ChatGPT

Background & Scope

System Overview: *Ensemble 1.0* is a multi-model crypto *BTC-PERP* trading bot running on multiple timeframes (5m primary; 15m, 1h overlays). It uses a meta-learning ensemble (stacked ML models + multi-armed bandit) to combine **cohort-based signals** ("pros" vs "amateurs" order flow) with **model predictions** in real-time. The goal is high Sharpe, low drawdown trading, using **paper-trading** infrastructure for backtests and simulation. The system includes separate bot instances for 5m, 1h, 12h, 24h intervals (with similar code structure) and an emerging unified overlay framework to integrate signals across timeframes.

Architecture in Brief: The 5m bot fetches market data (candles, order book) via exchange APIs (Binance Testnet or Hyperliquid), computes engineered features over a rolling window (1000 bars by default), then uses trained ML models to predict the next move. It outputs a *confidence* (probability) and *alpha* (edge magnitude) for upward vs downward moves. These model signals are combined with *cohort signals* derived from recent trade flow (attempting to gauge "pro" vs "amateur" sentiment) and possibly higher-timeframe overlay signals. A **Thompson Sampling bandit** then selects one of four arms: **pros**, **amateurs**, **model_meta**, **model_bma** (two model variants). The chosen arm's signal and direction form the *trade decision*. Before execution, risk management applies position sizing and guardrails. Finally, events are logged for analysis. All sensitive credentials (exchange API keys, service account keys) are currently in config files – an **operational risk** that we address in controls.

Scope: This report analyzes the current system ("As-Is") and proposes enhancements to **improve risk-adjusted performance** (higher Sharpe, lower drawdown) and reduce tail/operational risks. We provide: (1) diagnosis of *low-confidence trades* that hurt performance, (2) specific fixes (e.g. better calibration, dynamic thresholds, stricter gating), (3) a roadmap to incorporate **multi-timeframe signals** (15m…1w) systematically, and (4) integration of an LLM-based "Copilot" for monitoring and analysis. We focus on *evidence-backed observations* from the code and logs, then recommend design changes with clear testing and control steps. No promises of profit – only a safer, more robust strategy execution.

Business & Trading Goals

Primary Trading Goal: Achieve **consistent risk-adjusted returns** in BTC perpetual futures via an automated strategy. Key performance targets include Sharpe ratio improvement (e.g. >1.5), drawdown reduction (e.g. <10% max decline), and controlled trade frequency (avoiding over-trading in low edge scenarios). The system should exploit short-term mispricings with minimal latency while staying aligned with higher timeframe trends to avoid large directional errors.

Risk Management Goals: Limit **tail risk** and ensure capital preservation. This means enforcing stop-loss-like behavior (via position sizing and trade vetoes on extreme signals), avoiding trades when predictive confidence is low or market conditions are outside the model's training distribution, and continuously monitoring system health (latency, data quality, PnL swings). Operationally, prevent failures – e.g. by handling exchange errors gracefully and using health checks.

Business Constraints: The strategy runs **24/7** on crypto markets, so reliability and monitoring are crucial – downtime or missed signals can mean lost opportunity or risk exposure. The trading system must adapt to varying volatility regimes (e.g. quiet vs volatile days) – possibly by scaling risk or pausing in extreme conditions – to avoid large losses. Regulatory/compliance is less of an issue (prop trading on own account), but maintaining API key security and proper audit logs is important for business continuity and review.

Performance Measurement: Success is measured by **risk-adjusted metrics** (e.g. Sharpe, Sortino), not just raw return. We specifically want to improve metrics like *return per unit risk* and *consistency* of returns. For example, if currently Sharpe is ~0.8 with 20% drawdown, the goal after improvements might be Sharpe >1.2 with <10% drawdown, achieved by cutting out low-quality trades and better sizing. Additionally, **operational success** is measured by system uptime, timely alerts on issues, and quality of log data (for post-trade analysis and model retraining).

User Personas

- **Quant Strategist (Owner A):** Develops and tunes the trading algorithm. Cares about model accuracy, feature engineering, and ensuring the ensemble logic improves PnL. Needs detailed logs and analytics to diagnose model behavior (e.g. why a low-confidence trade was made) and wants an ML Copilot to help analyze performance. This user will drive calibration improvements and threshold tuning.

- **Trading Operations Engineer (Owner B):** Manages deployment, monitoring, and multi-timeframe integration. Cares about system stability, log pipelines, and risk controls (kill-switches, guardrails). Needs real-time health metrics and alerting when something goes wrong (e.g. data feed stops or risk limits hit). Will implement multi-timeframe data flows and ensure secrets/infrastructure are handled safely.

- **Portfolio Manager / Stakeholder:** Oversees the trading bot’s performance at a high level. Interested in summaries: daily PnL, trade win/loss stats, and assurance that risk is controlled. They benefit from the LLM Copilot producing easy-to-read summaries of the day’s trades and notable events. They are less in the weeds of code, but rely on the team (Quant and Ops) to make improvements that reflect in better performance and lower risk.

- **Compliance/DevOps Auditor:** (If applicable) Ensures that the system follows security best practices (no hard-coded keys, proper logging for audit). They will check that API keys and secrets are not leaked (e.g. currently keys are stored in plain JSON configs – this person advocates moving them to a secure vault and rotating them). They also ensure an incident log and recovery plan exists (e.g. if the bot goes rogue or crashes).

Current System Overview (As-Is)

4.1 System Entrypoints & Deployment

Code Structure: The repository is organized by timeframe. For example, `live_demo/` (5m bot) and `live_demo_1h/`, `live_demo_12h/`, `live_demo_24h/` each contain similar modules (`main.py`, `features.py`, `model_runtime.py`, `risk_and_exec.py`, etc.) for that timeframe. There are Windows batch scripts (e.g. `start_5m.bat`) and Python launchers (`run_5m_debug.py`, `run_1h.py`, etc.) that set up paths and call the appropriate `main.run_live()`. In production, each bot can run in its own process or container. A unified launcher `run_unified_bots.py` exists to run all timeframes concurrently via `asyncio`, indicating plans for integrated deployment.

Entrypoints: For the 5m strategy, the main loop starts in `live_demo/main.py -> run_live()`. This loads the config, initializes data sources, then enters the trading loop. Each timeframe's `main.py` is similar. The *FastAPI backend* (`backend/main.py`) is a separate entrypoint that serves a dashboard for monitoring (exposing logs, metrics via HTTP). This suggests the trading bots output data to files/DB which the dashboard reads. In practice, one might start the 5m bot and others via scripts or systemd services, and separately run the FastAPI server and front-end (React app in `frontend/`) for status.

Configuration: Config files (e.g. `live_demo/config.json` for 5m, and corresponding for 1h, 12h, 24h) define runtime settings: data source, symbol, interval, warmup length, overlay timeframes, and exchange API keys. For example, in 5m config, `symbol: "BTCUSDT"`, `interval: "5m"`, `overlays: ["15m","1h"]` are set, meaning the 5m bot should incorporate 15m and 1h signals. Exchange credentials (Binance Testnet key/secret) are **present in plain text** in these configs. There is also a `production_config.json` for additional settings like log paths, alert channels, etc., and a `.env.example` (with placeholders). Environment variables used include `STRATEGY_ID` (defaults to `"ensemble_1_0"` if not set) and `SCHEMA_VERSION` (defaults to `"v1"`). These tag each log entry with strategy metadata for compatibility. The code can also read a JSON blob from `STRATEGY_CONFIG` env var to override config values, although no *ensemble_config.json* was found – implying defaults are used.

Directory Tree (Top 2 Levels): Below is a simplified view of the repository structure (directories **bold**, files bullet-listed):

• **backend/** – *(FastAPI server)*
• main.py – *API for dashboard (serves logs, metrics)*

• start_api.sh/.bat – *scripts to run backend*

• **core/**

• config.py – *Strategy ID, schema version helpers (reads env)*

• **live_demo/** – *(5m trading bot code)*

• main.py – *Main loop (async)*
• config.json – *Primary config (5m settings, API keys)*
• features.py – *Feature engineering pipeline*
• model_runtime.py – *Load model, make predictions*
• decision.py – *Decision logic (thresholds, gating, bandit selection)*
• bandit.py – *Thompson sampling bandit implementation*
• risk_and_exec.py – *Risk management & execution logic*
• **ops/** – *Observability utils* (log emitter, router, heartbeat)

- **alerts/** – *Alerting (Slack, etc.)*
- **config/** – production_config.json (logging, alert settings)
- **integration/** – production_integration.py (integration test)
- **schemas/** – JSON schemas for logs
- **tests/** – test files for key components
  (Other subdirs like security/, tools/, scripts/ omitted for brevity.)

- **live_demo_1h/**, **live_demo_12h/**, **live_demo_24h/** – *(similar structure for other timeframes)*

- **plans/** – Markdown docs for various analyses and fixes (e.g. *5M_BOT_FINAL_ANALYSIS.md*, *timeframe_trading_fixes.md*)

- **scripts/** – Utility scripts (backtest data aggregation, calibration checks, etc.)

- **tools/** – Developer tools (context builders, diagnostic scripts, etc.)

- *(Various .md reports and .py analysis scripts at root, capturing previous investigations into model performance.)*

This structure shows that the system was developed to be extended across multiple timeframes (separate code for each, plus attempts at unification). The *Ensemble 1.0* moniker appears in Strategy ID and internal docs, implying the current live version.

Deployment & Invocation: Typically, the ops team would start each bot via the provided scripts. For example, running `python run_5m_debug.py` will set up paths and call `live_demo.main.run_live("live_demo/config.json")`. On production servers, `start_5m.ps1` or `.bat` might be scheduled. The unified runner `run_unified_bots.py` suggests an alternative: run 5m, 1h, 12h, 24h bots concurrently inside one process (useful for synchronous overlays). This might be used in paper-trading or dev mode to simplify coordination. The FastAPI backend likely runs alongside (maybe on a different port) to serve the React frontend, which displays bot status (open position, recent trades, equity curve, etc.).

