ChatGPT

Audit Findings

Incomplete & Broken Components

- **Risk Management Gaps:** The soft health breach handler in `engine/handlers/risk_handlers.py` is stubbed out [1] – it only logs and doesn't actually cancel open orders or tighten stops as intended. This needs implementation so the bot can gracefully degrade risk when a soft limit is hit. Also, the **Bybit exchange adapter** is not implemented (marked "TODO" in README [2]); the `engine_bybit` service currently uses generic engine code with no venue-specific client, so Bybit trading is non-functional.
- **Legacy Code & Duplicates:** Several outdated or placeholder modules remain from earlier milestones. For example, `ops/run_live.py` (an old M8 live trading script) contains only a TODO and is not used in the current Docker-compose flow [3]. Similarly, the `NAUTILUS_BINANCE_STARTER_PACK/strategies/hmm_policy/` directory is a deprecated skeleton (M4 placeholder) for the HMM strategy [4]; the real HMM logic now lives in `engine/strategies/policy_hmm.py`. These duplicates can confuse the codebase. All such legacy stubs (e.g. the backtest harness in `backtests/run_backtest.py` [5]) need removal or replacement with actual implementations.
- **Strategy Integration Issues:** Each strategy module varies in completeness:
- *HMM/Trend Ensemble:* The core **MA+HMM ensemble** logic is implemented (`policy_hmm.py` and `ensemble_policy.py` combine model signals with a moving-average vote [6] [7]). However, the HMM policy depends on a trained model file and uses a placeholder for price (always `px=0.0` in meta [8]). We should verify the model ingestion and perhaps update the logic to record the latest price in `meta`. The decision threshold and sizing are simplistic (fixed `quote_usdt` sizing with optional calibration multipliers [9]). There's also an older `HMMPolicyStrategy` class (M4 skeleton) that isn't wired to the engine. This old class should be deleted to avoid confusion, ensuring **all HMM signals come from the unified** `policy_hmm.decide()` **pipeline**.
- *Trend-Following:* The **trend-following module** (`TrendStrategyModule` in `trend_follow.py`) appears feature-complete. It computes multi-timeframe SMA/RSI signals, determines stop-loss and take-profit levels, and produces structured buy/sell signals with tags like `"trend_follow_long"` [10] [11]. It properly handles state (FLAT, LONG_ACTIVE, COOLDOWN) and logs entries/exits [12]. The main concern is ensuring it's fully enabled: it only runs if `TREND_ENABLED=true` in config, and its signals are executed via the strategy loop (`on_tick`). Based on the code, this integration is in place [13] [14]. We should add more unit tests or backtests for various market conditions to validate that stop/target updates and cooldowns work as expected in live trading.
- *Scalping:* A scalping module exists (`ScalpStrategyModule`) with its own tick handler and a bracket order watcher for stop-loss/take-profit exits [15] [16]. It's wired into the event bus ("trade.fill" subscriber) to monitor fills and spawn exit orders [17] [18]. We need to ensure the scalping strategy's signal generation (likely based on short-term volatility) is implemented and tested; also verify the bracket watcher threads (_start_scalp_bracket_watch) reliably place closing orders via `_execute_strategy_signal` [19]. There may be a minor race condition risk in thread handling, but overall the logic appears solid.

