ChatGPT

Repository Audit and System Architecture Overview

The **Nautilus HMM Starter Pack** repository implements a comprehensive, modular crypto trading bot system with multiple coordinated components. Below we break down each part of the codebase – from data ingestion and strategy logic to order execution, portfolio management, and the integration with a front-end control UI – and assess how close the system is to its goal of fully autonomous, consistently profitable trading. We also identify any legacy or incomplete elements and provide a step-by-step roadmap for reaching the project's "complete autonomy" and profit goals.

System Components and Service Topology

Multi-Service Architecture: The project is organized into several services (microservices/containers), each with a specific responsibility, all working in concert. Key components include:

- **Engine services (Execution Engines):** FastAPI-based trading engines for each exchange venue (e.g. Binance, Kraken, IBKR) running from a unified codebase (`engine/app.py`). Each engine exposes REST APIs for order execution (`/orders` endpoints), metrics, and health checks [1] [2]. For example, there are separate containers like `hmm_engine_binance` (port 8003) and `hmm_engine_kraken` (8006), each running `engine/app.py` with exchange-specific adapters [3]. The engine also has a paired "exporter" mode (read-only replica) for scraping metrics without sending orders [4].

- **Ops service (Control Plane API):** The `ops/ops_api.py` service (container `hmm_ops` on port 8002) acts as the **central control and aggregation layer** [5]. It collects metrics from all engines, provides governance endpoints, and exposes a unified API for higher-level operations (such as strategy orchestration and capital management). This Ops API includes endpoints for status, toggling trading, strategy management, and broadcasting events via WebSockets or Server-Sent Events [6] [7]. It is essentially the "brain" that oversees the execution engines.

- **Supporting microservices:** Additional FastAPI services support specific dynamic functions:

- **Universe Service (`universe/service.py`):** Maintains and updates the list of tradable symbols, e.g. fetching exchange info and top-volume symbols. It provides a `/universe` endpoint returning categorized symbol sets and updates metrics on symbol bucket sizes [8] [9]. This helps the system **dynamically select which symbols to trade** based on market conditions (e.g. newly listed coins or trending assets).
- **Situations Service (`situations/service.py`):** A pattern-matching service for complex event detection (e.g. detecting certain market patterns or "situations"). It can receive external feature payloads and evaluate them against rules, emitting events/metrics when patterns trigger. This feeds into event-driven strategies.

- **Screener Service (`screener/service.py`):** Computes alpha signals by scanning market data (like historical klines or order book info) for opportunities. It forwards any detected "hits" to the Situations service and also exposes metrics [10]. In essence, Universe, Screener, and Situations work together to **adapt the trading universe and identify opportunities** in real time, feeding those into the strategy layer.

- **Observability stack:** Containers for Prometheus (`hmm_prometheus`), Grafana (`hmm_grafana`), Loki/Promtail (logging) provide monitoring and logging infrastructure [11]. The engines and ops service expose metrics endpoints (Prometheus format), which these tools scrape. Grafana dashboards (e.g. a "Command Center" dashboard) are set up to visualize PnL, latency, fill rates, etc., and alert rules are defined for abnormal conditions [12] [13].

All services are designed to communicate over a common Docker network (as noted in the setup instructions) [14]. The **service topology** ensures a clear separation of concerns: *execution engines* handle low-level trading per exchange, the *ops service* handles cross-engine coordination and governance, and *support services* provide market intelligence. This modular design improves maintainability and allows scaling or extending each piece independently (for example, adding a new exchange adapter or new strategy service in the future) [1].

Data Flow: From Market Data to Trade Execution

Market Data Ingestion: The engines subscribe to market data (price ticks, order book updates) via exchange WebSocket or REST feeds [15]. Incoming tick data is published into the engine's event bus and fed to the strategy logic. In the unified data-flow diagram, exchange feeds produce `market.tick` events that flow into the **Systematic strategy modules** inside the engine [15] [16]. External signals (like outputs from the Universe/Screener services or webhooks for special events) are injected via the engine's event bus as well (e.g. posted to an `/events/external` endpoint, which then routes to event-driven strategies) [16]. This design means the bot can react both to continuous market ticks and to asynchronous external events.