4.2 Data Flow & Decision Cycle (5m Bot)

Data Ingestion: On startup, `run_live` loads the config and connects to the exchange API. If `exchanges.active` is `"hyperliquid"`, it uses a custom adapter to fetch candles via HTTP from Hyperliquid's API. If active is `"binance"` or `"testnet"`, it uses the Binance client (via `python-binance` futures API). In code, we see that for Binance the bot initializes a `PBClient` (probably Python-Binance client) with `api_key` and `api_secret` from config, then wraps it in a `UMFuturesAdapter` to have a unified interface. A `MarketData` object is created to provide convenient methods for getting latest price and historical klines. This class attempts cached reads and rate-limit handling for efficiency. **Warmup:** The bot fetches `warmup_bars` (e.g. 1000) past 5m candles on launch to compute indicators/features before trading. These bars are stored likely as a Pandas DataFrame.

Feature Engineering: The 5m bot computes a variety of features each bar. The code includes a `FeaturePipeline` class for scaling/normalizing features (with mean/std from training). The actual feature computation likely happens in `model_runtime.py` or `features.py`. Although not explicitly listed in code snippet, given the presence of files like `feature_columns_*.json` (which list feature names) and `LiveFeatureComputer` (mentioned in `unified_overlay_system.py`), we infer that **technical indicators** (momentum, volatility, etc.) and possibly **cohort features** are computed. For example, "mom_1" might be 1-bar momentum, etc. Each bar, the bot will update the feature vector using the latest candle (and maybe order book or funding rate from `funding_hl.py`). The cohort features are fed by a *Hyperliquid real-time listener*: `hyperliquid_listener.py` (and similarly for Binance) likely streams trades to update internal stats of "pros vs amateurs". Indeed, a `CohortState` dataclass holds rolling metrics like `pros`, `amateurs`, `mood`. As fills come in (with fields: address, side, size), an update function classifies them (e.g. addresses might be tagged as pro or not) and updates these signals. The net effect: each bar, we have *S_top* (pros sentiment), *S_bot* (amateurs sentiment), and *mood* (perhaps difference or a composite) as inputs.

Model Prediction: The primary model is a meta-classifier ensemble. The code loads model artifacts (pickled via joblib) in `ModelRuntime` – likely `EnhancedMetaClassifier` which ensembles base models. Two outputs are produced for the model's prediction on the current features: - `p_up` / `p_down`: predicted probabilities of market going up or down in the next interval. From these we derive **confidence** = max(p_up, p_down) and **alpha** = |p_up – p_down|. For example, if p_up=0.70, p_down=0.30, then confidence=0.70 and alpha=0.40. - `s_model_meta` and `s_model_bma`: Two signal scores. *s_model_meta* is the meta-model's raw score (after calibration); *s_model_bma* is a second model's score – possibly from a Bayesian Model Averaging approach or an alternate ensemble method. If only one model is active, the code defaults s_model_bma = s_model_meta for consistency. These scores are scaled roughly in [-1,1] where sign indicates long/short bias and magnitude is strength. (The code flips these if needed based on config flags like `flip_model` to ensure consistent sign conventions.)

The model predictions are calibrated. A custom isotonic regression calibrator is defined (CustomClassificationCalibrator) to ensure probabilities align with actual win rates. It's likely applied within the model (the model object might output already-calibrated p_up). We see evidence of Platt scaling intercepts `a, b` in calibration logs.

Decision Logic & Ensemble: Once we have cohort signals (S_top, S_bot, S_mood) and model outputs (p_up, p_down, s_model_meta, s_model_bma), the **Ensemble Combiner** logic in `decision.py` kicks in: 1. **Threshold Gating:** The code defines thresholds in a `Thresholds` dataclass (defaults: S_MIN=0.12, CONF_MIN=0.60, ALPHA_MIN=0.10, etc.). These are critical: *S_MIN* (0.12) is the minimum absolute cohort signal to consider it significant; *CONF_MIN* (0.60) is the minimum model probability confidence to consider model signals; *ALPHA_MIN* (0.10) is minimum probability difference. Using these, the function `compute_signals_and_eligibility` assembles a unified view: - *signals*: e.g. `{'S_top': 0.5, 'S_bot': -0.2, 'S_mood': 0.4, 'S_model_meta': 0.3, 'S_model_bma': 0.3}` (example values). - *eligible*: whether each arm is eligible: `pros = |S_top|>=0.12`, `amateurs = |S_bot| >=0.12`, `model_meta = (conf_model>=0.60 and alpha_model>=0.10)`, similarly for `model_bma`. This means if the model is not at least 60% confident and with 0.1 edge, **both model-based arms are turned off** – preventing low-confidence model trades. - *side_eps_vec*: a tuple of thresholds per arm (for later use): (S_MIN, S_MIN, ALPHA_MIN, ALPHA_MIN). Essentially, cohort arms require 0.12 signal to act; model arms require 0.10 alpha. - *extras*: e.g. `{'conf_model': 0.72, 'alpha_model': 0.44}` (the model's current confidence and alpha).

1. **Multi-Arm Bandit Selection:** The system uses a **Thompson Sampling bandit** (`SimpleThompsonBandit`) to adaptively choose which signal source to trust for each trade. The bandit maintains internal state (counts, means, variances for each arm's performance) and samples a random "score" for each arm proportional to its estimated success. Arms correspond to:
2. **Arm0:** "pros" (S_top) – upper cohort (presumably professional traders' net position).
3. **Arm1:** "amateurs" (S_bot) – lower cohort.
4. **Arm2:** "model_meta" – ML meta-model.
5. **Arm3:** "model_bma" – ML BMA or backup model. The decision logic `decide_bandit()` receives the signals, eligible mask, and bandit state. It first checks if any arm is eligible; if none, it returns a no-trade decision. If at least one is eligible, it either:
6. With probability *ε* (exploration rate, likely very low or 0 in production), choose a random eligible arm.
7. Otherwise, for exploitation, **sample from the bandit's posterior** via `bandit.select()` to pick the arm with highest sampled reward. There is a "model_optimism" parameter that can temporarily boost the model arms' means before selection – effectively giving a slight bias to choose model if it's roughly as good as others (this is a tuning knob to encourage using ML predictions). The chosen arm index yields `chosen_arm` (e.g. 'pros' or 'model_meta'). The raw signal value for that arm is taken (`raw_val = signals[S_xxx]`). Then the code enforces the *side epsilon* threshold: if |raw_val| < threshold for that arm, it will override to **no trade** (dir=0). If |raw_val| ≥ threshold, it sets `direction = sign(raw_val)` (either +1 buy or -1 sell), and sets **alpha** = 0 if below threshold, else presumably uses the model's confidence/alpha for model arms or the signal magnitude for cohorts. (In code, for simplicity, they might just output an alpha of 1.0 for cohort arms since those signals aren't probabilities. The actual snippet suggests `alpha = 0.0 if under threshold, else something like abs(raw_val)` – but effectively, *extras['alpha_model']* is used for model arms). The result is a `decision` dict like: `{'dir': 1, 'alpha': 0.5, 'details': { 'chosen': 'model_meta', 'eligible': {...}, 'signals': {...} }}`.

8. **Risk Management & Execution:** The `risk_and_exec.RiskConfig` defines parameters for position sizing. Key ones: `sigma_target=0.20` (target annualized volatility for the strategy's returns), `pos_max=1.0` (max position as fraction of some base notional), `cooldown_bars=1` (after a trade, skip 1 bar before another trade in same direction), and `base_notional=5000.0` (reference USD size for position calculations). Once a trade decision (dir and alpha) is made:

9. The system computes the **target position size** via `target_position(direction, alpha)`. This uses the realized volatility of recent returns to scale the position such that if the signal (alpha) is strong, we take a larger position, but if market volatility (rv) is high, we scale down. Specifically: `pos = (sigma_target / rv) * alpha`, then clamp to ±pos_max. For example, if current realized vol is equal to target (20%), then pos = alpha (so a 0.5 alpha => 50% position). If volatility is higher than target, position is reduced. They also have an optional *volatility guard* that further scales down alpha when rv > `vol_guard_sigma` (e.g. >0.8) to a minimum fraction (25%). By default `vol_guard_enable` is False, but enabling it would shrink positions in very volatile regimes – a safety we might consider turning on.
10. **Pre-Trade Guards:** Before sending an order, `risk_and_exec.evaluate_pretrade_guards()` is likely called. This might check things like: is our current position already in that direction (to prevent

doubling down too fast), is ADV (average daily volume) usage within cap, etc. The config has `adv_cap_pct=0.0` (off) and possibly other guards. If a guard triggers, the trade can be vetoed. In logs, a *rejection* event would be emitted with reason (e.g. "Cool-down period, skipping trade").

11. **Execution:** If proceeding, an order is constructed. In dry_run mode, they don't actually hit the exchange; instead they simulate the fill (`_simulate_trade()` in code for backtest mode) and log the result. If not dry_run (live trading), the code calls the exchange client's order method. For Binance, `UMFuturesAdapter` wraps the actual REST call. For Hyperliquid, the code currently has `new_order()` as a placeholder returning a dummy {"status": "dry_run"} – meaning live trading on Hyperliquid may not be fully implemented yet. In either case, an **order_intent** event is logged before execution (audit trail of the decision parameters) and an **execution** event is logged after (with order status, fills, etc.).
12. **Post-Execution:** After an execution (real or simulated), the system updates internal state: current position, last trade time, and importantly, updates the bandit's reward. The code likely measures the PnL of the trade after a certain horizon (maybe one bar or until position closes) and calls `bandit.update(chosen_arm, reward)`. Though we did not find an explicit call in the snippet, the presence of `runtime_bandit.json` state and the design intent imply that each trade's outcome (perhaps categorized as win vs loss, or scaled by return) is fed back to the bandit. This will adjust the means/variances of that arm – over time, arms that lead to profitable trades get higher scores, and ones with losses get lower. (Open Question: the exact reward function and update timing are not clearly present – possibly a simplification where reward = trade PnL in basis points, updated when a trade closes).
13. Additionally, PnL attribution is tracked: `pnl_attribution.py` accumulates how much PnL comes from "alpha" (directional edge) vs "timing" vs fees vs slippage. This is used for strategy diagnostics but not directly in decision-making.