- *Momentum:* The real-time momentum breakout strategy (`MomentumStrategyModule` in `momentum_realtime.py`) is present and optional. It buffers tick-by-tick data and triggers when price jumps a certain percentage within a short window [20] with volume spike confirmation. It produces immediate market orders with stop-loss and trailing logic encapsulated in its returned signal dict (which includes `stop_price` and `take_profit` in meta). The main engine already enables this if configured [21] [22]. We should exercise this module in a fast market simulation to ensure it doesn't over-trigger or ignore legitimate breakouts.
- *Event-Driven Strategies:* These are mostly implemented but need end-to-end testing:
  - **Meme Coin Sentiment:** The `MemeCoinSentiment` class listens for social sentiment events (e.g. from Twitter/Reddit feeds) and applies guardrails like min social score, cooldowns, etc. [23] [24]. It sizes trades to ~0.5–1% of equity and sets predefined stop-loss/take-profit percentages. We need to verify that in a fully autonomous setup, there is a feeder that publishes sentiment events to the engine's event bus (the code expects events on topic `events.external_feed`). If such a feed connector is not yet implemented, we must create one (or stub it for now) so this strategy can actually trigger trades. The strategy's logic (parsing sentiment payloads and deciding trades) appears complete, but it hasn't been battle-tested; simulated "meme coin pump" events should be run through to validate the priority and velocity thresholds.
  - **Listing Sniper:** This module (`ListingSniper`) is designed to react to new Binance listing announcements in real-time. It is fully implemented and quite comprehensive: it parses announcement events (title, go-live time, symbols) [25] [26], enforces one-shot per symbol (uses `_seen_ids` to avoid duplicates) [27], and then spawns an async task to snipe the listing [28]. The sniping logic waits through any preset delay, monitors price and spread in the opening minutes, and either skips if conditions aren't favorable or executes a market BUY with controlled sizing [29] [30]. It then immediately places a stop-market and staggered take-profit limit orders [31] [32] to manage the new position. This logic is sound. We must ensure the **event source** for this strategy is active: likely the "universe" or "situations" service should scrape Binance's announcements feed and publish an event (with `source="binance_announcements"`). If that connection is not currently wired, implementing a small monitor (or using Binance's official announcement RSS) to feed events into the bus will make this strategy truly autonomous.
  - **Airdrop Promotions:** The `AirdropPromoWatcher` monitors exchange news for trading competitions or airdrop campaigns. It parses text for keywords (regexes for "trade at least $X", etc.) [33] [34] to estimate required volume and potential reward, and then decides whether to participate by placing a qualifying trade. The code is implemented with risk controls (e.g., only if expected reward $\geq$ threshold, spread < max, etc.) and uses `router.market_quote` to execute trades, similar to the listing sniper. As with the other event strategies, the missing piece is ensuring the source data (promotion announcements) are fed in. We should integrate this with the same feed that handles listing announcements or a separate scraper for Binance promo pages.
  - **Event Breakout:** An event-driven breakout module (`engine/strategies/event_breakout.py` with trailing logic in `event_breakout_trail.py`) is noted in the README [35]. This likely listens for large price displacement events (maybe from the "situations" pattern-matcher service) and then executes a breakout trade with a trailing stop. We need to verify its integration: presumably the `situations` microservice emits an event when a breakout pattern is detected, and the engine's strategy component picks it up. If any wiring is incomplete here (e.g., subscribing to the event bus topic), that should be added. The

trailing stop part (`event_breakout_trail.py`) should be reviewed to ensure it correctly tracks the price and exits at the optimal point.

