F L O F M A T R I X

Fractal Liquidity & Order Flow Trading System

EPISTEMIC ENGINE

TECHNICAL BLUEPRINT

AI Backtesting • Knowledge Graph • Semantic RAG • Dynamic TOML Profiling

4 Layers • 13 Database Tables • 5 Node Types • 6 Edge Types • 12 LLM Prompt Templates
Batch Runner → Trade Logger → Knowledge Graph → LLM RAG Interface

Version:	**3.5 (Epistemic Engine)**
Classification:	**CONFIDENTIAL**

1. What the Epistemic Engine Is

The Epistemic Engine is a meta-layer that sits on top of NautilusTrader and the FLOF Matrix. It does not trade. It observes, records, analyzes, and learns from every trade the bot takes across backtesting, paper trading, and live execution. Over time, it converts raw trade data into structured knowledge that makes the system progressively smarter.

The word "epistemic" means "relating to knowledge and the conditions for acquiring it." The engine's job is to answer the question: given everything this system has done, what should it do differently for this specific instrument in this specific regime?

1.1 The Four Layers

ID	LAYER	TECHNOLOGY	STORES	PURPOSE
E01	**Batch Runner**	Python + NautilusTrader + multiprocessing	Experiment manifests, run configs	Runs thousands of parameterized backtests in parallel. Tests toggle/constant variations. Outputs standardized JSON.
E02	**Trade Logger**	PostgreSQL + pgvector extension	Every trade record + vector embeddings	Structured storage of all trade data. Semantic similarity search via vector embeddings. Single source of truth.
E03	**Knowledge Graph**	Neo4j (graph database)	Instrument/setup/regime relationships	Maps discovered patterns as typed relationships. Answers relational questions: what works with what, and where.
E04	**LLM RAG Interface**	LangChain + Claude API	Prompt templates, conversation logs	Conversational interface. You ask questions, the LLM queries E02 + E03 and responds with data-grounded answers and TOML recommendations.

1.2 Data Flow

STEP	FLOW
1	NautilusTrader executes trades (backtest, paper, or live). After each trade closes, the execution engine emits a TradeEvent.
2	The Trade Logger (E02) receives the TradeEvent, writes a structured record to PostgreSQL, auto-generates a natural language summary, embeds the summary via an embedding model, and stores the vector in pgvector.
3	A nightly batch job reads new trade records, computes aggregate statistics per instrument/setup/regime, and updates the Knowledge Graph (E03) in Neo4j. New edges are created or updated when statistical thresholds are met.
4	The LLM RAG Interface (E04) is always available for ad-hoc queries. When you ask a question, LangChain retrieves relevant chunks from pgvector (semantic search) and relevant subgraphs from Neo4j (graph traversal), then sends both to the Claude API with a structured prompt.

5	If the LLM produces a TOML recommendation (e.g., change a constant for NVDA), it is written to a pending_recommendations table. A human reviews and approves or rejects. Approved changes are applied to the TOML config.

2. Layer E01: Batch Runner

The Batch Runner is the data generator. It runs headless NautilusTrader backtests in parallel, systematically testing toggle and constant permutations across instruments and date ranges.

2.1 Experiment Manifest Schema

Every batch run starts with a JSON experiment manifest that defines what to test. The Batch Runner reads this manifest, generates one TOML config per permutation, and launches a backtest for each.

```
{
  "experiment_id": "EXP-2025-03-15-001",
  "description": "Test T39 Extreme/Decisional impact on AAPL vs NVDA",
  "base_profile": "profile_equities.toml",
  "instruments": ["AAPL", "NVDA", "MSFT"],
  "date_range": {
    "start": "2024-01-01",
    "end": "2024-12-31",
    "in_sample_pct": 0.70,
    "out_of_sample_pct": 0.30
  },
  "vary_toggles": {
    "T39_extreme_decisional": [true, false],
    "T47_dynamic_scaleouts": [true, false]
  },
  "vary_constants": {
    "phase1_a_plus_scaleout": [0.20, 0.25, 0.30],
    "stop_buffer_atr_mult": [0.50, 0.75, 1.00]
  },
  "fill_levels": [1, 2, 3],
  "total_permutations": "auto_calculate",
  "worker_count": 8
}
```