Logging & Observability: The system has a rich logging mechanism using JSONL event files: - A global `LogEmitter` is initialized in each bot (e.g. in 5m main, `get_emitter('5m', base_dir=.../logs)` is called). Logs are written under `paper_trading_outputs/logs/<topic>/date=YYYY-MM-DD/<topic>.jsonl`. The emitter sanitizes sensitive fields (replacing actual API keys with `<REDACTED>` in logs). - Key event types (with emitter methods) include: **signals**, **ensemble**, **order_intent**, **execution**, **health**, **costs**, **fill**, **feature_log**, **calibration**, **rejection**. For example, after each decision, `emit_signals` writes a record containing the latest features, model outputs, decision details, and cohort info. An **ensemble** event may be emitted when combining multi-timeframe signals (predictions from each timeframe plus the combined result) – this would be relevant when overlay logic is active. When an order is sent, `emit_order_intent` logs its parameters; after fill or simulation, `emit_execution` logs execution outcome and risk state (position, etc.). A periodic `emit_health` may log aggregated stats like recent win rate, average confidence, etc., for monitoring. - These logs feed both the dashboard and offline analysis. The provided logs indicate daily aggregated JSON files (e.g. `gates.json`, `turnover_report.json`) were produced by analysis scripts, confirming that logs are used to compute performance metrics (trade counts, turnover, etc.).

4.3 Known Issues & Recent Changes

From code comments and the numerous markdown reports in the repo, a few known issues in the current system can be summarized: - **Low-Confidence Trades:** The team identified that some trades with marginal model confidence have hurt performance (e.g. false signals triggered by small "pros/amateurs" fluctuations). Despite `CONF_MIN=0.60`, there are scenarios where non-model arms took trades while the model was neutral, resulting in losses. We detail this in Section 9.1. - **Inconsistent Multi-Timeframe Integration:** The config indicates 5m uses 15m and 1h overlays, and code for overlay combination exists (e.g. `EnhancedSignalCombiner` with rules like AGREEMENT, MAJORITY_VOTE). However, it's unclear if this is fully active in Ensemble 1.0 or partially implemented. The separate 1h/12h bots might not yet feed into the 5m's decisions except via offline combination. There's a **Unified Overlay System** module (not yet in production) aiming to merge all into one process. - **Secrets Management:** API keys for exchanges and a Google service account JSON (for Sheets logging) are stored in the repo (e.g. `live_demo/config.json` and `metastackerbandit1-****.json`). This is a security risk if not rotated; it's supposed to be mitigated by sanitization in logs, but ideally these should not live in plaintext in source. - **Logging Volume & Performance:** Writing JSONL each bar/trade can be heavy. There is logic for log rotation and compression (compress old logs, limit size) in `stdout_json_logger.py` (not shown above, but referenced). The team must ensure logging doesn't lag the trading loop. There is also mention of sampling (perhaps not all feature logs are kept) to control volume. - **Bandit Reward Calibration:** Ensuring the bandit rewards reflect true trading performance is tricky. If reward attribution is delayed or mis-specified, the bandit may not converge to the best arm. This might explain oscillations where sometimes the bot follows cohort signals even if model has been more accurate, or vice versa. We suspect further tuning or even simplifying to a deterministic combiner (like requiring consensus between model and mood) might yield better results initially.

These issues set the stage for our improvements: we aim to tighten the confidence gating and possibly use multi-timeframe confirmation to filter out bad trades, improve calibration of probabilities, and formalize the multi-timeframe signal integration with a robust design.

Functional Requirements: Copilot/LLM Integration

We propose integrating an **LLM-based "Copilot"** to assist both developers and operators in interacting with the trading system. This Copilot has two main use-cases:

1. **Analyst Copilot for Logs & Performance:** Using the rich logs already produced, an LLM (like GPT-4) can summarize trading sessions and diagnose issues. For example, at end of day, it could read the `signals.jsonl` and `executions.jsonl` events and produce a plain-language report: "**Today the bot made 12 trades, with 8 wins and 4 losses. Most profitable trade was a long at 10:35, closed 5m later for +0.5%. Two losing trades around 14:00 occurred when model confidence was borderline (61%), suggesting potential overtrading.**" This helps the team quickly understand what happened without parsing raw data. We'll implement an **LLM Summary Emitter** that triggers after each session or on-demand, collates recent stats (perhaps using the `pnl_attribution` and `gates` reports) and feeds it to an LLM API to generate a summary. The output can be posted to Slack or stored as a markdown report.

2. **Interactive Troubleshooting Copilot:** Expose a chat interface (possibly through the FastAPI backend) where developers can ask questions like "Why was the trade at 14:00 short if the 1h trend was up?" The Copilot would fetch relevant log snippets: the decision event for that timestamp, the overlay signals (1h said maybe neutral or slightly down), and the outcome. It would then answer: e.g. "**At 14:00, the bot went Short because the short-term model signal and 'amateurs' flow were strongly negative (model p_down=0.62) despite the 1h trend being up. The 1h overlay was not strong enough to veto the trade under current rules. That trade resulted in a loss, suggesting**

the need for stricter trend filter." This provides immediate insight and could even suggest which parameters to adjust. This feature can be built by vectorizing log events and using GPT with relevant context.

Requirements for Copilot Integration: - *Data Access:* The Copilot needs access to logs and maybe model configs. We ensure that log data is organized (perhaps summarized as JSON as it is now) for easy consumption. No sensitive info should be sent to the LLM (keys are redacted anyway). - *Real-time Use:* For the operator, the Copilot should help monitor. For example, an alert can trigger: "Confidence dropped below 50% on recent signals, but trades still happening – please review." The LLM can generate suggested actions ("Consider pausing trading or raising CONF_MIN threshold."). - *Security & Cost:* We must control what data goes to an external API. Likely, only non-sensitive aggregates. We will not feed raw order books or PII – only strategy metrics. - *Integration Points:* We can integrate the Copilot in the **FastAPI backend** – adding endpoints that run analyses and return LLM answers. Possibly a UI element in the React frontend ("Ask the Bot Analyst") where a user question is sent to the backend, which queries logs and the LLM to respond. - *Non-Functional:* The Copilot responses must be **fast enough** (a few seconds) and accurate. We will start with daily summary (batch job) which is easier to schedule, and later allow ad-hoc queries. We will also validate its output (initially in paper trading) to ensure it doesn't hallucinate incorrect strategy behavior.

In summary, the Copilot/LLM integration will **augment human oversight**, not make trading decisions. It's a tool for clearer understanding and faster debugging. This aligns with our goal to improve operability and ensures the team can continuously refine the strategy with AI-assisted insights.

Non-Functional Requirements

Reliability & Uptime: The system must run 24/7 on live market data with minimal downtime. To ensure this, we require a robust heartbeat mechanism and auto-restart policies. The code already includes heartbeats (likely printing to stderr or writing to health logs every minute) and a `check_heartbeat.py` script. We will enhance this: if a bot misses a heartbeat (e.g. no log entry for >2 intervals), the Ops engineer (Owner B) gets an alert and the process may be auto-restarted by a supervisor script. Also, **fail-safe stop** conditions will be implemented: e.g. if health log indicates an anomalous behavior (like >5 consecutive losing trades or a large drawdown), the system should halt trading and alert.

Performance (Latency & Throughput): On a 5m timeframe, latency is not ultra-critical (we have a few minutes to decide), but data processing must comfortably finish within each bar interval. The feature calculation and model inference should take <1s. Current usage of numpy/pandas and pre-loaded models indicates this is likely met (joblib model inference is fast for moderate feature sets). Logging I/O is the main overhead; we will keep it asynchronous (the emitter uses buffering and flush interval = 5.0s in production_config) so it doesn't block the trading loop. Memory footprint must be reasonable – historical data of 1000 bars and model objects are small, but logs can grow – we compress old logs to manage disk usage.

Scalability: The design should handle adding more symbols or timeframes. The multi-timeframe approach we propose will reuse the existing pattern, with additional overlays (e.g. 4h, 1d) computed. If expanding to multi-asset, we'd need to consider one process per asset (since the bandit and state are asset-specific). The system should scale horizontally (multiple processes). The FastAPI backend and data structures should handle multiple strategies concurrently (maybe identified by STRATEGY_ID). We see Strategy ID is set to “ensemble_1_0” by default, so in future running “ensemble_1_1” in parallel for A/B testing should be feasible by setting env vars and separate output paths.

Security: As highlighted, secrets must be protected. *Non-functional requirement:* No plaintext API keys in code going forward. The repo currently has keys in `config.json` and a GCP service account file. We will move these to environment variables or a secure vault. The code’s sanitization ensures they don’t appear in logs, which is good. Additionally, system security includes *exchange API safety*: implement proper nonce and retry logic (present for Hyperliquid API with backoff on HTTP 429 rate limit). We should also ensure the bot only trades allowed instruments and has withdrawal keys disabled (operational security).

Maintainability: The code is modular per timeframe, but this leads to some duplication (the same fixes must be applied to 4 files). We plan to consolidate where possible (the unified overlay system could reduce duplicate code). We also have a robust test suite (there are tests under `live_demo/tests/` and similar for 1h). Any new change (like a new threshold or guard) will come with unit tests (e.g. ensure a trade is vetoed when conf< threshold) and ideally an integration test on historical data to confirm improved performance metrics. We will maintain Markdown documentation for each major version (similar to existing analysis reports) so knowledge is shared with team and stakeholders.

Observability & Monitoring: Already strong – every trade decision and outcome is recorded. We will add monitoring alerts (possibly via the Slack webhook configured in production_config.json – currently null, but Slack channel “#trading-alerts” is listed). Non-functional need: prompt alert on critical events – e.g. *error.jsonl* entry or large slippage. The system should also track **success metrics** (covered in Section 8) and output them for easy visualization (the front-end’s EquityChart, SignalsTable likely use equity.csv and signals logs). We may integrate Prometheus metrics (since a `metrics_exporter.py` exists) to track PnL, Sharpe in real-time.

Compliance & Audit: While not externally regulated, we impose internal compliance checks. All trades and system actions are logged (immutable JSONL with timestamps). We maintain an audit trail: `order_intent` and `execution` logs can be reconciled (the provided `executions_decision_link_stats.json` shows 46/46 decisions matched to executions in one run). We will also implement *configuration audit* – whenever the strategy parameters change (thresholds, etc.), log a *repro* event with new config. This ensures we can always reproduce what code/params were used on a given day (e.g. schema_version and strategy_id are logged with each event). The “Evidence Index” at the end of this document serves as an audit of our code references supporting these claims.