Data Ingestion & Execution Pipeline

- **Real-Time Data Feed:** Live market data ingestion is handled via `BinanceWS` (WebSockets). This component is designed to automatically reconnect with a backoff on disconnect [36] [37]. However, the current implementation resets the backoff to 1s after a successful connection and doesn't progressively increase the delay on successive failures (the `backoff` variable is never incremented beyond 1.0 in the loop) [38] [39]. We should enhance this with an exponential backoff or jittered retry to be gentler on the API during outages. Aside from that, the WS client is robust: it chunks subscriptions to avoid URL limits, handles futures vs. spot streams (mark price vs. mini-ticker) [40] [41], and updates shared price gauges and invokes the strategy tick callback for each tick [42] [43]. We will want to test how it behaves under network blips (simulate a disconnect and ensure it reconnects and continues streaming).
- **Order Execution & Routing:** The execution pipeline is generally strong. The `RiskRails.check_order()` is called before any live order placement to enforce notional limits, exposure caps, etc. [44] [45]. The system supports **spot, margin, and futures** markets on Binance: environment flags like `BINANCE_MARGIN_ENABLED` bring up a `BinanceMarginREST` client (for isolated margin trading) [46], and `BinanceREST` itself can handle all markets by choosing the appropriate endpoint per the `market` parameter [47] [48]. The code path for order execution in strategy: if not in dry-run, `_execute_strategy_signal` ultimately calls `order_router_instance.place_market_order(...)` with the resolved market (spot/margin/futures) [49]. Additionally, the `resolve_market_choice()` utility will map symbols to a preferred market (via `MARKET_ROUTE_MAP` config or defaults) [50] [51]. We should thoroughly test scenarios for each market type: e.g., ensure that if `prefer_futures=true` for momentum trades, the orders actually go through the futures API (and that the account has permissions). Similarly test margin mode with `AUTO_BORROW_REPAY` (the code sets `sideEffectType` appropriately for margin orders [52]). So far, the code suggests all routes are supported, but without real testing these code paths might hide bugs (like lot size handling differences).
- **Portfolio Persistence & Reconciliation:** The bot uses an in-memory portfolio with SQLite-backed persistence. Every fill results in a snapshot save (JSON to disk via `SnapshotStore`) and a database insert [53] [54]. On restart, it calls `reconcile_since_snapshot()` to pull any missed trades from the exchange and apply them to the portfolio state [55] [56]. This is crucial for restart safety so the bot resumes with correct position sizing and PnL. The reconciliation code is implemented for Binance (parsing `myTrades` results) [57] [58]. We should test a restart scenario: start the bot, execute some trades, shut it down, then restart and verify it correctly loads historical fills and sets the state (the equity should match exchange account equity).
- **Error Handling & Resiliency:** Throughout the engine, errors are generally caught and handled to avoid crashes. For instance, `on_tick` wraps each strategy call in a try/except to ensure one failing strategy doesn't stop the loop [59] [60]. Similarly, the WS loop continues on JSON parse errors or missing fields [61]. One area to improve is logging these suppressed exceptions: currently many `except Exception: pass` are in place (e.g., in metric updates or callbacks [62]). We might want to at least log these at DEBUG level to facilitate troubleshooting without halting the system.
- **Logging & Observability:** The system has extensive metrics via Prometheus (every service exposes `/metrics`). Key strategy metrics include `strategy_ticks_total`, latency, and per-strategy order counts [63] [64]. Logging is done to files (mounted in `engine/logs/`) and to console. We

should ensure the logging configuration captures all critical events (e.g., a trade execution failure in ListingSniper logs a warning [65], risk breaker triggers log warnings). The observability stack (Prometheus, Grafana, Loki) is optional but highly useful; we'll verify the provided Grafana dashboards for KPIs (like `event_breakout_kpis.json` and `slippage_heatmap.json`) are receiving data.

Roadmap to Full Autonomy

1. Unify & Stabilize the Codebase

- **Merge and Remove Duplicates:** Eliminate legacy placeholders that are no longer used. For example, delete or archive the old `strategies/hmm_policy` skeleton module and `ops/run_live.py` script, which have been superseded by the engine's integrated scheduler [3]. This prevents confusion and ensures we have a single source of truth for each strategy.
- **Consolidate Strategy Definitions:** Make sure all active strategies are defined in the unified `engine/strategies/` package and are instantiated through the main engine (via config flags). Double-check that environment flags in `.env` or `docker-compose.yml` correspond to the new implementations. For instance, `STRATEGY_ENABLED` / `STRATEGY_DRY_RUN` should govern the MA/HMM ensemble logic in `engine/strategy.py` (which it currently does [66]), and newer flags like `TREND_ENABLED`, `SCALP_ENABLED`, etc., toggle the respective modules. Update `docs/FEATURE_FLAGS.md` to list all strategy toggles and their effects so it's clear which modules will run.
- **Ensure Consistent Event Handling:** Audit how external event data enters the system and is routed. We likely have separate pathways for tick data (via `on_tick`) and for off-tick events (via the `event_bus` or direct calls to strategy classes). For consistency, use the engine's `EventBus` (`engine.core.event_bus.BUS`) to publish external signals into the engine. For example, when the **Universe/Screener** service detects a new Binance listing, it should publish an event like `{"source": "binance_announcements", "payload": {...}}` that the `ListingSniper.on_external_event()` is listening for [67]. We will implement or fix these bridges: subscribe the ListingSniper, MemeSentiment, and AirdropPromo modules to the appropriate bus topics during engine startup (in `engine.app:startup` or similar). This might involve writing small handler functions that take an event dict and call `listing_sniper.on_external_event(event)`, etc., and doing `BUS.subscribe("events.external_feed", meme_sentiment.on_event)` and so on.
- **Stabilize Config & Defaults:** Unify configuration loading for strategies. Currently, each module has its own `load_*_config()` from env variables (e.g., `load_trend_config()` [68], `load_meme_coin_config()` [69]). This is fine, but ensure that every important knob is exposed in the `.env` template with sane defaults. Cross-check that critical risk parameters (like `MAX_NOTIONAL_USDT`, `EXPOSURE_CAP_*`) are set in `.env` and honored by RiskRails. We'll also lock down any inconsistent naming (for example, `TRADE_SYMBOLS` vs `TREND_SYMBOLS` – the trend loader checks both [70]). Simplify where possible: e.g., use one unified symbol list for all systematic strategies if appropriate, or clearly document how to specify separate lists.
- **Update Documentation:** Revise the README and docs to match the unified design. The README's "Strategy & Modules" section already describes most components; we will ensure it no longer references removed placeholders or planned features that aren't implemented. Also, add a high-level diagram in the docs if not present to illustrate data flow: e.g., "WebSocket -> on_tick -> Trend/Scalp/Momentum -> StrategySignal -> OrderRouter, and External Feed -> Event Strategy -> OrderRouter." This helps any developer or operator understand the live flow after our unification.