2.2 Execution Pipeline

STEP	PROCESS
1	Parse the manifest. Calculate total permutations: instruments × toggle combos × constant combos = the run matrix. For the example above: 3 instruments × 4 toggle combos × 9 constant combos = 108 backtests per fill level, 324 total.
2	For each permutation, generate a temporary TOML file by applying the overrides to the base profile. Each temp TOML gets a unique run_id hash derived from (experiment_id + instrument + toggle_state + constant_values + fill_level).
3	Launch backtests in parallel using Python multiprocessing.Pool(worker_count). Each worker: loads the temp TOML, initializes NautilusTrader in headless mode, runs the backtest on the specified date range, and captures all TradeEvents.
4	Each completed backtest writes a RunResult JSON containing: run_id, config_hash, instrument, fill_level, toggle_states, constant_values, aggregate_metrics (win_rate, profit_factor, sharpe, max_drawdown, trade_count), and the full list of individual trade records.

5	All RunResults are batch-inserted into the Trade Logger (E02) PostgreSQL database. Individual trades are inserted into the trades table; aggregate metrics go into the backtest_runs table.

2.3 Progressive Fill Filtering

Not all permutations get tested at all fill levels. This saves enormous compute time.

STAGE	FILL LEVEL	FILTER
Screen	Level 1 (Optimistic)	Run ALL permutations. Fast, identifies promising parameter spaces. Results flagged as "unvalidated."
Validate	Level 2 (Standard)	Re-run only the top 10% of Level 1 results (by profit factor). Many will degrade. Only those remaining profitable proceed. Results flagged as "validated-L2."
Confirm	Level 3 (Conservative)	Re-run Level 2 survivors on OUT-OF-SAMPLE data only. Only configurations profitable at Level 3 on unseen data are stored in the Knowledge Graph. Results flagged as "confirmed."

ANTI-OVERFITTING: THE CARDINAL RULE

The Batch Runner automatically splits every date range into 70% in-sample and 30% out-of-sample. Level 1 and Level 2 backtests run on in-sample data only. Level 3 runs on out-of-sample data only. A configuration that scores well in-sample but fails out-of-sample is flagged as OVERFIT and excluded from all Knowledge Graph updates. This is non-negotiable. Overfitting is the single most common failure mode in quantitative research. No finding enters the Knowledge Graph without passing the out-of-sample test.

3. Layer E02: Trade Logger & Vector Database

The Trade Logger is the single source of truth for all trade data. Every trade from any source (backtest, paper, live) writes to the same PostgreSQL database using the same schema. This means you can compare a backtest trade from 6 months ago to a live trade from today using the same queries.

3.1 PostgreSQL Database Schema

3.1.1 Table: trades

COLUMN	TYPE	INDEX	DESCRIPTION
trade_id	VARCHAR(30) PK	**PK**	Unique trade ID. Format: T-{YYYY-MM-DD}-{seq}. Example: T-2024-12-15-001.
run_id	VARCHAR(40)	**IDX**	Links to backtest_runs.run_id for backtest trades. NULL for paper/live trades.
source	VARCHAR(10)	**IDX**	"backtest", "paper", or "live". Essential for filtering analysis by data quality.
instrument	VARCHAR(20)	**IDX**	Symbol. E.g., "ES", "AAPL", "BTC", "EUR/USD".
asset_class	VARCHAR(15)	**IDX**	"futures", "equities", "crypto", "forex", "options".
direction	VARCHAR(5)	—	"long" or "short".
grade	VARCHAR(3)	**IDX**	"A+", "A", "B", or "C".
total_score	SMALLINT	—	0–17. Total confluence score across all tiers.
tier1_score	SMALLINT	—	0–10. Structural + Order Flow score.
tier2_score	SMALLINT	—	0–4. Velez MA score.
tier3_score	SMALLINT	—	0–3. VWAP + liquidity criteria.
poi_type	VARCHAR(20)	**IDX**	ORDER_BLOCK, FVG, REJECTION_BLOCK, LIQUIDITY_POOL, SYNTHETIC_MA, BREAKER_BLOCK, GAP_FVG.
poi_tags	JSONB	—	{"is_extreme": true, "is_unicorn": false, "is_sweep": true, "is_fresh": true}.
entry_price	DECIMAL(12,4)	—	Actual fill price.
stop_price	DECIMAL(12,4)	—	Initial stop price at entry.
result_r	DECIMAL(6,2)	—	P&L in R-multiples. Positive = win, negative = loss. -1.0 = full stop hit.
result_pnl_dollars	DECIMAL(12,2)	—	Dollar P&L for the trade.
exit_reason	VARCHAR(30)	**IDX**	"phase1_target", "structural_trail", "climax_exit", "stop_loss", "tape_failure", "toxicity", "eod_flatten", "risk_overlord", "manual".
regime	VARCHAR(20)	**IDX**	"trending_bull", "trending_bear", "chop", "high_iv", "low_iv", "risk_off".