Strategy Signal Generation: The **strategy layer** can produce trading signals in two ways: 1. **Built-in Systematic Strategies (continuous):** The engine has an internal scheduler/strategy loop (`engine/strategy.py`) that runs strategies like a moving-average crossover combined with an HMM (Hidden Markov Model) predictor and others (trend-following, scalping, momentum) if enabled [17]. On each tick or at set intervals, these strategies compute buy/sell signals for allowed symbols. For example, the default strategy combines a simple MA crossover with an HMM-based regime model – prices are fetched, moving averages updated, and the HMM model (loaded from `engine/strategies/policy_hmm.py`) generates a probability-based signal; these are fused into an ensemble decision [18]. Additional modules like trend-following (`engine/strategies/trend_follow.py`) can run if their feature flags (`TREND_ENABLED`, etc.) are true [19] [20]. Each strategy has its own configuration section and can be toggled or tuned via environment variables (as noted by the numerous `TREND_*`, `SCALP_*`, etc. settings) [21] [22].

1. **Event-Driven and External Strategies:** These are reactive strategies triggered by discrete events. Examples mentioned include a *Listing Sniper* (to detect and trade newly listed assets quickly) and a *Meme Sentiment* strategy (perhaps trading based on social media sentiment signals) [23]. Such strategies don't run on every tick but wait for specific triggers – for instance, the Universe service might detect a new coin listing and emit an event that the engine's event bus picks up, activating the

listing sniper logic to place a trade. Similarly, external machine learning models or scripts can call the engine's API to post signals (the engine provides an `/events/external` hook for this) [16].

All generated **signals, regardless of source, are funneled into a unified execution path**. The system's design dictates that **every order signal passes through risk controls before execution** [24] [25]. Concretely, when a strategy (internal or external) wants to execute a trade, it calls a function that ultimately hits the engine's REST API (like an internal call to `post_strategy_signal` which routes to the `/orders` endpoint) [26]. This ensures even internally generated orders go through the same pipeline as external API requests.

Risk Management (Pre-Trade Checks): Once a trading signal is generated, the **RiskRails** module acts as a gatekeeper to enforce all risk constraints **before any order is sent to an exchange** [27] [25]. The method `RiskRails.check_order` (defined in `engine/risk.py`) validates that trading is enabled and then checks a series of policy rules: - *Toggle & Allowlist:* It verifies a global trading kill-switch (`TRADING_ENABLED`) and that the symbol is in the allowed trading list (`TRADE_SYMBOLS`) [27]. - *Notional and Exposure Limits:* It checks the order size against `MIN_NOTIONAL_USDT` (minimum trade size) and any caps on exposure per symbol or total (e.g. `RISK_LIMIT_USD_PER_SYMBOL` which defaults to 1000 USD) [28] [29]. - *Rate Limits:* Ensures the bot doesn't send too many orders too quickly (`MAX_ORDERS_PER_MIN`, etc., configurable in policies). - *Circuit Breakers:* Monitors exchange error rates and can halt trading on a venue if too many errors or failed orders occur (using metrics like `venue_error_rate_pct` and counters for rejections) [30]. - *Drawdown/Equity Floor:* If configured, stops trading if equity falls below a threshold or a max drawdown is hit (to prevent catastrophic losses).

Only if an order passes all these checks does it proceed. These risk configurations are all dynamic via environment variables or policy files, so they can be adjusted without code changes [31]. This layered risk mechanism exemplifies the **"safety before speed" principle** guiding the system [32] – protecting capital is prioritized.

Order Routing and Execution: After RiskRails approval, the order goes to the **Order Router**. The `OrderRouterExt` class (in `engine/core/order_router.py`) normalizes the order (ensuring the symbol, side, quantity meet the target venue's format and minimums) and then forwards it to the appropriate exchange client adapter [27]. The engine supports: - **Binance** via a `BinanceREST` client (for spot and futures, using API keys from env). - **Kraken Futures** via `KrakenREST` (with separate API keys and using USD quotes). - **IBKR (Interactive Brokers)** via an `IbkrClient` (to trade equities; requires a running IB TWS/Gateway, and handles unit conversion of quote currency internally) [33]. - **Bybit** – currently a stub: the infrastructure is in place (there's an `hmm_engine_bybit` container), but it currently reuses the generic Binance client logic. A proper Bybit REST/WebSocket adapter is **to be implemented** (marked TODO) [34]. This is one known gap – the code acknowledges that a dedicated Bybit adapter is **still pending** [35].