2. Clean Out Unused Legacy Logic

- **Strip Out Dead Code Paths:** Go through the codebase and remove or refactor blocks marked with old milestone comments (M3, M4, etc.) that were never completed. For example, the backtest harness `run_backtest.py` is a stub [5] ; we will either implement it (see step 5) or remove it for now to avoid any assumption that backtesting is ready. Similarly, if there are old experiment scripts in `ops/` or `scripts/` that interface with now-removed APIs, purge them or update them to the new engine API. This cleanup will reduce confusion and potential misuse.
- **Eliminate Duplicated Config/Logic:** Determine if any functionality is implemented twice. The **HMM strategy** was one candidate (old vs new); another could be "symbol universe management." If both `TREND_SYMBOLS` and the new `SYMBOL_SCANNER` exist: the scanner dynamically picks symbols based on momentum/ATR filters [71] , whereas TREND_SYMBOLS is a static list. To avoid conflict, we'll enforce that if `SYMBOL_SCANNER_ENABLED=true` , the static list is ignored (which is already implied by code). Document this behavior and ensure only one method is active at a time. Another duplicate area might be order execution: we see both `router.market_quote` being used directly in event strategies [72] and the `_execute_strategy_signal` path used in on_tick strategies [73] . While both ultimately call the order router, using one unified interface is preferable. We might refactor event strategies to also use `_execute_strategy_signal` for consistency – this would automatically leverage idempotency keys and risk checks uniformly. If we do that, ensure the event strategies pass a `dry_run` flag appropriately (some currently call RiskRails.check_order manually and then `router.market_quote`). Aligning these will make maintenance easier.
- **Refactor Legacy Patterns:** Some older patterns can be modernized. For instance, thread-based loops (like the scalp bracket watcher spawning threads) could possibly be converted to asyncio tasks now that the engine is async. If time permits, refactor `_start_scalp_bracket_watch` to use `asyncio.create_task` and sleeps instead of a dedicated `threading.Thread` , to avoid potential threading issues (the rest of the engine is largely async). This isn't critical, but it's a cleanup that aligns with the overall architecture (the engine uses FastAPI + async for everything else). Similarly, any global state caches (like the way `ensemble_policy._last_ts` dict is used for cooldowns) might be relocated into a more formal state tracker, but only if it simplifies the design.

3. Implement Missing or Broken Functionality

- **Risk Soft-Breach Handler:** Complete the logic in `on_cross_health_soft` . When a soft risk threshold (like high error rate or minor equity drawdown) is crossed, the system should take protective actions short of full shutdown. We will implement code to: fetch all open orders (perhaps via the portfolio or order router), cancel any open entry orders (while possibly keeping stop-loss orders active), and tighten stop-losses on open positions. For example, we can integrate with `engine.ops.bracket_governor` or directly call the order router's cancel endpoint for each open order tagged as "entry". Logging will be kept at INFO level for these actions (so the operator knows the bot is proactively cutting risk) [1] . This ensures the bot can recover from a false alarm without human intervention. After implementing, test by artificially triggering a soft breach condition (e.g., set a low exposure cap and attempt an order) and observing that the handler indeed cancels orders.
- **Bybit Exchange Support:** Develop a `BybitREST` client (perhaps by subclassing or modeling after `BinanceREST`) and a connector in `engine/core/` that can submit orders to Bybit's API. At minimum, implement market order and price fetch functionality so that the `engine_bybit` container can execute basic strategies. Since Bybit was marked as TODO, we likely will initially support spot trading on Bybit (or futures if needed) with a single symbol set. This includes adding