killzone	VARCHAR(15)	**IDX**	"NY_AM", "NY_PM", "London", "Asian", "Crypto_UTC".
toggles_active	JSONB	—	["T07","T08","T17","T36","T39","T44","T47"]. Array of toggle IDs that were ON.
constants_snapshot	JSONB	—	Key constants at time of trade: {"atr_mult": 0.5, "phase1_pct": 0.25, ...}. Enables correlating outcomes with specific parameter values.
tape_conditions	JSONB	—	{"absorption": true, "whale_block": false, "tape_velocity_pct": 350, "toxicity_fired": false}.
portfolio_gates	JSONB	—	{"P1_passed": true, "P2_passed": true, "exposure_at_entry": 0.032, "group": "A"}. From Portfolio Manager (M16).
duration_minutes	DECIMAL(8,1)	—	Time from entry to final exit.
opened_at	TIMESTAMPTZ	**IDX**	UTC timestamp of entry fill.
closed_at	TIMESTAMPTZ	—	UTC timestamp of final exit.
text_summary	TEXT	—	Auto-generated natural language summary. This field is embedded into the vector DB.
embedding	VECTOR(1536)	**ANN**	pgvector column. 1536-dim embedding of text_summary. Indexed with ivfflat or hnsw for approximate nearest neighbor search.

3.1.2 Table: backtest_runs

COLUMN	TYPE	DESCRIPTION
run_id	VARCHAR(40) PK	Hash of experiment_id + instrument + config. Unique per backtest run.
experiment_id	VARCHAR(30) FK	Links to experiments.experiment_id.
instrument	VARCHAR(20)	Symbol tested.
fill_level	SMALLINT	1 (Optimistic), 2 (Standard), 3 (Conservative).
data_split	VARCHAR(15)	"in_sample" or "out_of_sample".
config_snapshot	JSONB	Full TOML configuration used for this run (toggle states + constant values).
total_trades	INTEGER	Number of trades taken.
win_rate	DECIMAL(5,4)	0.0000 to 1.0000.
profit_factor	DECIMAL(8,4)	Gross profit / gross loss. > 1.0 = profitable.
sharpe_ratio	DECIMAL(6,3)	Annualized risk-adjusted return.
max_drawdown_pct	DECIMAL(5,4)	Largest peak-to-trough drawdown as decimal.
avg_r	DECIMAL(6,3)	Average R-multiple per trade.
validation_status	VARCHAR(15)	"unvalidated", "validated-L2", "confirmed", "overfit".
created_at	TIMESTAMPTZ	When the run completed.

3.1.3 Table: pending_recommendations

COLUMN	TYPE	DESCRIPTION
rec_id	SERIAL PK	Auto-increment ID.
instrument	VARCHAR(20)	Target instrument for the recommendation (or "global").
recommendation_type	VARCHAR(20)	"toggle_change", "constant_change", "profile_override".
proposed_change	JSONB	{"toggle": "T39", "from": true, "to": false} or {"constant": "stop_buffer_atr_mult", "from": 0.5, "to": 0.75}.
reasoning	TEXT	LLM-generated explanation with trade ID citations.
evidence_trade_ids	JSONB	["T-2024-11-02-005", "T-2024-11-05-012", ...]. Trade IDs supporting the recommendation.
confidence	VARCHAR(10)	"high" (>100 trades, L3 confirmed), "medium" (50-100 trades, L2 validated), "low" (<50 trades, unvalidated).
status	VARCHAR(10)	"pending", "approved", "rejected", "applied".
created_at	TIMESTAMPTZ	When the LLM generated this recommendation.
reviewed_at	TIMESTAMPTZ	When the human reviewed it. NULL if still pending.