The OrderRouterExt picks the client based on the symbol or configuration, then submits the order via that client's API. If one engine process handles multiple venues, the `ENGINE_ENDPOINTS` list (configurable in `.env`) allows the ops service or strategy router to know all active engine base URLs [36]. For example, `ENGINE_ENDPOINTS` might include `http://engine_binance:8003,http://engine_kraken:8006` so that higher-level components know where to send orders.

Post-Trade: Portfolio and Persistence: When an exchange fills an order (immediately for market orders, or later for limit orders), the engine receives a fill update. The in-memory **Portfolio** module (`engine/core/`

`portfolio.py`) then applies the fill: it will adjust the position size for that symbol, update cash balance (debit for buys, credit for sells), update realized and unrealized PnL, etc. [37] [38]. This portfolio state is considered the source of truth for the bot's current holdings across all symbols on that venue. Two mechanisms ensure this state is robust: - A **SnapshotStore** periodically saves the entire portfolio state to disk (and on engine shutdown) so that if the engine restarts, it can restore positions and balances exactly [39]. - All orders and fills are also appended to a **SQLite database** (via `engine/storage/sqlite.py`) and to structured JSON Lines log files [40]. This provides an audit trail and durability (even if the bot or machine crashes, trade history is not lost).

The engine also has a **reconciliation** process: on startup or on-demand, it can query the exchange for any fills that might have occurred while it was down or missed, and then apply them so the local state catches up [41]. This ensures consistency with the exchange's actual account state.

Position Exit and Order Management: The system includes logic to manage active positions after entry, aiming to autonomously handle **selling/closing positions** under various conditions: - It supports bracket orders such as **One-Cancels-Other (OCO)** orders and **Trailing Stop losses** [42]. For example, if a strategy opens a position, the engine can place a profit-taking limit order and a stop-loss order together (OCO), so that whichever hits first closes the position and cancels the other. Trailing stops adjust the stop-loss level as price moves favorably, locking in profits. - There's an **Event Breakout** module that initially runs in dry-run mode (simulating the trade) and only executes if certain conditions persist, to avoid false breakouts [43]. - A **Stop Validator** runs server-side to ensure stop-loss orders are always in place (if a strategy intended to set an SL but it failed or wasn't acknowledged by the exchange, this module will detect and fix it) [43]. This prevents situations where a position is open without a protective stop. - The strategy logic itself often contains exit rules. For instance, the trend-following strategy monitors indicators (ATR, RSI, etc.) and will signal to exit if trends reverse or an RSI threshold is hit (e.g. `rsi_exit` parameter triggers a sell) [44] [45]. Similarly, the HMM/MA ensemble might close a trade when the model's confidence shifts.

All these ensure that **positions are eventually closed by either reaching a target, hitting a stop, or by strategy logic** – minimizing unattended open positions. Every order funneling through the unified RiskRails + Router pipeline means even exit orders undergo the same checks and logging.

Metrics and Telemetry: At each step, the system emits telemetry: - The engine updates Prometheus **metrics counters/gauges** on events like order submission, fills, rejections, current exposure, etc. For example, it tracks `portfolio_equity_usd` (total account equity), counts of orders filled (`orders_filled_total`), latency of order acknowledgments, and so on [38] [46]. These are exposed on each engine's `/metrics` endpoint. - The Ops service polls the engines (and other services) for data and maintains aggregated metrics. It uses an **engine polling loop** to periodically fetch the latest portfolio snapshot from the engine API and publish a consolidated view (including total equity, open positions, etc.) [47] [48]. This data is cached in `ACCOUNT_CACHE` and also written to a daily snapshot log (JSONL in `data/ops_snapshots`) for historical records [49] [50]. - The Ops API itself has a `/metrics` endpoint combining all subsystem metrics (engines, ops internals) into one Prometheus feed [6]. This is what Prometheus server scrapes. Additionally, Ops provides higher-level endpoints like `/metrics_snapshot` (a quick JSON dump of latest metrics) and specialized ones for PnL or strategy performance.