Bybit API keys in the `.env` and wiring up environment flags (e.g., `BYBIT_ENABLED=true` to spawn that service). If a full implementation is too large, consider falling back to disabling Bybit in production until ready, but the roadmap target is full autonomy on all supported venues, so implementing Bybit is desirable. We'll mirror the risk controls for Bybit as well (exposure caps per venue are already parameterized by venue in RiskRails [74] [75]).

- **IBKR & Others:** The README mentions IBKR (Interactive Brokers) support as optional [76]. If this is intended to be part of the autonomous stack, verify that `engine/connectors/ibkr_client.py` exists and can connect to TWS. If it's incomplete or untested, mark it clearly as beta or disable by default, to avoid any blocking issues in our 24/7 run. Focus on solidifying Binance and Kraken (which are actively supported); IBKR can be a future enhancement unless the user explicitly needs it now.
- **Feed Connectors for Event Strategies:** As noted, the event-driven strategies need data feeds. Implement lightweight connectors:
  - *Binance announcements:* Use Binance's official announcements RSS feed or web scraping to detect new listings and promotions. This can run as part of the `universe` service or a standalone daemon. Upon finding a new listing, construct the event dict as expected by `ListingSniper` (with `payload.title`, `id`, `publishTime`, etc. matching the fields parsed in code [25]) and push it onto the engine's event bus. This should be done frequently (Binance listings are announced unpredictably; polling every 1–5 seconds when `LISTING_SNIPER_ENABLED` is true might be acceptable). Test this by simulating an announcement and ensuring the sniper fires a trade.
  - *Social sentiment:* Hook into a Twitter API or sentiment service for meme coins. This could be more involved; as a stop-gap, we might consume events from popular crypto tweet aggregators or use a saved dataset of events for testing. The strategy expects fields like `score, mentions, velocity, sentiment` in the payload [77]. We can either provide a custom integration that calculates these (e.g., from LunarCrush or Santiment APIs) or simplify the approach by having a script that monitors specific Twitter accounts and assigns a heuristic score. Given the complexity, this connector can start simple: e.g., listen for any mention of configured tickers on Twitter with certain keywords and then trigger `MemeCoinSentiment` with a fabricated score above thresholds to simulate a pump. Over time, refine this with actual sentiment analysis.
  - *Other event feeds:* For the airdrop promo, monitor Binance's "announcement" section for keywords (the Binance announcements feed includes promos too). Also consider CoinMarketCal or similar for market-moving events, if relevant. These integrations ensure no human is needed to tell the bot about opportunities – it will discover and act on them automatically.
- **Finalize Strategy Action Pipeline:** Ensure that for *every* strategy, a generated signal leads to an execution. For systematic strategies (trend, scalp, momentum, ensemble), this is achieved via `on_tick -> StrategySignal -> order execution` and looks correct in code [78] [79]. For event strategies, some currently call the order router directly. We will standardize this by perhaps wrapping event-driven orders in the same `StrategySignal` object and calling `_execute_strategy_signal`. For example, when ListingSniper decides to buy, instead of directly doing `router.market_quote`, we can construct `sig = StrategySignal(symbol="XYZUSDT.BINANCE", side="BUY", quote=notional, tag="listing_sniper", ...)` and post it to the `/strategy/signal` endpoint internally. This way, we automatically get idempotency and post-trade logging (the `orders.jsonl`) for those trades as well [80] [81]. If we do this refactor, we must still handle immediate stop-loss placement – one approach is to extend `StrategySignal.meta` to carry the desired stop/TP and have the

execution path set those after the market order fills. Alternatively, we can leave the post-order stop placement as is (since it happens after the market order returns) but still unify the initial entry via StrategySignal. The result will be a cleaner, single funnel for all orders.

