamp_utc":"2025-12-01T14:05:00Z",
  "model_version":"gpt-fin-2025-11-v3",
  "prompt_hash":"abcd1234",
  "inputs": {
    "market_snapshot": {...},
    "macro_inputs": {...}
  },
  "evidence":[
    {"source_id":"CRS.R47525","type":"CRS","filecite":"R.12|L12-L26","excerpt":"..."}
  ],
  "llm_output":{
    "p_success":0.8,
    "p_confidence":0.87,
    "horizon_days":5,
    "expected_delta":{"fav":0.12,"neutral":0.00,"unfav":-0.05},
    "suggested_instrument":{"type":"LETF","ticker":"SSO","leverage":2}
  },
  "quant_checks":{
    "ev_gross":0.08,
    "letf_decay":0.002,
    "ev_net":0.05,
    "viability_pass":true
  },
  "human_review":{
    "required":false
  },
  "execution_plan":{...},
  "audit_hash":"sha256"
}
```


```
# clone repo
git clone git@github.com:yourorg/hedge-engine-impl.git
cd hedge-engine-impl
docker-compose up --build -d
pip install -r requirements.txt
# run tests
pytest -q
# open notebook
jupyter lab notebooks/01_backtest_hedge_engine.ipynb
```

Expected outputs: equity curve PNG, Decision Records in `data/decision_records/`, logs, and chart PNGs under `artifacts/`.

```yaml
name: CI
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'
      - name: Install deps
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Run tests
        run: pytest -q
      - name: Validate notebooks
        run: |
          pip install nbval
          pytest --nbval notebooks/01_backtest_hedge_engine.ipynb -q
      - name: Lint
        run: |
          pip install flake8
          flake8 src
```

```
USER: SIGNAL_GENERATION
Context: {market_snapshot, macro_indicators, options_skew, ptr_swarm}
Task: Provide JSON with:
- p_success (0–1),
- p_confidence (0–1),
- horizon_days (int),
- expected_delta {fav, neutral, unfav},
- suggested_instrument {type,ticker,leverage},
- evidence[] {source_id,type,filecite,excerpt},
- chart_spec (optional),
- manuscript_snippet (<= 300 words)
Return only JSON.
```

Example human approval note template

makefile

Copy code

```
Human approval required because p_confidence < 0.7.
Reviewer: {name}
Decision: Approved / Rejected
Notes: {rationale}
Signature: {electronic signature}
```

```
hedge-engine-impl/
├─ README.md
├─ LICENSE
├─ requirements.txt
├─ docker-compose.yml
├─ .github/
│  └─ workflows/
│     └─ ci.yml
├─ data/
│  └─ generate_sandbox_data.py
├─ src/
│  ├─ __init__.py
│  ├─ decay_sim.py
│  ├─ ev_calc.py
│  ├─ pretrade_checks.py
│  ├─ risk_engine.py
│  ├─ execution_stub.py
│  ├─ audit.py
│  └─ cli.py
├─ prompts/
│  ├─ research_prompt.json
│  └─ signal_template.json
├─ notebooks/
│  └─ NOTEBOOK_PLACEHOLDER.md
├─ tests/
│  ├─ test_decay_sim.py
│  ├─ test_ev_calc.py
│  ├─ test_execution_stub.py
│  └─ test_audit_replay.py
└─ scripts/
   ├─ run_backtest.py
   └─ run_sandbox.py
```

````
# Hedge Engine Implementation (Skeleton)

This repository contains a runnable skeleton for Hedge Engine: an LLM-prompted Macro ETF

## Quickstart (local sandbox)

1. Clone:
   ```bash
   git clone <repo-url>
   cd hedge-engine-impl
````

```
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```
```

2. Create venv & install:

```bash
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

3. Generate sandbox data:

```bash
python data/generate_sandbox_data.py
```

4. Run tests:

```bash
pytest -q
```

5. Run sandbox runner (mock LLM + mock broker):

```bash
python scripts/run_sandbox.py
```

6. Run backtest:

```bash
python scripts/run_backtest.py
```

Repo layout

See the repository root for `src/` modules: `decay_sim`, `ev_calc`, `risk_engine`, `execution_stub`, `audit`.

Notes

- All execution defaults to `mode='sandbox'`. Live broker integrations are gated behind operator approval and are not enabled by default.
- The `prompts/` folder contains the enforced research→writing prompt and JSON schema used in the Implementation Addendum.

yaml Copy code