3.2 Text Summary Generation

The text_summary field is auto-generated by a template function after each trade closes. This summary becomes the content that gets embedded into the vector database for semantic search.

```
TEMPLATE:
"Trade {trade_id}. {direction} {instrument} ({asset_class}).
 Grade: {grade} ({total_score}/17). POI: {poi_type}
 [extreme={is_extreme}, unicorn={is_unicorn}, fresh={is_fresh}].
 Regime: {regime}. Killzone: {killzone}.
 Tape: absorption={absorption}, whale={whale_block},
 velocity={tape_velocity_pct}%.
 Result: {result_r}R ({result_pnl_dollars}).
 Exit: {exit_reason}. Duration: {duration_minutes}min.
 Key toggles: {toggles_active}.
 Phase progression: P1={phase1_hit}, P2={phase2_hit}, P3={phase3_hit}."

EXAMPLE OUTPUT:
"Trade T-2024-12-15-001. Long AAPL (equities).
 Grade: A+ (16/17). POI: REJECTION_BLOCK
 [extreme=true, unicorn=false, fresh=true].
 Regime: trending_bull. Killzone: NY_AM.
 Tape: absorption=true, whale=false, velocity=350%.
 Result: 2.8R ($342.00). Exit: structural_trail.
 Duration: 47min. Key toggles: T07,T08,T17,T36,T39,T44,T47.
 Phase progression: P1=true, P2=true, P3=false."
```

3.3 Vector Embedding Pipeline

ELEMENT	SPECIFICATION
Embedding Model	OpenAI text-embedding-3-small (1536 dimensions) or Anthropic's voyage-3 via VoyageAI. Both produce high-quality embeddings for structured financial text. The model is called via API — no local GPU required.
Storage	pgvector extension in PostgreSQL. The embedding column uses the VECTOR(1536) type. Indexed with HNSW for fast approximate nearest neighbor (ANN) search.
When Embedded	Immediately after the text_summary is generated (on trade close). The embedding is computed once and stored. It does not change unless the summary template changes.
Semantic Search Example	User query: "Find trades similar to an A+ Rejection Block that failed in high IV." LangChain embeds the query → pgvector finds the 10 nearest neighbors by cosine similarity → returns trade records with summaries for the LLM to analyze.

4. Layer E03: Knowledge Graph (Neo4j)

The Knowledge Graph maps discovered relationships between instruments, setups, regimes, toggles, and outcomes. While the Trade Logger stores raw data, the Knowledge Graph stores conclusions drawn from that data. It answers questions like: "What setup type works best for NVDA?" and "Which toggle has the most impact on BTC win rate?"

4.1 Node Types

NODE TYPE	EXAMPLES	PROPERTIES
Instrument	AAPL, ES, BTC, EUR/USD	asset_class, avg_daily_range, chop_frequency, avg_va_width, total_trades, overall_win_rate, last_updated.
SetupType	Extreme_OB, Rejection_Block, Unicorn, Gap_FVG	global_win_rate, global_avg_r, total_occurrences, best_regime, worst_regime.
Regime	Trending_Bull, Chop, High_IV, Risk_Off	avg_duration_days, frequency_pct (how often market is in this regime), best_setup, worst_setup.
Toggle	T39, T47, T41, T44	global_impact_on_win_rate, global_impact_on_sharpe, most_beneficial_instrument, least_beneficial_instrument.
Session	NY_AM, NY_PM, London, Asian	avg_range, avg_trade_count, best_setup_in_session, worst_setup_in_session.