4. End-to-End Testing of Signal-to-Trade Workflow

- **Backtest and Simulations:** Introduce a systematic way to test each strategy on historical data to ensure the full pipeline works without exceptions. We will implement a simple **backtesting harness** (replacing the M4 placeholder in `backtests/run_backtest.py`). The plan is to allow feeding historical price data into the strategy modules and simulating the order execution offline:
  - Create a minimal **BacktestEngine** class that can load a CSV or Parquet of historical ticks/kline data for a symbol and iterate through it, calling the same `strategy.on_tick(symbol, price, ts)` that the live engine would. We can use the existing strategy modules with a slight tweak: in backtest mode, we'll substitute the real OrderRouter with a dummy that records orders instead of sending to an exchange. This could be as simple as monkey-patching `_execute_strategy_signal` to log the would-be order to a list, or using the `dry_run=True` mode globally.
  - Focus first on **Trend** and **Ensemble HMM** backtesting: these are systematic and easier to simulate. Feed them recent 1-minute data for BTCUSDT and verify that buy/sell signals occur in plausible spots (e.g., after a golden cross for trend, or when HMM confidence > threshold). Check that the cooldown logic actually prevents over-trading (the metrics `strategy_cooldown_window_seconds` can be observed or we just inspect timestamps of signals).
  - Expand backtesting to scalping and momentum by simulating a volatile period and confirming that multiple quick trades are opened and closed. Scalping might require simulating partial fills or immediate fills to test the bracket stop logic – we can simulate a fill event to trigger `_scalp_fill_handler` and see that it schedules an exit order [82].
  - Use these backtests to identify any bugs (e.g., division by zero, missed attribute, etc.) in the strategy code when faced with real data edge cases (like flat price periods, gaps, etc.), and fix them.
- **Paper Trading Dry-Run:** Before live deployment, run the entire system in **dry-run mode (paper trading)** for an extended period (several days) on live data. All strategies should run with `*_DRY_RUN=true` so that no real orders are sent, but the logic and metrics operate as if trading. Monitor the logs and Prometheus metrics:
  - Check that no exceptions are being thrown repeatedly. If any strategy continuously errors out (e.g., due to unexpected data shape), address that.
  - Confirm the strategies are generating signals at a reasonable rate. For instance, if trend-following is configured on 5 symbols, we expect to see a handful of trend signals per week, not hundreds per day (unless markets are extremely trending). If it's overactive, we may need to tighten criteria or found a bug in signal generation.
  - Observe metric `strategy_orders_total` labeled by source [83] [84] to ensure each strategy actually attempts trades. If a strategy is enabled but `strategy_orders_total{source="..."}=0` over a long window, then it isn't firing at all – investigate why (maybe misconfiguration or its conditions never met).
  - Use the Prometheus metrics to compute theoretical PnL for dry-run trades. In dry-run, orders are "simulated" and logged to `orders.jsonl` [85]. We can post-process these logs to

reconstruct trades and estimate PnL (since we have fill prices in the log). This will give us a rough performance baseline for each strategy.
    - Particularly test **multi-strategy interaction**: e.g., enable Trend, HMM ensemble, and Scalping together (with dry_run) to see if they ever conflict (do they try to trade the same symbol in opposite directions?) or if risk management correctly handles multiple signals at once. The risk module should reject an order if another just filled and exposure caps would be exceeded [86] [87]; verify such rejections appear in logs and are handled gracefully (they should show "rejected" status with an error code in the strategy response).
- **Full Live Test (Small Scale):** Once dry-run results are satisfactory, do a limited live test with real funds on Binance (perhaps on testnet or with minimal notional). Start with one systematic strategy (trend-following on a single symbol with tiny size) and one event strategy (maybe listing sniper in testnet if possible). Carefully monitor:
    - Execution correctness: orders go out to the exchange and appear in the exchange order book. Fills are recorded in SQLite (`engine/state/orders.db` or similar) and reflected in metrics.
    - Timing: measure `strategy_tick_to_order_latency_ms` [63] – it should be consistently <100ms as targeted, meaning our pipeline is fast. If we see higher latency, profile where the delay is (WS delay, model prediction time, etc.) and optimize (e.g., reduce model size or increase concurrency).
    - Recovery: intentionally restart the engine containers to ensure the persistence and reconciliation work (positions should not double-count and trading resumes correctly after a short pause). Also, simulate network outages or Binance API errors – e.g., kill the internet connection for a minute. The WS should reconnect and the rest client should handle transient failures (our HTTP calls already retry on 429/418 with small backoff [88] [89]). Confirm the bot doesn't crash and continues when connectivity returns.