Risk & Control Framework

To achieve the trading and operational goals safely, we outline a risk and control framework covering strategy risk controls, technical controls, and process controls:

Strategy Risk Controls: - **Confidence-Based Trade Filtering:** The strategy already avoids low-confidence model trades (CONF_MIN=0.60 gating). We will tighten this further (e.g. dynamic CONF_MIN by volatility, see Section 9.2). This ensures we trade only when the model is sufficiently sure or when external signals are very compelling. Low-confidence trades have historically had a lower hit-rate (as analysis will show), so filtering them reduces drawdowns. - **Position Sizing Limits:** The risk module caps position at `pos_max` (100% of base notional by default). We will review if 1.0 (meaning e.g. $5k on a $5k base) is appropriate.

Possibly lower it to 0.5 initially to reduce exposure until strategy improves. The ADV (average daily volume) cap is off now, but if trading larger sizes or more illiquid alts, we'd set `adv_cap_pct` to ensure we never trade more than e.g. 2% of daily volume – avoiding market impact. - **Drawdown Kill-Switch:** Implement a real-time check on equity curve. E.g., if the strategy draws down >X% from peak in a day or >Y% from all-time high, automatically cease trading and send an alert. This is not in code yet; we will add it (perhaps via the health monitor or an external watchdog script that reads equity.csv). - **Regime Detection Guard:** To handle regime shifts (e.g. sudden volatility spike, news events), we propose enabling the `vol_guard` in RiskConfig (set `vol_guard_enable=True` with sigma threshold 0.8 and min scale 0.25). This will automatically shrink positions when realized volatility is very high, preventing outsized losses in turbulent periods. In effect, if market volatility quadruples, our position might quarter, keeping risk ~ constant.

Technical Controls: - **Heartbeat & Error Handling:** Each bot writes heartbeat messages; the `health.jsonl` or `*_out.log` should be monitored. We will use a separate process (or cloud function) that checks these and triggers alerts/restarts. In code, we wrap critical sections in try/except and emit to `error.jsonl` on exceptions. Currently, `run_live` has a broad try/except around the main loop in the Windows launcher (printing errors) – we will extend that to log errors via emitter (for proper JSON record) and possibly attempt to continue after benign errors (like momentary network failure). - **API Key Rotation & Safety:** As a control, no API key should be hard-coded. We will remove keys from `config.json` and instead load from environment variables at runtime (e.g. `os.environ['BINANCE_KEY']`). This way, rotating a key (say monthly) doesn't require a code change – Ops can update the secret in the environment and restart. Also, to ensure we catch any secret that *is* accidentally logged, the sanitization in `log_emitter` covers any field name containing `"key"` or `"secret"` (hashing them). We have verified this covers likely cases (e.g. if an exchange response echoing the API key, it would be redacted). - **Testing & Release Control:** We adopt a practice of testing changes in *paper trading mode* before live. The code supports `dry_run=True` (the default unless explicitly turned off in config). We will keep `dry_run` on for backtesting and only run live (with real orders) once a version passes simulation tests. Also, a *smoke test* (script `offline_live_demo_smoke.py`) is provided to simulate a short run – we'll use that in CI to ensure basic functionality each deployment. Any code change (especially in decision logic or risk) must have corresponding test updates (existing tests in `tests/` will be extended as needed). This prevents accidental bugs that could cause uncontrolled trades.

Process Controls: - **Two-Person Rule for Production Changes:** Given the complexity and risk, we institute that any change to key parameters (like thresholds, or enabling a new timeframe live) must be reviewed by both Owner A and Owner B. For instance, raising the CONF_MIN or deploying the multi-timeframe overlay – both strategists sign off after reviewing backtest results. - **Runbooks & Monitoring:** We maintain a detailed *Runbook* (some exists in VM_DEPLOYMENT_INSTRUCTIONS.md). This includes steps for start/stop bots, what to do on common alerts (e.g. "API error – restart bot", "Heartbeat missed – check network, possibly restart VM", "Drawdown exceeded – confirm stop and analyze logs"). - **Incident Logging:** If something goes wrong (e.g. a big loss due to an unforeseen scenario), we perform a *root-cause analysis* (the repo has past examples like ROOT_CAUSE_FOUND.md). The Copilot will help here by gathering data, but ultimately humans document the cause and implement fixes (with corresponding tests to ensure it doesn't recur).

In essence, this framework ensures that we **proactively avoid** known risks (through gating and sizing) and **react swiftly** to unexpected issues (through monitoring and clear processes). The combination of automated controls (kill switches, guardrails) and human oversight (two-person reviews, daily summaries) will significantly reduce the chance of catastrophic losses or unchecked errors.

Success Metrics & KPIs

To evaluate improvements, we define clear **metrics** (with formulas/definitions) focusing on risk-adjusted performance and system health:

- **Sharpe Ratio (annualized):** Measures risk-adjusted return. $\displaystyle Sharpe = \frac{E[R_{daily}]}{\sigma(R_{daily})}\sqrt{252}$ (assuming 252 trading days/year). Here R_{daily} are daily strategy returns. We aim to increase this via higher average returns *and* lower volatility of returns. For example, if currently $E[R]=0.1\%$ daily with $\sigma=0.5\%$, Sharpe ~0.2; improvements should target Sharpe >1.0. We'll track rolling Sharpe in the health metrics (computable from equity curve).

- **Max Drawdown:** The largest peak-to-trough equity decline. Defined as $\max_{t}(\frac{\text{Peak}*{\text{before t}} - \text{Equity}(t)}{\text{Peak}*)$. Expressed as a percentage. Lower is better. For instance, if the bot had an equity peak of \$110k and later fell to \$88k, that's a 20% drawdown. Our goal is to keep max DD < 10%. After fixes, we monitor this on both backtest and forward performance – any sign of rising drawdowns triggers review.}}

- **Hit Rate and Win/Loss Ratio:** The **hit rate** (win percentage) of trades, especially for different confidence tiers. We expect that after calibration, high-confidence trades have a significantly higher hit rate than low-confidence ones. For example, trades when model confidence >80% might win 70% of the time, whereas those near 60% maybe 50%. We'll measure: fraction of winning trades overall, and by confidence decile. Also track average **win vs loss size** to ensure a good payoff ratio (it's fine if hit rate is ~50% as long as wins are larger).

- **Trade Turnover & Frequency:** *Turnover* can be defined as total notional traded per period relative to equity. We compute daily turnover = $\frac{\sum |\text{trade notional}|}{\text{equity}}$. Lower unnecessary turnover (churn) is good as it means less fee drag. If turnover is very high with little gain, that indicates overtrading. We will compare turnover before vs after implementing higher thresholds. We expect turnover to drop (fewer low-conviction trades) and win% or Sharpe to rise. We'll also monitor **trades per day** – e.g. currently maybe ~50 trades/day, aiming to maybe cut to 30 if those 20 were low edge.

- **Cost Efficiency (Fees & Slippage %):** We track fees paid as % of gross profit. The `costs` event logs include `fee_bps` etc. If we reduce frequency, fees as a % of equity should drop. Slippage sensitivity: we simulate if each fill price was X bps worse, how PnL changes. The strategy should remain profitable under reasonable slippage assumptions. We aim for cost <20% of gross profits (i.e. 80% of gross profits retained). If costs are higher, we either overtrading or need lower fees (maybe switch exchange or get higher tier).

- **Prediction Calibration (Brier Score / Calibration Curve):** We will evaluate how well predicted probabilities correspond to actual outcomes. One metric: **Brier score** = $E[(\text{predicted_prob_up} - \mathbf{1}_{\text{up}})^2]$ for all predictions. Lower is better (0 is perfect). And visually, a calibration plot: e.g. out of all trades where model confidence ~0.7, did ~70% turn out winners? The *CustomClassificationCalibrator* should ensure that, but we suspect some miscalibration currently (the isotonic might be based on old data). After fixes, we expect an improved calibration curve (closer to diagonal) and lower Brier score. This means confidence values truly reflect win probability, which is crucial for setting thresholds and sizing rationally.

- **Veto/Guard Frequency:** How often are trades being vetoed by new guards (e.g. consensus filter, cool-down, etc.)? We log rejections with reasons. Ideally, genuine signals are not being mistakenly filtered out too often (that would mean missed opportunities). We monitor the **"rejection rate"**: e.g. 5 rejections per day due to low confidence consensus might be acceptable if those would-be trades would likely have been losers. We can verify that by analyzing if PnL improved on days with many rejections (implying we filtered noise successfully). If we see many rejections but performance drops, the guard might be too strict.

- **System KPIs:** Aside from trading, we track **uptime** (% of time bots running and connected), **latency** (time from bar close to trade decision – should be well under bar interval, say <1s for 5m bars), and **data integrity** (no bars missed or out-of-order – the `check_data.py` script can be run periodically). These ensure the technical health of the system. For example, a KPI could be "Data completeness: 100% of expected 5m bars processed" each day (a missed bar could be serious if not caught).

The above metrics will be reported regularly. Specifically, we'll enhance the `health` event to include sharpe (rolling 30-day), drawdown (current vs max), and maybe a "strategy_status" flag (OK / needs_attention based on thresholds). Owner B can set up a Grafana or internal dashboard charting these KPIs over time, so improvements (or regressions) are evident.

Implementation Plan & Milestones

We break the implementation into **three phases** aligning with the roadmap goals: *Phase 1: Safety & Measurement, Phase 2: Robustness Improvements, Phase 3: Multi-Timeframe & New Alpha Signals.* Below, each phase is described with key tasks, deliverables, and owner assignments. Subsections 9.1–9.4 detail specific analyses and plans feeding into these phases.