4.2 Edge Types (Relationships)

EDGE TYPE	SPECIFICATION
PERFORMS_WELL_WITH	Direction: [Instrument] → PERFORMS_WELL_WITH → [SetupType] Created when: Win rate for this instrument+setup exceeds the global average by ≥ 1 standard deviation AND sample size ≥ 50 trades AND finding is confirmed at fill Level 3. Properties: win_rate, sample_size, avg_r, confidence ("high"/"medium"), last_updated. Example: [AAPL] → PERFORMS_WELL_WITH → [Extreme_OB] {win_rate: 0.74, sample: 87, confidence: "high"}
PERFORMS_POORLY_WITH	Direction: [Instrument] → PERFORMS_POORLY_WITH → [SetupType] Created when: Win rate is ≥ 1 SD BELOW global average AND sample ≥ 50. Example: [TSLA] → PERFORMS_POORLY_WITH → [Decisional_OB] {win_rate: 0.38, sample: 62}
OPTIMAL_TOGGLE	Direction: [Instrument] → OPTIMAL_TOGGLE → [Toggle] Created when: Enabling this toggle improves the instrument's risk-adjusted return (Sharpe) by ≥ 10% AND confirmed at L3. Properties: sharpe_with, sharpe_without, delta_pct, sample_size. Example: [BTC] → OPTIMAL_TOGGLE → [T41_Rejection] {sharpe_with: 1.42, sharpe_without: 1.18, delta: +20.3%}
THRIVES_IN	Direction: [Instrument] → THRIVES_IN → [Regime] Created when: Win rate for this instrument in this regime exceeds 65% AND sample ≥ 30. Example: [SPY] → THRIVES_IN → [Trending_Bull + NY_AM] {win_rate: 0.71, sample: 145}

STRUGGLES_IN	Direction: [Instrument] → STRUGGLES_IN → [Regime] Created when: Win rate drops below 45% in this regime AND sample ≥ 30. Example: [QQQ] → STRUGGLES_IN → [Chop] {win_rate: 0.39, sample: 54}
OPTIMAL_CONSTANT	Direction: [Instrument] → OPTIMAL_CONSTANT → [Constant Name (virtual node)] Created when: Batch Runner identifies a constant value that outperforms the default by ≥ 10% on Sharpe AND confirmed at L3. Properties: constant_name, optimal_value, default_value, sharpe_improvement. Example: [BTC] → OPTIMAL_CONSTANT → [stop_buffer_atr_mult] {optimal: 1.0, default: 0.5, sharpe_improvement: +15%}

4.3 Nightly Update Process

STEP	PROCESS
1	A scheduled job (cron, 2:00 AM UTC daily) reads all new trade records from PostgreSQL since the last update timestamp.
2	For each instrument with new trades, recalculate: win rate per setup type, win rate per regime, win rate per session, average R per toggle configuration.
3	Compare each metric to the global average. If the delta exceeds ± 1 standard deviation AND the sample size meets the minimum threshold (≥50 for edges, ≥30 for regime edges), create or update the corresponding edge in Neo4j.
4	For OPTIMAL_TOGGLE and OPTIMAL_CONSTANT edges, only create/update from backtest_runs with validation_status = "confirmed" (Level 3 out-of-sample). Never from unvalidated or overfit runs.
5	Age decay: Edges older than 180 days without new supporting evidence have their confidence downgraded. Edges older than 365 days without update are archived (not deleted — moved to a historical subgraph).

5. Layer E04: LLM RAG Interface

This is the conversational layer where you interact with the Epistemic Engine. You ask questions in natural language. The system retrieves relevant data from the Trade Logger (pgvector semantic search) and Knowledge Graph (Neo4j graph traversal), then sends both to the Claude API with a structured prompt.

5.1 RAG Pipeline Architecture

STEP	PIPELINE
1	USER QUERY. Example: "Why are we losing on QQQ but winning on SPY?"
2	QUERY ROUTER. LangChain's query routing classifies the intent: (a) similarity search ("find trades like X"), (b) graph query ("what works for instrument Y"), (c) comparison ("X vs. Y"), (d) recommendation ("what should we change"). Most queries trigger BOTH vector and graph retrieval.
3a	VECTOR RETRIEVAL. Embed the query. Search pgvector for the 20 most similar trade summaries. Filter by instrument if specified. Return the trade records with their text_summary fields.
3b	GRAPH RETRIEVAL. Convert the query to a Cypher query. For the example: MATCH (i:Instrument)-[r]-(t) WHERE i.name IN ['QQQ','SPY'] RETURN i, r, t. Returns all edges (PERFORMS_WELL_WITH, STRUGGLES_IN, etc.) for both instruments.
4	CONTEXT ASSEMBLY. Combine the vector results and graph results into a structured context block. Add the system prompt (see Section 5.2). Add the user's question.
5	LLM CALL. Send the assembled prompt to the Claude API (claude-sonnet-4-20250514). Temperature = 0.1 (we want factual, grounded responses, not creative ones).
6	RESPONSE PARSING. The LLM returns a structured response with: answer (plain language), cited_trades (list of trade IDs), confidence (based on sample sizes in the evidence), and optionally proposed_toml (a TOML snippet for a recommended change).
7	If proposed_toml is present, it is inserted into pending_recommendations for human review. The LLM NEVER directly modifies the bot configuration.