5. Enhance Reporting, Backtesting, and Analytics

- **Robust Backtesting Framework:** Expand the backtest harness into a reusable tool. This can be developed into a `BacktestEngine` class that can simulate **full portfolio and risk logic**. For instance, feed it historical tick data and let it use the real `RiskRails` (perhaps in a mode that doesn't actually connect to exchange, but simulates fills immediately at price). The objective is to produce **performance metrics** for each strategy: total return, max drawdown, Sharpe ratio, win rate, etc., over historical periods. We will output standardized reports (CSV or JSON) in `data/backtests/` for analysis. These reports will guide parameter tuning (for example, the trend strategy's auto-tuner can be validated by seeing how it would have adjusted parameters historically).
- **Daily PnL and Metrics Reports:** Implement an **end-of-day summary** routine. This could be a simple cron-like async task in the ops service or a separate script run via Docker (perhaps triggered by `make autopilot-status`). The summary should compile the day's trading results: realized PnL, unrealized PnL, number of trades, win rate for the day, and maybe cumulative stats. We can get most of this from the Prometheus metrics (e.g., `trade.fill` events and the portfolio equity change). For reliability, we might directly query the SQLite fills table for trades in the last 24h and aggregate profit/loss. The report can be logged and also sent via notification (email/Telegram). This gives the human operator a concise view of performance without digging through Grafana.
- **PnL Attribution & Logging:** Augment the logging around trade executions to include PnL context. For example, after an exit order is executed, log the P&L for that round-trip trade (we have the entry and exit price and quantity, so compute profit%). Some of this is already in Trend module (it logs

`pnl_pct` on exit [90]). We can extend that approach to other strategies – e.g., when ListingSniper sells at take-profit, log the ROI. These logs, combined with a daily roll-up, will help identify which strategies are contributing most to profits or losses.

- **Metrics Enhancements:** Add custom Prometheus metrics for key financial outcomes. We can add a gauge for daily realized PnL (reset each day) and a counter for cumulative realized PnL. Perhaps use the existing `record_realized_total` from the deck publisher [91] to track total profits. In addition, metrics like `strategy_win_rate{strategy=XYZ}` could be calculated and exposed – though not trivial to maintain state in metrics, we might update such values at the end of each trade or day. The goal is to make the Grafana dashboard not just about operations (latency, counts) but also about performance (returns, drawdowns).
- **Backtest vs Live Parity:** Ensure that any divergence between backtest and live is minimized. Because our strategies rely on real-time streaming and event ordering, backtest results should closely resemble what live would have done given the same data. We'll validate this by replaying recent periods through the backtester and comparing to actual live trades taken (once the system has been running). If discrepancies are found, adjust the backtest logic (or strategy code if necessary) so that we can confidently use backtesting for strategy validation going forward.

6. Deployment Hardening and Alerts

- **Docker Deployment Review:** The Docker setup already isolates each service and uses a shared network. We will check container resource usage and consider adding resource limits in `docker-compose.yml` (e.g., memory limits for each container to prevent any runaway memory from crashing the host, and CPU shares if needed). The engine containers run as non-root (using `appuser`) which is good for security [92]. We'll also implement a **restart policy** in docker-compose (e.g., `restart: always` for critical containers), so that if a container crashes, Docker will automatically attempt to bring it back. This covers unexpected failures without human intervention.
- **Fail-safe Modes:** Introduce additional safety fallbacks for truly unexpected scenarios. For example, if the bot's equity drops below a very hard floor (e.g., 50% drawdown), we may want the system to automatically halt trading (set `TRADING_ENABLED=false`) and send an urgent alert. RiskRails does trigger a breaker on floor breach [93] [94], which flips `_equity_breaker_active` and causes further orders to be rejected [95]. We will extend that by writing a flag (as done in critical breach handler) and perhaps having the ops service watch that flag to send an alert. Another scenario: if the **health state** (a metric combining error rates, etc.) goes critical, the system could switch to a passive mode until recovery – this could be as simple as pausing new entries but managing exits normally. We'll formalize such modes and ensure they can be resolved automatically or with minimal input (e.g., auto-resume after X minutes if conditions improve, or require manual toggle via a documented procedure in `OPS_RUNBOOK.md`).
- **Alerting Integration:** Set up **Telegram and/or Email alerts** for important events. The codebase has a Telegram notifier utility (the Stop Validator mentions an optional Telegram ping for missing SLs [96]). We will configure this with a bot token and chat ID in the `.env`, and test sending messages. Key triggers for alerts: - Risk critical breach (trading halted by breaker) – send immediate Telegram alert with the reason (floor or drawdown) and the fact that trading is disabled [97]. Include instructions or an automatic note if the system will try to re-enable after cooldown or not. - Daily PnL report – as discussed, send the summary each day to the operator's email or chat. - Exception notifications – if an unexpected exception bubbles up (we might instrument a last-resort exception handler in the FastAPI app or a background task that monitors logs for "ERROR" entries), send an alert that manual intervention might be needed. For example, if the WS connection fails