```
---

# `requirements.txt`
```

numpy>=1.24
pandas>=1.5
pytest>=7.0
flask>=2.0
pyyaml
nbval
plotly
python-dateutil

(You can add `ib_insync` later for Interactive Brokers live integration.)

Key modules — code you can copy/paste

`src/decay_sim.py`
```python
"""
LETF Decay Simulator
simulate_lef_decay: monte carlo bootstrap of LETF T-day returns using historical underly
"""
import numpy as np
from typing import Dict, Any

def simulate_lef_decay(underlying_returns: np.ndarray,
                       leverage: float,
                       window: int,
                       trials: int = 20000,
                       seed: int = 42) -> Dict[str, Any]:
    rng = np.random.default_rng(seed)
    n = len(underlying_returns)
    if n < window:
        raise ValueError("Underlying series shorter than window")
    starts = rng.integers(low=0, high=n - window + 1, size=trials)
    results = np.empty(trials)
    for i, s in enumerate(starts):
        slice_returns = underlying_returns[s:s+window]
        letf_factors = 1.0 + leverage * slice_returns
        cumulative = np.prod(letf_factors)
        results[i] = cumulative - 1.0
    stats = {
        "mean": float(np.mean(results)),
        "median": float(np.median(results)),
        "p10": float(np.percentile(results, 10)),
        "p90": float(np.percentile(results, 90)),
        "worst": float(np.min(results)),
        "best": float(np.max(results)),
        "trials": int(trials)
    }
    return stats

# CLI helper
if __name__ == "__main__":
    import pandas as pd
    import sys
    path = sys.argv[1] if len(sys.argv) > 1 else "data/sandbox_etf_prices.parquet"
    df = pd.read_parquet(path)
    # example: compute returns for SPY
    spy = df[df['ticker']=='SPY'].sort_values('date')
    returns = spy['adj_close'].pct_change('ropna().to_numpy()
    print(simulate_lef_decay(returns, leverage=3.0, window=5, trials=2000))
```

```python
"""
EV calculation and viability gate
"""
from typing import Dict

def compute_ev(signal: Dict, decay_stats: Dict, trading_costs: float, slippage: float, s
    p = float(signal.get('p_success', 0.0))
    expected = signal.get('expected_delta', {})
    fav = float(expected.get('fav', 0.0))
    unfav = float(expected.get('unfav', 0.0))
    ev_gross = p * fav + (1 - p) * unfav
    letf_decay = 0.0
    if decay_stats is not None:
        letf_mean = float(decay_stats.get('mean', 0.0))
        # if mean is negative (drag), capture positive drag value
        letf_decay = max(0.0, -letf_mean)
    ev_net = ev_gross - letf_decay - trading_costs - slippage
    viability_pass = ev_net > safety_margin
    return {
        "ev_gross": ev_gross,
        "letf_decay": letf_decay,
        "ev_net": ev_net,
        "viability_pass": viability_pass
    }

# small helper for t_max heuristic
def compute_t_max(leverage: float, est_vol_annual: float) -> int:
    if leverage >= 3.0:
        return max(1, int(5 * (0.20 / max(est_vol_annual, 0.01))))
    if leverage >= 2.0:
        return max(1, int(10 * (0.20 / max(est_vol_annual, 0.01))))
    return 30
```

```
stats: Dict, trading_costs: float, slippage: float, safety_margin: float = 0.01) -> Dict:
', 0.0))
_delta', {})
 0.0))
av', 0.0))
unfav


ts.get('mean', 0.0))
), capture positive drag value
_mean)
- trading_costs - slippage
y_margin




_pass



est_vol_annual: float) -> int:

 / max(est_vol_annual, 0.01))))

0 / max(est_vol_annual, 0.01))))
```

```python
"""
Liquidity & pre-trade checks
"""
from typing import Dict

def pass_liquidity_filters(ticker_metrics: Dict, adv_threshold_usd: float=1_000_000, spr
    adv = ticker_metrics.get('adv', 0.0)
    spread_bps_actual = ticker_metrics.get('spread_bps', 9999.0)
    if adv < adv_threshold_usd:
        return False
    if spread_bps_actual > spread_bps:
        return False
    return True

def instrument_allowed(instrument: Dict, allowed_universe: list) -> bool:
    return instrument.get('ticker') in allowed_universe
```