5.2 System Prompt Template

This is the system prompt sent to the Claude API with every query. It defines the LLM's role, constraints, and output format.

```
SYSTEM PROMPT:

You are the Epistemic Engine analyst for the FLOF Matrix trading system.

YOUR ROLE:
- Analyze trade data to find patterns and make recommendations.
- Every claim must cite specific trade IDs or backtest run IDs.
- If sample size < 50 trades, state this and mark confidence as LOW.
- Never speculate beyond what the data shows.
- Never recommend changes validated only at Fill Level 1 (optimistic).

AVAILABLE CONTEXT:
```

```
- VECTOR_RESULTS: Similar trades from the vector database.
- GRAPH_RESULTS: Relationships from the Knowledge Graph.
- BACKTEST_RUNS: Aggregate metrics from relevant backtest experiments.

OUTPUT FORMAT (always use this structure):
{
  "answer": "Plain language analysis...",
  "cited_trades": ["T-2024-12-15-001", "T-2024-12-18-003"],
  "cited_runs": ["RUN-EXP001-AAPL-L3-042"],
  "confidence": "high|medium|low",
  "sample_size": 87,
  "proposed_toml": null | "# Optional TOML recommendation\n..."
}

CONFIDENCE LEVELS:
- HIGH: >= 100 trades, Level 3 confirmed, out-of-sample validated.
- MEDIUM: 50-100 trades, Level 2 validated.
- LOW: < 50 trades, or Level 1 only, or live-only data (no backtest).
```

5.3 Query Prompt Templates

These are pre-built prompt templates for common analytical questions. Each template automatically constructs the right combination of vector and graph retrievals.

TEMPLATE NAME	USER TRIGGER	RETRIEVAL STRATEGY
instrument_comparison	*"Why does X win but Y lose?"*	Graph: all edges for both instruments. Vector: 10 losing trades from Y, 10 winning trades from X. Compare patterns.
setup_analysis	*"How does setup Z perform on instrument X?"*	Graph: PERFORMS_WELL/POORLY edges for X+Z. Vector: all trades matching X+Z. Backtest_runs: experiments that tested Z on X.
toggle_impact	*"Should I enable T39 for AAPL?"*	Graph: OPTIMAL_TOGGLE edge for AAPL+T39. Backtest_runs: runs with T39 on vs. off for AAPL. Compare Sharpe, win rate, profit factor.
constant_optimization	*"What's the best stop ATR mult for BTC?"*	Graph: OPTIMAL_CONSTANT edge for BTC + stop_buffer_atr_mult. Backtest_runs: experiments varying this constant for BTC.
regime_analysis	*"When should we avoid trading QQQ?"*	Graph: STRUGGLES_IN edges for QQQ. Vector: losing trades for QQQ grouped by regime. Identify regime(s) with worst performance.
failure_forensics	*"What do our A+ losses have in common?"*	Vector: all A+ trades with result_r < 0. Cluster by regime, session, poi_type, tape_conditions. Report common failure modes.
session_performance	*"Which session is best for ES?"*	Graph: THRIVES_IN edges for ES filtered by Session nodes. Trades: group by killzone and compare win rate, avg R.
portfolio_health	*"How has overall performance been?"*	Trades: last 30 days, all instruments. Aggregate: win rate, profit factor, avg R, drawdown. Compare to previous 30-day window. Trend analysis.
recommendation_batch	*"Generate this week's TOML updates."*	Graph: all edges updated in last 7 days. For each instrument with new findings, generate a sub-profile recommendation. Output as pending_recommendations.
similar_trade_search	*"Find trades like this one" (with trade_id)*	Vector: embed the referenced trade's summary, find 10 nearest neighbors. Compare outcomes. Useful for checking if a specific trade pattern is consistently profitable.
exit_analysis	*"Are we exiting too early on NVDA?"*	Trades: all NVDA trades. Compute: avg result_r by exit_reason. If structural_trail exits avg 1.5R but phase1_target exits avg 2.0R, the runner is underperforming the partial — suggesting trail is too tight.
custom_query	*Any natural language question*	The LLM uses tool-use to decide which combination of vector search, graph traversal, and SQL queries best answers the question. Most flexible but least structured.