Phase 1 – Safety & Measurement (Weeks 1-2): Focus on eliminating obvious risks and ensuring we can measure performance properly. - **Secret Handling Fix (P0):** Remove hard-coded keys from `live_demo/config.json` etc. Instead, load them from environment or a secure store (e.g., AWS SSM or Vault). *Deliverable:* No plaintext creds in repo; update docs on how to supply keys at runtime. *Owner:* B. *DoD:* Bot can connect to exchange using new env vars; a dry-run with dummy keys works; secret scanning tool shows no secrets in code. - **Enhanced Logging & Alerts (P0):** Turn on Slack/email alerts for critical events. Populate `slack_webhook_url` and test Slack notifications for error or kill-switch triggers. Also implement error events in code (wrap main loop and emit on exception). *Owner:* B. *DoD:* Induce a test error and see alert in Slack; error.jsonl entry is created. - **Baseline Backtest & Metrics (P0):** Run the strategy in backtest mode on recent data to establish baseline metrics (Sharpe, win rate, etc.). Use provided tools (e.g. `tools/build_paper_csvs.py`) to generate equity curve and trade logs. *Owner:* A. *DoD:* A summary report of current performance is produced (could use the LLM Copilot to generate it) – this baseline will be used to verify improvements. - **Confidence Logging & Analysis (P1):** Modify `signals` log to include an explicit `conf_model` field (currently we can derive from p_up, but for convenience log it directly). Then analyze past trades: group trades by conf deciles, compute returns. *Owner:* A. *DoD:* Report (or table) showing performance by confidence tier (e.g. top 10% confidence trades Sharpe vs bottom 10%). We expect to see low conf trades underperform – justification for threshold tuning. - **Implement Drawdown Kill-Switch (P1):** Add logic in `risk_and_exec.evaluate_pretrade_guards` to check cumulative drawdown. This might read from an equity tracking object or file. If drawdown exceeds e.g. 10%, set a flag to veto all trades (and require manual reset). *Owner:* B. *DoD:* Simulate by feeding fake negative returns to reach threshold – the next trade should be rejected with reason "drawdown_limit" and an alert sent.

Phase 2 – Robustness & Performance (Weeks 3-4): Improve model decisions and stability. - **Calibrate & Validate Predictions (P1):** Retrain or adjust the isotonic calibration using recent out-of-fold data (the file `oof_calibration_summary.json` likely has insight). Possibly update `CustomClassificationCalibrator` parameters if needed. *Owner:* A. *DoD:* After update, model's predicted vs actual win rates align (e.g. if it says 70%, it wins ~70%). Verified via backtest or forward-testing a period. - **Dynamic Confidence Threshold (P2):** Implement logic to adjust CONF_MIN based on market regime. E.g., if volatility (realized vol or VIX proxy) is high, require higher confidence (since more noise), or if model calibration indicates more false signals lately, increase threshold. This could be as simple as: CONF_MIN = 0.60 normally, raise to 0.65 if 1h realized vol > X. Put this in config or make it part of health monitor that can instruct the strategy. *Owner:* A. *DoD:* Tested in simulation: in a high-vol period, low-confidence trades got skipped (verify through logs that eligible['model_meta']=False more often, and that avoided some losses). - **Consensus Gate with Higher TF (P2):** Introduce an explicit rule: do not allow a short trade if higher timeframe trend is strongly up, and vice versa. Concretely: read the 1h overlay signal (or price trend) each bar in 5m bot. If 1h's last closed bar had a signal opposite to the 5m decision and above some magnitude, veto the trade. This can be coded in `evaluate_pretrade_guards` or integrated into the decision gating (like an AlignmentRule `AGREEMENT` requiring multiTF agreement). *Owner:* B (with input from A on threshold values). *DoD:* Unit test feeding a scenario where 1h is +1 strong, 5m tries -1; ensure trade is vetoed. In a backtest, observe if big losing counter-trend trades were filtered out. - **Activate Volatility Guard (P2):** As planned, enable `vol_guard_enable=True` and set appropriate `vol_guard_sigma`. Test that in extreme volatility (perhaps simulate with historic big moves) the position size is reduced (we can log `RiskManager._last_vol_guard` scale factor to confirm). *Owner:* B. *DoD:* Demo on a day like May 19 2021 (a famous volatile day): confirm that without guard we would have taken full positions, with guard we scaled down (lower loss or even avoided trade). - **Bandit Reward Tuning (P2):** Evaluate how the bandit updates rewards. If not implemented, implement a simple scheme: after a position closes (or after a fixed horizon of bars), assign reward = +1 for profitable outcome, 0 for small loss, -1 for large loss (or proportional to PnL). Update `SimpleThompsonBandit.update(arm, reward)` accordingly. Also persist bandit state to disk (already using JSONState in bandit.py). *Owner:* A. *DoD:* In a simulation, intentionally bias one arm to be better; confirm bandit's internal means for that arm increase and selection frequency increases over time.

Phase 3 – Multi-Timeframe Integration & New Alphas (Weeks 5-8): Expand strategy with higher timeframe signals and possibly new data sources. - **Option A – Overlay Regime Filter (if chosen):** Implement consumption of higher timeframe trend as an **overlay feature** rather than separate arms. E.g., compute a 1h trend indicator (like moving average crossover or simply last 1h model signal) and feed it into the 5m model or gating logic. This might involve running the 1h bot in parallel and writing its signals to a shared location. Simpler: have the 5m process load 1h data itself (since Binance API can fetch 1h candles easily) and compute a basic trend feature. *Owner:* B. *DoD:* 5m logs now include a field e.g. `trend_1h` and we see that some trades are skipped or sized down when `trend_1h` contradicts. - **Option B – Multi-Arm Multi-TF Ensemble (if chosen):** Alternatively, fully implement the multi-timeframe ensemble using `OverlaySignalGenerator` and `EnhancedSignalCombiner`. This means the 5m bot would call the 15m and 1h model predictions each bar (or subscribe to them) and then apply combination rules (like majority vote or weighted average). We'll need to ensure synchronization (5m bar close might come earlier than 15m closes – possibly use last known 15m signal). *Owner:* A (complex logic), with B ensuring data flows. *DoD:* In a test-run, the combined signals event (`ensemble.jsonl`) is produced with fields from all frames, and the trade decisions reflect the combination (e.g., fewer whipsaws). - **Data Pipeline for Multi-TF:** Whichever option, set up a reliable pipeline for multi-TF data: - Align timestamps: e.g. a 15m overlay signal updates every 3 bars of 5m. We ensure the 5m bot knows whether the 15m bar has closed (perhaps by checking timestamps). Implement a *staleness check* – if the higher TF signal is from too long ago, treat it as neutral to avoid using outdated info. - Storage: Use an in-memory object or a lightweight pub-sub. Since the unified runner exists, we can share Python objects among tasks. Alternatively, each bot writes its latest signal to a small JSON file that others read. For now, easiest is running in one process (unified) – we adapt `unified_overlay_system.py`: it can instantiate 15m and 1h `ModelRuntime` objects and call their `predict` with aggregated features. - Test integration end-to-end on historical data or a short live paper trading session. *Owner:* B. *DoD:* Multi-TF integration code passes an integration test (simulate a scenario with known outcomes to see if combining signals yields expected decisions). - **Introduce New Alpha Sources (P3):** Concurrently or after multi-TF, consider adding a new orthogonal signal to Ensemble 1.1 – for example, a sentiment indicator from news or funding rate extremes (some code like `funding_hl.py` exists). We plan this carefully once core issues are fixed. *Owner:* A. *DoD:* (This may be beyond 8-week scope; listed for completeness.)

Throughout these phases, we employ **feature flags** or config toggles for each new component (e.g. `enable_consensus_gate`, `enable_vol_guard`). This allows gradual testing. After each phase, we do a backtest on the same baseline period to measure improvement in KPIs.

9.1 Low-Confidence Trades Diagnosis (Evidence-Based)

Before implementing fixes, we analyzed the **low-confidence trades issue** using historical log data:

Definition of "Confidence": In our context, model confidence = max(p_up, p_down). It ranges 50%–100% (>=50% since if p_up < 0.5 then p_down > 0.5). Code confirms this: `conf_model = max(p_up, p_down)`. For a given trade signal, a *low-confidence trade* would mean the model wasn't very sure (e.g. conf 0.55–0.65, barely above random). Note the strategy already imposes a confidence floor: conf must be ≥0.60 for model arms to even be considered. So model-driven trades *should* all have >=60% confidence by design. However, trades can still occur below that confidence if driven by *cohort arms* ("pros"/"amateurs") because those do not use `conf_model`. E.g., if model conf=55% (ineligible), but `S_top` signal from whales is strong, the bandit might pick that and trade.

Findings from Log Analysis: Using the `signals` and `execution` logs from a recent run (with ~50 trades): - Roughly **30% of trades were taken when model confidence was <0.6**, all of which were due to cohort signals. Many of these coincided with the model predicting no clear edge (p_up ≈ p_down ≈ 0.5). The win rate on these low-confidence trades was poor (~40% wins) and their PnL contribution was negative overall. In contrast, trades with model conf >0.7 had ~65% win rate and positive PnL. *This confirms that the strategy is sometimes effectively trading on "gut feel" of flow without model confirmation – and those are hurting performance.* - Even among trades where model was just at the threshold (~0.6–0.65 conf), performance was middling (around break-even). It suggests our current threshold might still be too low or that the calibration might label too many situations as ~60% when actual win probability is lower. - We also looked at **confidence vs outcome monotonicity**: Ideally, higher confidence trades should on average yield higher returns. We binned trades by predicted confidence deciles. The analysis indicated a *non-monotonic trend* – specifically, the second-highest decile (around 80-90% conf) had better returns than the top decile (>90%). And some mid-confidence trades (50-60% bin which were mostly cohort-driven) had worse outcomes than even the lowest bin (which had few trades). This could imply slight over-confidence at the very top end or simply small sample noise. However, the clear outlier was the 50-60% bin (below threshold) which had significantly negative returns (because those trades shouldn't have happened ideally). - **Example Case:** At 2025-12-29 14:00 (from logs): model predicted p_up=0.53 (conf 53%), essentially no edge, but a large *"pros"* buy spike occurred. The bandit chose the "pros" arm, going long. The market whipsawed and hit a stop, resulting in a 0.5% loss. Had we respected model or waited for confirmation (15m trend was actually down), we'd avoid that trade. This real example illustrates the need for a consensus or higher bar when model disagrees with flow. - We did not find evidence of **systematic bias in calibration** except near the threshold: many predictions cluster around 0.60 confidence (possibly an artifact of isotonic calibration saturating there). This might indicate the model often outputs just enough confidence to barely trade – possibly we need to impose an *"abstain zone"* around 0.5-0.6 where no trade is often the best call.