```python
s



ticker_metrics: Dict, adv_threshold_usd: float=1_000_000, spread_bps: float=5.0) -> bool:
t('adv', 0.0)
ker_metrics.get('spread_bps', 9999.0)
usd:

spread_bps:



rument: Dict, allowed_universe: list) -> bool:
ticker') in allowed_universe
```

```python
Vol targeting, VaR and stop rules (simple baseline implementations)
"""
import numpy as np
from typing import Tuple

def compute_scale_factor(target_annual_vol: float, returns: np.ndarray, max_scale: float
    recent_vol = float(np.std(returns) * np.sqrt(252))
    if recent_vol <= 0:
        return 1.0, recent_vol
    scale = target_annual_vol / recent_vol
    scale = min(max(scale, min_scale), max_scale)
    return scale, recent_vol

def parametric_var(portfolio_returns: np.ndarray, alpha: float=0.01) -> float:
    mu = np.mean(portfolio_returns)
    sigma = np.std(portfolio_returns)
    z = np.abs(np.percentile(np.random.normal(0,1,100000), alpha*100))
    var = -(mu + sigma * z)
    return var
```

```python
e implementations)



, returns: np.ndarray, max_scale: float=2.0, min_scale: float=0.3) -> Tuple[float,float]:
252))




 alpha: float=0.01) -> float:


,100000), alpha*100))
```

python Copy code

```python
"""
Simple execution stub and sandbox mock broker client
"""
import time
import random
from typing import Dict

def _mock_fill_price(limit_price: float, side: str, slippage_bps: float=2):
    slippage = slippage_bps / 10000.0
    sign = 1 if side.upper() == 'BUY' else -1
    return limit_price * (1 + sign * slippage)

def execute_order(execution_plan: Dict, mode: str='sandbox') -> Dict:
    if mode == 'sandbox':
        fills = []
        for o in execution_plan.get('orders', []):
            limit = o.get('limit', None) or 100.0
            price = _mock_fill_price(limit, o.get('side', 'BUY'), slippage_bps=2)
            fills.append({
                'ticker': o['ticker'],
                'qty': o['qty'],
                'fill_price': round(price, 4),
                'filled': True
            })
            time.sleep(0.05)
        return {'status': 'ok', 'fills': fills}
    else:
        raise NotImplementedError("Live broker integration disabled in stub")
```

```python
"""
Audit Decision Record creation and replay
"""
import json
import hashlib
from datetime import datetime
from typing import Dict

def make_decision_record(decision: Dict) -> Dict:
    decision_id = decision.get('decision_id') or hashlib.sha1(json.dumps(decision, sort_
    decision['decision_id'] = decision_id
    decision['timestamp_utc'] = datetime.utcnow().isoformat() + 'Z'
    raw = json.dumps(decision, sort_keys=True)
    audit_hash = hashlib.sha256(raw.encode()).hexdigest()
    decision['audit_hash'] = audit_hash
    return decision

def save_decision_record(decision: Dict, path: str):
    with open(path, 'w') as f:
        json.dump(decision, f, indent=2)

def replay_decision(path: str) -> Dict:
    with open(path) as f:
        decision = json.load(f)
    # In a real replay: re-run deterministic gates and compare numeric fields
    # Here we simply return the record and a boolean indicating presence of required fie
    required = ['llm_signal','quant_checks','prompt_hash','model_version']
    ok = all(k in decision for k in required)
    return {'decision': decision, 'replay_ok': ok}
```

python

Copy code

```python
"""
Simple CLI to run a sandbox flow for manual testing.
"""
import json
from src.decay_sim import simulate_lef_decay
from src.ev_calc import compute_ev
from src.audit import make_decision_record, save_decision_record
import numpy as np
import pandas as pd