6. Instrument-Specific Dynamic TOML Profiling

This is the highest-value output of the Epistemic Engine: per-instrument TOML sub-profiles that tune the bot's behavior based on validated Knowledge Graph findings.

6.1 TOML Loading Hierarchy

ORDER	FILE	DESCRIPTION
1	**flof_base.toml**	All 50 toggles at defaults. All constants at universal defaults. This is the foundation.
2	**profile_{asset_class}.toml**	Asset class overrides. Example: profile_equities.toml sets T44=ON, T45=ON, T49=ON. Only delta values specified.
3	**constants.{INSTRUMENT}.toml**	NEW. Instrument-specific overrides generated by the Epistemic Engine. Example: constants.NVDA.toml sets T41=OFF, phase1_a_plus_scaleout=0.20. Only delta values from the asset class profile.

Each layer only specifies values that differ from the previous layer. An instrument sub-profile might contain only 3–5 overrides. The bot merges all three layers at startup: base → asset_class → instrument. The last writer wins.

6.2 Example Sub-Profile: NVDA

```
# constants.NVDA.toml
# Generated by Epistemic Engine on 2025-03-22
# Based on: 142 backtest trades (L3 confirmed) + 38 paper trades
# Confidence: HIGH

# NVDA is high-momentum, low-chop. Extreme OBs outperform.
# Rejection Blocks underperform (51% vs 67% global avg).

[toggles]
T41_rejection_blocks = false  # Underperforms on NVDA (-16% vs avg)

[constants]
phase1_a_plus_scaleout = 0.20  # Let runners ride (NVDA trends hard)
phase1_a_plus_target_r = 2.5   # Higher R target (momentum supports it)
stop_buffer_atr_mult   = 0.75  # Wider stops (NVDA wicks are large)
toxicity_dead_tape_min = 3     # Shorter dead tape window (NVDA moves fast)
```

6.3 Example Sub-Profile: KO (Coca-Cola)

```
# constants.KO.toml
# Generated by Epistemic Engine on 2025-03-22
# Based on: 98 backtest trades (L3 confirmed) + 22 paper trades
# Confidence: HIGH
```

```
# KO is mean-reverting, heavy chop. Directional trades underperform.
# Iron Condors via T50 are highly profitable (78% win rate).

[toggles]
T11_fast_move_switch   = false  # KO never moves fast enough to justify
T47_dynamic_scaleouts  = false  # Use rigid 50% (runners don't extend on KO)
T50_chop_to_condor     = true   # KO's chop is consistent and profitable

[constants]
chop_va_width_threshold = 1.2   # Tighter chop detection (KO ranges are narrow)
condor_profit_target    = 0.60  # Take profit at 60% (KO reverts reliably)
```

6.4 Weekly Profiling Cycle

STEP	PROCESS
1	Every Saturday at 6:00 AM UTC, the Epistemic Engine runs the recommendation_batch template for all actively traded instruments.
2	For each instrument, it queries the Knowledge Graph for edges updated in the last 90 days. It synthesizes these into a behavioral profile.
3	If the profile suggests any changes to the current sub-profile (or if no sub-profile exists yet), it generates a proposed constants.{INSTRUMENT}.toml and inserts it into pending_recommendations.
4	On Saturday/Sunday, the human reviews all pending recommendations. Each one includes: the proposed TOML changes, the reasoning with trade citations, the confidence level, and a comparison to the current configuration.
5	Approved recommendations are applied. The bot loads the new sub-profiles at Monday’s session start. Rejected recommendations are logged with the human’s reason (this feedback trains the engine to make better recommendations over time).