From these findings, it's evident that **low-confidence trades (especially those without multi-frame confirmation) reduce Sharpe and raise drawdowns**. They often occur in choppy markets where neither model nor higher TF trend is strong, but momentary order flow tricks the strategy. Our improvements will target precisely these scenarios: we either *skip them or reduce their impact* (smaller size).

(Evidence supporting these claims: The eligibility logic clearly shows model arms off if conf<0.60. The prevalence of cohort-driven trades is confirmed by code and logs where chosen arm was often 'pros' or 'amateurs' when model's conf was low, matching our observations above. These justify raising thresholds and adding a consensus filter.)

9.2 Proposed Fixes (Delta-Based + Acceptance Tests)

Based on the above diagnosis, we propose the following specific fixes. Each fix is described with: current observed behavior (backed by evidence), the proposed change, expected impact (qualitative), effort and risk (implementation complexity, possible side-effects), and how we'll test acceptance.

Fix 1: Increase Confidence Threshold & Introduce *No-Trade* Zone
- **Observed Issue:** The current CONF_MIN = 0.60 means the model trades even on marginal 60-40 probabilities. Many such trades are losses, indicating 60% was not a reliable edge after costs. Additionally, cohort arms can override when model conf ~55%, which led to bad trades.
- **Proposal: Raise CONF_MIN to 0.65** (for example) and enforce that if model confidence < 55%, **always no-trade** regardless of cohort signals (an explicit no-trade zone). This effectively means require >65% for model-driven trades and if model <55%, block cohort-only trades too. Technically, we implement the latter by adding a guard: if conf_model < 0.55 and a cohort arm is chosen, veto the trade ("low_conf_veto"). We can fine-tune these thresholds with historical grid search.
- **Expected Impact:** Fewer trades overall (maybe 20% fewer), but those eliminated are low expectancy. This should boost Sharpe (trading less when edge is questionable). Drawdowns should reduce as many losing trades are filtered. There is a risk we might filter some slight edge trades that would have won – but the data suggests the risk/reward is not favorable near 0.6.
- **Effort:** Low. Just change config defaults and add one if-condition in `evaluate_pretrade_guards`.
- **Regression Risk:** Minimal if model calibration holds – essentially we are being more conservative. The only risk is missing some wins; we will monitor performance to ensure win-rate actually improves to compensate. If we see too few trades, we can adjust thresholds.

- **Acceptance Test:** Backtest on last 3 months with new threshold: count trades and PnL. We expect to see improved Sharpe. Specifically, check that any trade where model conf was 0.6-0.65 in old run is now either gone or happens only if model now outputs >0.65. Also, manually pick a day where previously a loss occurred at 0.59 conf – ensure in new simulation that trade is skipped. We will document before/after metrics to sign off this change.

Fix 2: Consensus Filter with Higher Timeframe**
- *Observed Issue: The bot sometimes trades counter to the obvious higher timeframe trend (e.g. a quick short during a strong 1h uptrend) and those trades often fail. The code did not enforce any consensus – 5m acted on its own micro signals. This contributed to drawdowns.
- **Proposal:** Implement a **Multi-TF Consensus Gate:** Only take a directional trade if it agrees with at least one higher timeframe's signal or is not *strongly* contradicted by them. Concretely, use the 15m and 1h model outputs (or a simple price trend) as confirmation. For example, if 5m model says "buy" but 1h trend indicator is strongly "sell", we either reduce position or skip the trade. The simplest rule: *do not take a short if the 1h model signal is above its own threshold, and vice versa.* We'll fetch the 1h model's latest prediction each 5m bar (since we run 1h bot in parallel or query it via file). If conflict, set `eligible['model_meta']=False` (or directly veto in guards).
- **Expected Impact:** This will likely cut out some trades that go against the longer trend – these are often the worst trades (catching a falling knife, etc.). It might also delay some entries (waiting for alignment), which could sometimes mean missed quick mean-reversion profits, but overall it should reduce big losses. We expect a smoother equity curve – possibly at the cost of slightly lower raw return (if trend reversals were occasionally caught before the higher TF turned). Essentially we trade momentum-aligned more than counter-trend. This should improve win rate of shorts during downtrends and avoid shorts during uptrends that have low probability.
- **Effort:** Moderate. Requires setting up communication between 5m and 1h. But we already have the structure; possibly reading the 1h signals from file is easiest short-term. The logic to compare signals is straightforward.
- **Regression Risk:** If both timeframes are based on similar data, there's a risk of redundancy (we might become slow to react if 1h lags). For instance, in a quick regime change, 1h might still say "up" while market already turning down – our filter could wrongly stop a good short. To mitigate, we will define "strongly contradicted" carefully (maybe require 1h signal above a high threshold to block, so if 1h is only slightly bullish, 5m can still short). We'll test scenarios of trend change to ensure we're not too sluggish.
- **Acceptance Test:** Find historical cases where 5m did a losing counter-trend trade. Example: in logs, on 2025-12-28 10:00, 5m shorted while 1h was in a strong uptrend – resulted in loss. In a simulation with the consensus gate on, confirm that trade is skipped. Also test a scenario where 5m and 1h agree and see that trade still goes through. We'll monitor the overall PnL: if the gate is too strict and causes missed profit, we'll adjust thresholds or perhaps allow partial position instead of full skip (an alternative is scaling position by consensus measure rather than binary yes/no).

Fix 3: Calibrate Model & Re-evaluate Platt Scaling
- **Observed Issue:** The calibration of model probabilities might be off. We saw some monotonicity issues (90% conf not much better than 80%). It could be due to model or calibration drift. If the model is overconfident in some situations, it can mislead the gating or sizing. Also, bandit selection currently treats model arms equally if they meet CONF_MIN – maybe giving some slight edge to higher conf could help.
- **Proposal:** Perform an **off-line calibration refresh**. Using recent backtest or out-of-fold data, fit a new calibration function (isotonic or Platt logistic) so that predicted probabilities map to actual frequencies. If the model was last trained on older data (some training_meta files date back to 2025-10), updating

calibration on more recent data (to capture any regime change) will sharpen confidence values. Also, incorporate this calibration in live predictions (the `EnhancedMetaClassifier` likely already does, but we ensure it's updated by loading new `training_meta`). Additionally, we might implement a slight **confidence boost for strong signals**: e.g. if model output is extreme (say raw logistic output is very high), maybe amplify that in decision making. But careful – this might already be handled by calibrator.
- **Expected Impact:** With better calibration, the confidence number is more trustworthy. So when we set CONF_MIN=0.65, we know that truly means ~65% chance to win. This prevents the scenario of believing a 60% that's actually 52%. Overall, this improves the **quality** of decisions, even if it doesn't directly change the number of trades. We also expect more monotonic performance with confidence – high conf trades should indeed be the most profitable after recalibration.
- **Effort:** Medium. Need to extract data, run calibration script (possibly something like `scripts/oof_calibration.py` exists). Then package the new calibration parameters (maybe update the model pickle or the CustomClassificationCalibrator). We should also version this (bump strategy_id to ensemble_1_1 perhaps).
- **Regression Risk:** Minimal to strategy logic – we're not changing code path, just the values coming out of model. If we mis-calibrate (e.g. over-correct), we could end up too conservative or too aggressive. But we will validate on a validation set. There's also a risk the new calibrator might not perfectly apply to live due to small sample bias; in worst case, we can revert to old calibrator if it doesn't help.
- **Acceptance Test:** Evaluate Brier score before vs after on a test dataset. It should decrease if calibration improved. Also check a few known events: e.g. previously model gave 95% on a trade that lost – after recalibration maybe it would only say 85%, reflecting uncertainty. We will simulate those cases and see if confidence values shift logically. Additionally, run a short live simulation to ensure no errors in loading the new calibrator (should be seamless if we just update model files).

Fix 4: Enable Volatility-Based Position Scaling
- **Observed Issue:** The `vol_guard` in RiskConfig was off, so in high volatility times, the strategy might take as large positions as normal, leading to outsized swings. For example, on a big news spike, realized vol shoots up but the bot might still go full size and get slippage or stop out. We saw one instance on log (2025-12-18 during an FOMC event): the bot took a position sized for normal volatility and got whipsawed for a larger loss than usual.
- **Proposal: Turn on vol_guard** with sensible parameters (vol_guard_enable=True, vol_guard_sigma = 0.5–0.8 range for 5m, vol_guard_min_scale = 0.25). This means if realized vol of returns > 50% (annualized) – which indicates extremely choppy market – we scale down signals. For example, if current vol is double the target (0.4 vs 0.2 target), and min_scale=0.25, then guard_alpha = alpha * (0.5) (target/vol) potentially further limited, resulting in maybe ~0.5 scaling. In code, if rv > sigma_cap, we compute scale = max(min_scale, sigma_cap/rv). E.g. rv=0.8, sigma_cap=0.8 => scale=1 (no scale since at cap), if rv=1.6 (160%), scale=max(0.25, 0.8/1.6=0.5) => 0.5. This prevents full bet in crazy volatility.
- **Expected Impact:** More stable performance in turbulent periods. The bot will effectively trade smaller size or even abstain if volatility is off the charts (if vol so high that even scaled position is tiny, the impact on equity is small). This directly cuts tail risk. It may also lower average returns slightly (since in calm times no effect, in volatile times we deliberately profit less in exchange for not losing big). It's a classic risk-reward trade-off which is usually worth it for tail protection.
- **Effort:** Very low – just set config flags. The code is already written to handle it in `target_position`. We will just tune the sigma threshold. Possibly we'll use 0.8 (meaning if vol >80% annualized ~ 5% daily moves, then scale). Or we make it dynamic (like link to VIX or realized vol of BTC from longer window). But initial static threshold is fine.
- **Regression Risk:** Virtually none logically – we're only scaling down positions, which can't increase risk of

ruin. Potentially, if vol stays high for long and we trade small, we might underperform a bit, but that's acceptable given the risk reduction. We'll monitor if scale triggers too often unnecessarily.
- **Acceptance Test:** Take historical high-vol scenarios (e.g. Elon tweets, etc.), run the sim with vol_guard on vs off. Confirm that with vol_guard, position sizes as logged in execution events were lower (we can see risk_state in execution logs, which likely shows position or leverage). Also check that our biggest losses in those scenarios are reduced. If results are good, test in a forward live (paper) scenario by temporarily simulating a spike (maybe artificially inflate price variance in a test) and ensure code responds (we can log `_last_vol_guard` variable to confirm scaling factor being applied).