def run_sandbox_signal_example():
    df = pd.read_parquet('data/sandbox_etf_prices.parquet')
    spy = df[df['ticker']=='SPY'].sort_values('date')
    returns = spy['adj_close'].pct_change().dropna().to_numpy()
    decay = simulate_lef_decay(returns, leverage=3.0, window=5, trials=2000)
    signal = {
        'p_success': 0.7,
        'expected_delta': {'fav': 0.08, 'neutral':0.0, 'unfav':-0.05},
        'horizon_days': 5,
        'suggested_instrument': {'type':'LETF','ticker':'SSO','leverage':2}
    }
    ev = compute_ev(signal, decay, trading_costs=0.0002, slippage=0.0003)
    decision = {
        'llm_signal': signal,
        'quant_checks': ev,
        'prompt_hash': 'demohash',
        'model_version': 'gpt-demo',
    }
    rec = make_decision_record(decision)
    save_decision_record(rec, 'data/decision_00001.json')
    print('Decision saved: data/decision_00001.json')

if __name__ == "__main__":
    run_sandbox_signal_example()
```

`scripts` examples

`scripts/run_backtest.py`

python Copy code

```python
"""
Lightweight backtest runner to exercise modules.
"""
from src.cli import run_sandbox_signal_example
if __name__ == "__main__":
    run_sandbox_signal_example()
    print("Backtest (demo) finished. See data/decision_00001.json")
```

`scripts/run_sandbox.py`

python Copy code

```python
"""
Demo sandbox runner (would be extended to loop and call LLM).
"""
import time
from src.execution_stub import execute_order
from src.audit import replay_decision
# For demo run we assume decision exists
decision_path = 'data/decision_00001.json'
res = replay_decision(decision_path)
print("Replay status:", res['replay_ok'])
plan = {
  'orders': [{'ticker':'SPY','side':'BUY','qty':10,'limit':430.0}],
  'sor_policy': 'percent_of_adv=0.05',
  'max_slippage_bps': 5
}
fills = execute_order(plan, mode='sandbox')
print("Fills:", fills)
```

Tests

`tests/test_decay_sim.py`

python

```python
import numpy as np
from src.decay_sim import simulate_lef_decay

def test_decay_sim_basic():
    rng = np.random.default_rng(123)
    underlying = rng.normal(0, 0.01, 1000)
    stats = simulate_lef_decay(underlying, leverage=3.0, window=5, trials=1000)
    assert 'mean' in stats
    assert stats['trials'] == 1000
```

`tests/test_ev_calc.py`

python

```python
from src.ev_calc import compute_ev
def test_ev_calc_simple():
    signal = {'p_success':0.8, 'expected_delta':{'fav':0.1,'unfav':-0.05}}
    decay = {'mean': -0.01}
    res = compute_ev(signal, decay, trading_costs=0.0005, slippage=0.0005)
    assert 'ev_net' in res
    assert isinstance(res['viability_pass'], bool)
```

`tests/test_execution_stub.py`

python

```python
from src.execution_stub import execute_order
def test_exec_stub():
    plan = {'orders':[{'ticker':'SPY','side':'BUY','qty':1,'limit':100.0}]}
    out = execute_order(plan, mode='sandbox')
    assert out['status'] == 'ok'
    assert len(out['fills']) == 1
```

`tests/test_audit_replay.py`

python

```python
from src.audit import make_decision_record, replay_decision
def test_audit_replay(tmp_path):
    dec = {'llm_signal':{}, 'quant_checks':{}, 'prompt_hash':'x', 'model_version':'v'}
    rec = make_decision_record(dec)
    p = tmp_path / "d.json"
    from src.audit import save_decision_record
    save_decision_record(rec, str(p))
    out = replay_decision(str(p))
    assert out['replay_ok']
```

`docker-compose.yml` (dev sandbox)

yaml

Copy code

```yaml
version: '3.8'
services:
  app:
    build: .
    volumes:
      - ./:/app
    command: tail -f /dev/null
    ports:
      - "8888:8888"
  mock-broker:
    image: python:3.10-slim
    working_dir: /broker
    volumes:
      - ./src:/broker
    command: python -u -c "from flask import Flask, jsonify, request; app=Flask('b'); @ap
    ports:
      - "5001:5001"
```

Prompts (deep research→writing) — `prompts/research_prompt.json`

json

Copy code

```json
{
  "system": "You are HEDGE ENGINE RESEARCH AGENT — an expert, auditable macro research as
  "user_template": "SIGNAL_GENERATION: {market_snapshot, macro_indicators, event_calendar
}
```

Also include `prompts/signal_template.json` with the `llm_signal` schema (as in the Addendum).

`docker-compose.yml` (dev sandbox)

yaml

Copy code