continuously for 5 minutes (no ticks coming in), alert that the data feed is down. - Strategy-specific alerts – e.g., listing sniper could send a Telegram message when it participates in a new listing ("Bought XYZ on new listing at $10, stop $8, target $12"). This is not only informative but also serves as a sanity check that the strategy is doing something. We can add a hook in the ListingSniper after a successful buy to publish such a message (the code can call a simple alert function if TELEGRAM_ENABLED).

- **Final Dry Run and Go-Live:** Before going fully live 24/7, do one more dry-run with all the above changes to ensure no regressions were introduced (especially by refactoring the order execution path for event strategies and adding new threads/tasks). Once stable, deploy the updated Docker stack with real API keys and **monitor it closely for the first few days** with the alerting in place. We will use the Grafana dashboards and our new daily reports to verify that the bot is autonomous: it is finding opportunities (e.g., taking some trades every day across strategies), managing risk on its own (we should see risk metrics like `venue_exposure_usd` and `risk_equity_drawdown_pct` [98] [99] stay within bounds), and recovering from any minor errors without intervention.

By following this structured roadmap – starting with codebase cleanup and proceeding through targeted fixes, thorough testing, and deployment hardening – we will transform the repository into a production-grade, fully autonomous trading bot. Each strategy module will have a clear path from signal generation to order execution, all major failure modes will be handled internally, and the system will actively inform us of its status, truly achieving hands-free 24/7 operation.

Sources: Implementation details and code references were derived from the repository `mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK` – e.g., risk handler stubs [1], strategy logic in trend [100] [11] and HMM ensemble [101] [66], and Docker/compose configuration for persistence and services [102] [103]. The roadmap steps address these findings in sequence, ensuring completeness and reliability.

---

1 97 risk_handlers.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/handlers/risk_handlers.py

2 35 63 71 76 96 README.md
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/README.md

3 run_live.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/ops/run_live.py

4 strategy.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/NAUTILUS_BINANCE_STARTER_PACK/strategies/hmm_policy/strategy.py

5 run_backtest.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/NAUTILUS_BINANCE_STARTER_PACK/backtests/run_backtest.py

6 7 13 14 15 16 17 18 19 21 22 44 45 49 53 59 60 64 66 73 78 79 80 81 82 83 84 85 101 strategy.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/strategy.py

8 9 policy_hmm.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/strategies/policy_hmm.py

10 11 12 68 70 90 100 trend_follow.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/strategies/trend_follow.py

20 momentum_realtime.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/strategies/momentum_realtime.py

23 24 69 77 meme_coin_sentiment.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/strategies/meme_coin_sentiment.py

25 26 27 28 29 30 31 32 65 67 72 listing_sniper.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/strategies/listing_sniper.py

33 34 airdrop_promo.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/strategies/airdrop_promo.py

36 37 38 39 40 41 42 43 61 62 binance_ws.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/core/binance_ws.py

46 91 app.py
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/app.py

47 48 52 88 89 binance.py

https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/core/binance.py

50 51 market_resolver.py

https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/core/market_resolver.py

54 55 56 57 58 reconcile.py

https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/reconcile.py

74 75 86 87 93 94 95 98 99 risk.py

https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/engine/risk.py

92 Dockerfile

https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/Dockerfile

102 103 docker-compose.yml

https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/32bcdf9f732b732400a94bf72eae9cde186a4b1e/docker-compose.yml