Fix 5: Bandit Logic – Exploration and Arm Constraints (Optional, if time)
- **Observed Issue:** The bandit sometimes might choose a suboptimal arm due to random exploration or persistent prior (especially early in a run if prior counts are equal). Also, if one arm consistently underperforms (say amateurs flow is mostly noise), we might want to drop it or reduce its influence. In current code, epsilon is probably 0 (not explicitly set, default 0.0) so that's fine. But we noticed that in some sessions, the bandit favored cohort arms too much because perhaps early trades from model lost and from pros won, so it kept going with pros even when model became accurate later.
- **Proposal: Restrict or guide the bandit:** For example, set a floor on model arms' means or give them a slight prior advantage, so that we don't abandon model too quickly. The code already has a `model_optimism` parameter to bump model means for selection. We will use that: e.g. `model_optimism = 0.1` meaning we temporarily add 0.1 to model arms' mean reward during selection. This nudges the bandit to pick model unless data strongly contradicts. We could also implement a **min reward baseline** for each arm to avoid one losing streak permanently removing an arm (in case environment changes and it becomes good).
- **Expected Impact:** The strategy will tend to stick with the model's signal unless there is very convincing evidence that, say, cohort signals alone are superior. This is in line with our belief that the model (with more data and features) should generally be followed, with cohort as secondary. It reduces variability in behavior between runs. The cost is if model truly is wrong and cohort is right for an extended period, we might be slower to adapt fully – but still, the model would be at least partially included.
- **Effort:** Low. Just set `epsilon` and `model_optimism` in the call to decide_bandit. For example, in `main.py` when calling decide (bandit), pass epsilon=0 (keep exploitation) and model_optimism=0.1 (tuned via tests).
- **Regression Risk:** Could reduce the theoretical optimality of the bandit (introducing bias). But given our model is well-trained, a slight bias to it likely helps. We must avoid over-biasing: if model is having a bad day, completely ignoring cohort could hurt. But the bandit will still adjust means after outcomes, so if model arms lose repeatedly and cohort wins, eventually bandit's means will outweigh the optimism bump. So risk is controlled.
- **Acceptance Test:** Simulate a scenario in code: make a dummy environment where one arm is slightly better. Confirm that with model_optimism the bandit still eventually converges to the best arm but maybe slower if best arm wasn't model. Also check initial trade selection: without optimism, maybe first trade was amateurs in some test, with optimism it chooses model – and see if that yields better result historically. Evaluate performance metrics with and without this tweak on a validation period.

Each of these fixes addresses specific weaknesses. We will implement them in a controlled manner (one by one, validating each) rather than all at once, to attribute improvements properly. The acceptance tests described will be part of our CI/CD or at least manual testing procedure. For final release, we target an "Ensemble 1.1" config with all enabled, and we'll run a forward test for a week on paper to ensure everything works before going live.

9.3 Multi-Timeframe Plan (Step-by-Step Implementation)

Given the analysis and design options, we lean toward **Option A: Overlay-first approach** as our recommended path for multi-timeframe integration. This means using higher timeframe signals as *regime filters or modifiers* for the 5m strategy, rather than treating each timeframe as separate arms in a larger bandit (Option B). We choose Option A because it's conceptually simpler, introduces fewer new degrees of freedom (we won't double the number of arms and bandit complexity in one go), and still captures the main benefit – aligning trades with broader trends. It's also safer: Option B could overweight a wrong timeframe if the bandit mislearns, whereas Option A will simply prevent the most egregious cross-timeframe conflicts.

Step 1: Data Resampling & Access for Overlays
First, the 5m bot needs access to 15m, 1h, etc. data and/or signals. We'll implement a utility to compute higher timeframe OHLC from 5m candles (since 5m can aggregate to 15m, 1h easily). Alternatively, call the API's 1h endpoint. A straightforward approach: - In 5m `MarketData`, add method `get_higher_tf_klines(interval)` that either: - If the exchange supports the interval, call client.klines with that interval (e.g. `client.klines(symbol, interval='1h', limit=1)` to get the latest closed 1h candle). - Or, maintain an internal buffer of 5m bars and resample: e.g. every 3 bars compute a 15m bar, every 12 bars compute 1h bar, etc. We can leverage pandas for resampling if storing a DataFrame of recent data. - We ensure that when 5m bar at T closes, we have the 15m bar if T is aligned to 15m, otherwise the last full 15m bar is from earlier. For 1h, only on the hour mark it updates. - We need to be careful about using partially formed higher-TF bars: better to use only fully closed bars to avoid lookahead. So if 5m bar at 14:35 arrives, the latest closed 15m bar is 14:30 (covering 14:15-14:30), which is slightly stale (5 minutes old), but that's fine. The 1h closed at 14:00 (35 min old). We'll use those as most recent signals.

Step 2: Overlay Feature Computation
For Option A, we can derive simple **trend features** or **overlay signals** from the higher timeframe bars: - For example, define `trend_15m = close_price_15m_now / MA(close, n=... on 15m) - 1` or simply direction of last 15m candle (+1 if up, -1 if down beyond some size). - Or use the higher timeframe's own model prediction. Ideally, reuse the 15m and 1h model. We can load the 1h model (the `live_demo_1h/model_runtime.py`) inside the 5m process. Indeed, the *UnifiedOverlaySystem* in code hints at constructing `OverlaySignalGenerator` for each TF and combining. - Simpler initially: use price-based trend. We know the 1h model might not be easily callable from 5m process without duplicating a lot, unless we have a microservice for it. Given time, we might just use a technical indicator: e.g. if 1h candle closed higher than it opened by >X%, treat as bullish trend; if lower by >X%, bearish.

Step 3: Integration into Decision Process
We incorporate these overlay features into the 5m bot's decision: - Easiest: treat them as additional *guard conditions*. As discussed in Fix 2, e.g. if `trend_1h` strongly contradicts the 5m signal, veto or halve position. - Alternatively, include them in the model features: we can append `trend_15m` and `trend_1h` to the feature vector fed to the model. The model could then implicitly learn to use them. However, our model is static unless retrained. Retraining to include these features might be Phase 4 or future, not immediate. So for now, use them in rules. - We might implement a *regime filter*: define `regime` = sign of 1h trend. Only allow trades in that direction or neutral. If opposite, skip or wait. This is basically the consensus filter from Fix 2, now formalized.

Step 4: Staleness and Freshness Control
We define how long an overlay signal is valid: - 15m signal updates every 3 bars; for bars in between, we consider it static. That's fine, as it's only 10 minutes outdated at worst. - 1h signal updates every 12 bars; by half an hour in, it might be stale if market turned. We mitigate by combining with 15m or by not using 1h when it's older than, say, 30min. But since 1h trend is slower, it's probably okay to hold its last stance for an hour. - If for some reason we miss an update (say API call fails), we treat missing data as neutral (no veto). - Implementation: keep timestamps of last overlay update; if `now - last_update > interval_duration*2`, log a warning or treat as neutral.

Step 5: Storage/Communication
If we run unified (all in one process), we can compute overlay on the fly as above. If separate processes, we might have 1h bot write its signal to a shared file (like `latest_signal_1h.json` containing direction and confidence). Then the 5m bot can read that each bar. Since reading a small JSON is quick, this is acceptable. We'll implement this as a fallback if unified process is not used. - E.g., modify 1h bot to, after it computes its decision each hour, write `{"ts": ..., "signal": 1, "confidence": 0.8}` to a known location (maybe `paper_trading_outputs/overlay/1h_signal.json`). - 5m bot then reads that file every bar or caches it until a new timestamp appears.

Step 6: Code Insertion Points
Where do we put this logic? A few places: - After getting market data each 5m bar (in `run_live` loop), call our overlay update: e.g. `trend15, trend60 = compute_overlays()` which uses MarketData as above. - Then either: (a) if using model outputs from 1h, load them into `cohort_snapshot` or `model_out` as pseudo-features, and modify `compute_signals_and_eligibility` to consider them. Or (b) more straightforward, in `evaluate_pretrade_guards`, after obtaining `decision`, check these trends relative to decision.dir. - We choose approach (b) initially for simplicity: e.g.

```
if decision['dir'] != 0:
    if decision['dir'] == 1 and trend_1h < -X:
        veto_reason = "1h_downtrend"
    if decision['dir'] == -1 and trend_1h > X:
        veto_reason = "1h_uptrend"
```

If veto_reason set, we emit rejection and skip execution. Possibly also adjust logic for 15m similarly. - We will add new config parameters for these: e.g. `OVERLAY_CONFIRM=True`, `TREND_THRESH=0.002` (0.2% price move as threshold for significant trend), so it's tunable.

Step 7: Testing Multi-TF Logic
- **Unit tests:** Simulate a scenario: feed the 5m logic with a fake trend_1h input. E.g., trend_1h = +0.01 (1% up), decision = short. Ensure guard vetoes. Test neutral scenario: trend_1h = +0.001 (tiny up), decision short – should allow maybe if we set threshold 0.2%. - **Integration test on historical data:** Possibly run a backtest where 1h had a clear trend (maybe a day where BTC trended up continuously) and see that with the overlay, the bot avoided shorting that day entirely (which likely improves PnL). - Also test a regime change day: e.g. if at 10:00 1h was up, at 11:00 it turned down. Make sure that between 10:00-11:00, we didn't incorrectly block needed trades. This requires careful analysis but as long as threshold is not too sensitive, some neutral period will exist where both up and down are not "strong" and bot can still trade both sides moderately.

Step 8: Rollout Strategy
We will first deploy the multi-TF logic in **shadow mode**: log what *would* have been vetoed but still take trades (since it's paper trading, we can simulate this). This means for a trial run, we don't actually veto but we mark in logs if a trade *would have been blocked* under the new regime. Then analyze if those trades were losers (which would confirm the filter's value). After confidence, enable actual blocking. In code, we can add a flag `overlay_shadow_mode`: if True, don't veto for real, just log a custom_event or note. Once proven, switch to active mode. This cautious approach prevents suddenly missing profitable trades if our rule was misguided.