```
app=Flask('b'); @app.route('/order', methods=['POST'])\n\ndef o(): return jsonify({'ok':Tru
```

Prompts (deep research→writing) — `prompts/research_prompt.json`

json

Copy code

```
- an expert, auditable macro research assistant for Hedge Engine. Output only JSON per sche
napshot, macro_indicators, event_calendar, ptr_summary}. Return JSON with llm_signal, evider
```

Also include `prompts/signal_template.json` with the `llm_signal` schema (as in the Addendum).

`docker-compose.yml` (dev sandbox)

yaml

Copy code

```
app=Flask('b'); @app.route('/order', methods=['POST'])\n\ndef o(): return jsonify({'ok':Tru
```

Prompts (deep research→writing) — `prompts/research_prompt.json`

json

Copy code

```
sistant for Hedge Engine. Output only JSON per schema. For probabilistic claims, include at
, ptr_summary}. Return JSON with llm_signal, evidence[], chart_spec (optional), manuscript_
```

Also include `prompts/signal_template.json` with the `llm_signal` schema (as in the Addendum).

`docker-compose.yml` (dev sandbox)

yaml

Copy code

```
o(): return jsonify({'ok':True,'fill':request.json}); app.run(host='0.0.0.0', port=5001)"
```

Prompts (deep research→writing) — `prompts/research_prompt.json`

json

Copy code

```
nclude at least 2 evidence items with filecite. Do not output raw research text.",
nuscript_snippet (<=300 words). If confidence low, set flags.requires_human_review=true."
```

Also include `prompts/signal_template.json` with the `llm_signal` schema (as in the Addendum).

CI — `.github/workflows/ci.yml`

yaml

Copy code

```yaml
name: CI
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'
      - name: Install deps
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt
      - name: Run tests
        run: pytest -q
```

Sandbox data generator — `data/generate_sandbox_data.py`

python

Copy code

```
-01', days=500, start_price=100.0):
:a(days=i) for i in range(days)]
0xffffffff)

is)
 ticker, 'open': prices, 'high': prices*1.01, 'low': prices*0.99, 'close': prices, 'adj_clos



:F','GLD','SHY','SMH','SOXX','SSO']



:)

:et')



 '3x' in t else 2)
 nav']])
:f_navs.parquet')

:3-06-14'),'event_type':'FOMC','name':'FOMC Jun 2023','metadata':''}])
'alse)
:t and csv")
```

Sandbox data generator — `data/generate_sandbox_data.py`

python

Copy code

```python
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import os

def make_etf_prices(ticker, start_date='2022-01-01', days=500, start_price=100.0):
    dates = [pd.Timestamp(start_date) + timedelta(days=i) for i in range(days)]
    rng = np.random.default_rng(hash(ticker) & 0xffffffff)
    returns = rng.normal(0, 0.01, days)
    prices = start_price * np.cumprod(1 + returns)
    df = pd.DataFrame({'date': dates, 'ticker': ticker, 'open': prices, 'high': prices*1.
    return df

def main():
    tickers = ['SPY','XLK','XLV','XLY','TLT','IEF','GLD','SHY','SMH','SOXX','SSO']
    frames = []
    for t in tickers:
        frames.append(make_etf_prices(t))
    df = pd.concat(frames).reset_index(drop=True)
    os.makedirs('data', exist_ok=True)
    df.to_parquet('data/sandbox_etf_prices.parquet')
    # make LETF navs (mock)
    letf = []
    for t in ['SSO','SSO3x']:
        base = df[df['ticker']=='SPY'].copy()
        base['letf_ticker'] = t
        base['nav'] = base['adj_close'] * (3 if '3x' in t else 2)
        letf.append(base[['date','letf_ticker','nav']])
    pd.concat(letf).to_parquet('data/sandbox_letf_navs.parquet')
    # minimal event calendar
    ev = pd.DataFrame([{'date':pd.Timestamp('2023-06-14'),'event_type':'FOMC','name':'FOM
    ev.to_csv('data/event_calendar.csv', index=False)
    print("Sandbox data written to data/*.parquet and csv")

if __name__ == "__main__":
    main()
```

Sandbox data generator — `data/generate_sandbox_data.py`

python

Copy code

```
: prices, 'adj_close': prices, 'volume': (rng.integers(100000, 1000000, days)).tolist()})
```