7. Function Specifications

FUNCTION	INPUTS	OUTPUTS	DESCRIPTION	LAYER
run_experiment()	manifest: ExperimentManifest	list[RunResult]	Master function for E01. Parses manifest, generates TOML permutations, launches parallel backtests, collects results. Returns all RunResult objects.	E01
generate_toml_permutations()	manifest: ExperimentManifest base_toml: dict	list[TempTOML]	Calculates the Cartesian product of all toggle/constant variations and generates one temporary TOML per combination.	E01
progressive_filter()	results: list[RunResult] top_pct: float	list[RunResult]	Ranks results by profit_factor, returns the top N%. Used between fill level stages.	E01
log_trade()	event: TradeEvent	trade_id: str	Writes a trade record to PostgreSQL. Auto-generates text_summary, calls the embedding API, stores the vector. Core E02 function.	E02
embed_summary()	text: str	vector: list[float]	Calls the embedding model API (OpenAI or VoyageAI). Returns a 1536-dimension vector.	E02
semantic_search()	query: str top_k: int filters: dict	list[TradeRecord]	Embeds the query, performs ANN search on pgvector, applies optional filters (instrument, source, grade), returns top_k results.	E02
update_knowledge_graph()	new_trades: list[TradeRecord]	edges_created: int, edges_updated: int	Nightly batch job. Computes aggregate stats per instrument/setup/regime, creates or updates Neo4j edges when statistical thresholds are met.	E03
query_graph()	cypher: str	list[GraphResult]	Executes a Cypher query against Neo4j and returns results. Used by E04 for graph retrieval.	E03
rag_query()	user_question: str template: str	AnalysisResponse	Master function for E04. Routes the query, performs vector + graph retrieval, assembles	E04

			context, calls Claude API, parses response.	
generate_recommendation()	instrument: str graph_edges: list trade_data: list	PendingRecommendation	Called by the weekly profiling cycle. Synthesizes graph findings into a TOML sub-profile recommendation.	E04

8. Technology Stack Summary

COMPONENT	TECHNOLOGY	RATIONALE
Backtesting Engine	NautilusTrader (Python/Rust)	Already the core execution engine. Supports headless mode for batch runs. Rust backend handles tick processing; Python strategies define logic.
Trade Database	PostgreSQL 16+ with pgvector	Production-grade relational DB. pgvector extension adds native vector similarity search without needing a separate vector DB. Single database for structured data + embeddings.
Knowledge Graph	Neo4j Community Edition	Best-in-class graph database. Cypher query language is intuitive for relationship queries. Free community edition sufficient for this workload.
Embedding Model	OpenAI text-embedding-3-small or VoyageAI voyage-3	High-quality 1536-dim embeddings via API. No local GPU needed. Cost: ~$0.02 per 1M tokens (negligible at trade-log volumes).
LLM	Claude (Anthropic API)	Strong analytical reasoning. Excellent at synthesizing structured data with citations. Temperature 0.1 for factual responses.
Orchestration	LangChain	Manages the RAG pipeline: query routing, vector retrieval, graph retrieval, context assembly, LLM calls, response parsing. Well-supported Python ecosystem.
Batch Processing	Python multiprocessing	Standard library. No external dependency. Worker pool with configurable count. Sufficient for hundreds of backtests in parallel.
Job Scheduling	cron (Linux) or APScheduler	Nightly Knowledge Graph updates and weekly profiling cycles. Simple, reliable.

WHAT THE EPISTEMIC ENGINE DOES NOT DO

- It does NOT trade. It has no connection to any exchange or broker. It observes and recommends.
- It does NOT autonomously change the bot's configuration. Every recommendation requires human approval.
- It does NOT guarantee profitability. It identifies statistically validated patterns. Markets change; past performance does not guarantee future results.
- It does NOT replace your judgment. It is a research assistant that is grounded in YOUR data, not general market opinions.

E N D O F D O C U M E N T

FLOF Matrix — Epistemic Engine Technical Blueprint v3.5

Read in conjunction with all prior FLOF Matrix documents (v1.0 through v3.4).