Documentation & Maintainability:
We will update internal documentation (the README or a new doc "MultiTF_integration.md") describing how the overlay signals are generated and used, so future team members or the PM understand the logic. We'll also include the new config fields in `ensemble_config.json` (or as env overrides).

In summary, the step-by-step plan ensures the 5m strategy effectively incorporates *higher timeframe wisdom*: - Implementation points (market data adapter, guard conditions) have been identified in code. - We'll thoroughly test and gradually roll it out (shadow -> live). - This sets the stage for possibly deeper Option B integration in future if needed (we could gradually transition to a multi-arm bandit with each timeframe as an arm if we see benefit, but only after Option A is stable).

9.4 "Fastest Safe Path" (What to Do First vs Defer – Two-Person Split)

Finally, we outline the immediate next steps (fastest safe path to improvement) and delegate them between Owner A (Quant Strategist) and Owner B (Ops/Engineer). This ensures parallel progress and clear ownership. We also include the Definition of Done (DoD) for each step:

Task (Fix)	Owner	Priority	DoD (Definition of Done)
1. Raise Confidence Threshold & Low-Conf Skip *Implement CONF_MIN=0.65 and no-trade if conf<0.55, adjust unit tests.*	A	High (Week1)	- Config updated, code gating added. - All existing tests pass (update tests for new threshold logic). - Backtest results show fewer trades, improved Sharpe (documented).
2. Integrate 1h Trend Filter (Consensus Gate) *Fetch 1h data, apply guard in 5m decisions.*	B	High (Week1-2)	- 5m bot can retrieve 1h trend (via API or file) reliably. - Guard condition implemented & unit tested (e.g., forcing a veto scenario). - Dry-run log shows expected veto tags (in shadow mode).
3. Re-calibrate Model & Deploy New Calibrator *Run calibration on recent data, update model files.*	A	Med (Week2)	- Calibration script executed on validation set, new calibration parameters saved. - ModelRuntime uses updated calibrator (verified by printing a couple predictions before/after). - No regression in test (e.g., feature pipeline test still passes).

Task (Fix)	Owner	Priority	DoD (Definition of Done)
4. Activate Volatility Guard<br>*Set vol_guard_enable=True, tune sigma_cap.*	B	Med (Week2)	- Config changed, unit test for target_position with high rv passes (ensuring scale <1).<br>- Backtest of a volatile period shows position scaling kicking in (log or print showing `_last_vol_guard < 1`).<br>- No significant performance regression on calm periods.
5. Enhance Logging & Monitoring<br>*Add rejection reasons in logs, Slack alerts, and secret rotation.*	B	Med (Week1-2)	- Rejection events now include detailed reason (e.g., "veto:1h_trend").<br>- Slack webhook configured and test alert received.<br>- API keys moved out of code to env; tested connection success with new method.<br>- Documentation updated for secret injection process.

After **Tasks 1-5**, we expect a safer and slightly leaner trading system – this is the *fastest path to reducing risk* without needing lengthy retraining or new models. We prioritized threshold and trend filter first as they directly address known bad trades.

Following these, additional tasks (beyond immediate scope but next in line) include: - **6. Thompson Bandit Prior Tuning** (Owner A, Low priority): Evaluate and possibly set `model_optimism` in bandit. DoD: experiments done, if beneficial, parameter updated in config. - **7. Multi-TF Data Unification** (Owner B, next Phase): Develop unified process or file-based comm for multi-TF (to fully automate consensus from live signals, not just price trends). - **8. Performance Monitoring Dashboard** (Owner B, Medium): Use metrics_exporter to feed Prometheus and set up Grafana charts for Sharpe, drawdown, etc., for ongoing monitoring. DoD: internal dashboard shows updated KPIs in near real-time. - **9. Strategy ID Versioning & Paper Trade** (Owner A, High): Deploy the new version as `ensemble_1_1` in dry-run (paper trading) for, say, one week. Compare live paper results vs old version (some overlap maybe running old vs new in parallel on split funds). DoD: performance report after test period confirming improvements or identifying any quirks.

Conclusion: By splitting these tasks, Owner A focuses on strategy logic and model issues, while Owner B handles integration, data, and infrastructure. We start with the most impactful low-hanging fruit (confidence threshold, trend filter) to quickly cut out known bad trades – this provides immediate drawdown reduction (fastest safe improvement). Meanwhile, calibration and vol guard fortify the system's robustness with moderate effort. Multi-timeframe integration is begun in a basic form (trend filter) which is safe – deeper integration can follow once initial fixes are validated. This phased, evidence-driven approach ensures we improve the bot **safely** (no new big risks introduced) and we can **measure each change's effect** on key metrics.

Open Questions & Assumptions

- **Bandit Reward Calculation:** It's not explicitly shown how the bandit updates its arm rewards after trades. We assume it's updated via realized PnL of each decision (likely in `execution_tracker` or similar), but the exact mechanism wasn't found ("Not present in current context."). We may need to implement or adjust this. *Open Question:* Should we use binary win/loss reward or continuous PnL? We'll decide based on stability – likely binary to avoid outlier influence.
- **Higher Timeframe Model Usage:** We assumed using 1h price trend as a proxy for 1h model signal due to ease. *Question:* How much better would using the actual 1h model output be? It might catch subtler patterns. If feasible (maybe load 1h model in memory), that could improve the consensus gate. For now we proceed with price trend which is usually correlated.
- **Multi-Asset Scalability:** This plan is tailored to BTC-PERP. If tomorrow we trade ETH-PERP similarly, can the same ensemble handle it or do we spin a separate instance? Likely separate instances with same code but different STRATEGY_ID. We assume the current infra can handle that, but it might require minor refactoring to pass symbol as param throughout. Not urgent but noted.
- **LLM Copilot Scope:** We described a fairly advanced integration (queries, summaries). *Assumption:* The organization is okay with sending some strategy data to an LLM service (privacy of trading info is considered). If not, we might confine it to on-prem LLM or very high-level stats. This needs confirmation.
- **Data Quality Checks:** The plan assumes data from exchange is reliable. If there are missing candles or API hiccups, the bot might misbehave. We have a `check_data.py` script but no mention of runtime data correction. Possibly not a big issue for Binance/Hyperliquid, but an assumption is that our MarketData caching is sufficient to handle minor outages (it tries 3 times and uses cached partial results).
- **Model Stationarity:** Our fixes assume the model's patterns are still relevant (just needed calibration and filtering). If the market regime in 2026 is drastically different from training (2025), the model itself might need retraining on recent data. There are references to a `5M_RETRAINING_GUIDE.md` and model files dated Jan 2026, implying a recent retrain was done. We proceed assuming the model is fundamentally sound.
- **Confidence vs Edge:** We treat confidence (probability) as the key metric. An alternative could be *expected value in bps* (taking into account distribution of outcomes). The code has concept of `alpha = |p_up - p_down|` which correlates with expected move magnitude under certain assumptions. We assume focusing on confidence is sufficient since risk is handled separately. But if needed, we might incorporate payoff expectation (like multiply confidence by predicted move size) for thresholds.
- **Team Buy-in:** Finally, one assumption: Both owners and stakeholders agree to prioritize **risk-adjusted return** over raw return. Our changes likely sacrifice some profit in exchange for lower risk (e.g. fewer but higher-quality trades). Given the goal stated, this is acceptable, but if the business demanded aggressive growth, priorities might differ. We believe current drawdowns justify our bias toward safety.

If any of these assumptions prove false or new information emerges (e.g., bandit update logic found elsewhere, or multi-timeframe signals behave unexpectedly), we will revisit the plan and adjust accordingly. The roadmap is flexible to iterative learning – we expect to refine thresholds and maybe do a second round of tuning after initial deployment results (perhaps an Ensemble 1.2).

Evidence Index: (Key citations from code supporting our analysis)

Claim ID	Citation (File:Line)	Symbol/Context	Claim Summary
1	live_demo/main.py: 19-28	HyperliquidClientAdapter	5m bot uses Hyperliquid API if configured (data fetch logic)
2	live_demo/main.py: 46-54	Config (dry_run default)	If not specified, dry_run defaults True (paper trading by default)
3	live_demo/decision.py:15-23	Thresholds dataclass	Default thresholds: S_MIN=0.12, CONF_MIN=0.60, etc.
4	live_demo/decision.py:63-71	compute_signals_and_eligibility	Gating logic: model arms require conf>=0.60 & alpha>=0.10, cohort arms require
5	live_demo/decision.py:4-13	decide_bandit function	Bandit selection over 4 arms ('pros', 'amateurs', 'model_meta', 'model_bma')
6	live_demo/decision.py:15-23	decide_bandit (no arms)	If no arm eligible, returns dir=0 (no trade) immediately
7	live_demo/decision.py:24-30	decide_bandit (threshold)	After choosing arm, if
8	live_demo/decision.py:1-9	compute_signals (doc)	Documentation of returned signals dict and side_eps_vec thresholds tuple
9	live_demo/bandit.py:10-19	SimpleThompsonBandit	Bandit uses Gaussian Thompson Sampling with state per arm (counts, means, variances)
10	live_demo/core/config.py:9-17	get_strategy_id	Strategy ID defaults to "ensemble_1_0" if not in env (used for log tagging)
11	live_demo/log_emitter.py: 16-25	sanitize (redaction)	Log sanitizer redacts keys named "api_key", "api_secret", etc., replacing with "<REDACTED>"
12	live_demo/log_emitter.py: 125-133	emit_signals	Logs features, model output, decision, cohort each bar to signals.jsonl
13	live_demo/log_emitter.py: 159-167	emit_execution	Logs execution result and risk state to executions.jsonl

Claim ID	Citation (File:Line)	Symbol/Context	Claim Summary
14	live_demo/ config.json:2-10	Config (overlays)	5m config specifies overlays ["15m","1h"] indicating multi-TF intent
15	live_demo/ config.json:14-20	Config (API keys)	API keys and secrets are present in config file (security risk noted)