Finally, these metrics feed into **Grafana dashboards** for human monitoring. The included dashboards (auto-provisioned in `ops/observability/`) visualize things like per-strategy PnL, Sharpe ratio, win rates, drawdowns, latency, etc., and can trigger alerts on anomalies [13] [51]. This gives the operator or developer a real-time view of the bot's behavior, which is crucial for trust in an autonomous system.

Strategy Modules and Adaptability

One of the core goals of the system is **adaptive, autonomous trading** – the ability to adjust to market changes without manual intervention. The repository implements multiple strategy types and a governance mechanism to achieve this:

Built-in Strategies: Under `engine/strategies/`, we find several strategy implementations: - **HMM Ensemble Policy (`policy_hmm.py` & `ensemble_policy.py`):** This is an AI-driven strategy using a Hidden Markov Model to identify market regimes (e.g. trending vs ranging) and an ensemble approach to combine signals. The Moving Average (MA) crossover acts as a baseline strategy and the HMM overlay adjusts trade decisions based on predicted regime (e.g. filtering out signals during choppy markets). The ensemble policy logic fuses these into final buy/sell decisions and can adjust confidence thresholds for execution [52]. The HMM model can be trained offline (with scripts in `scripts/train_hmm_policy.py`) and the system allows hot-swapping models (discussed later in governance). - **Trend-Following (`trend_follow.py`):** A strategy that uses multiple timeframe SMAs and RSI to detect momentum. Its configuration (loaded via `load_trend_config`) supports **auto-tuning** of parameters [19]. For example, it can automatically adjust the ATR-based stop-loss multiplier or re-calibrate risk thresholds after a certain number of trades [53] [54]. The trend module also can use a **symbol scanner** (via `SymbolScanner` and Universe data) to periodically refresh which symbols it trades – keeping a *shortlist* of trending symbols rather than a static set [19]. If `TREND_AUTO_TUNE_ENABLED=true`, it continuously optimizes its strategy parameters (like ATR multipliers, etc.) based on recent performance, which exemplifies the system's adaptive design. - **Scalping and Momentum strategies:** There are likely strategies for short-term mean-reversion or breakout scalping (`scalp` module) and a momentum strategy (`momentum_realtime.py`), given the mentions in docs [55]. These would generate frequent, quick trades on small price movements. They too have their own env-configurable parameters (`SCALP_*`, `MOMENTUM_RT_*`, etc.) and can be turned on/off independently. - **Event-Driven Strategies:** As noted, specialized ones like `listing_sniper.py` (to catch new listings) and possibly `meme_sentiment.py` (not explicitly seen in code, but "MEME_SENTIMENT_*" flags are mentioned [21]). The listing sniper might subscribe to events from the Universe service when a new symbol appears in the "new listing" bucket, then quickly place a market buy to capitalize on initial pump, selling shortly after. The code indicates such event strategies are handled by an `events.external_feed` bus channel [16], meaning external events get turned into strategy triggers internally.

Each strategy module runs *only if enabled* via feature flags, which are documented in `docs/FEATURE_FLAGS.md` (all important flags are also summarized in the generated `.env.example`). This feature-gating allows new strategies to be deployed in "dark mode" (not active) and toggled on gradually – a safety measure to test one component at a time [56].

Adaptability and Learning: Beyond individual strategy logic, the system has a higher-level mechanism to learn and adapt: - **Machine Learning Service (ML):** The `ml_service/app.py` (FastAPI) provides endpoints for training and inference of models, particularly the HMM. It is an optional component (not automatically started in Docker Compose) meant to be run when needed [57]. For example, one could send new market data to `/train` to update the HMM model or call `/infer` to get model predictions. The idea is to allow the trading system to continuously learn from new data. Currently, this is a **manual step** – the ML service isn't fully integrated or automated in the main deployment (marked as a future integration point) [58]. - **Online Learning and Canary Models:** The system supports deploying new strategy models in a "canary" fashion. This means a new model (say a new HMM variant or a different strategy algorithm) can be introduced with a small percentage of traffic to test its performance against the current main model. The **Strategy Router** (discussed in the next section) allows weighted random assignment of signals to different models [59] [60]. A canary model might get, for example, 10-15% of the trades (weight 0.1-0.15) while the primary model gets the rest [61]. The system monitors performance (Sharpe, PnL, win rate) of each model. - **Auto-Promotion of Models:** There is a **governance daemon** that evaluates model performance and can automatically promote a canary to become the new primary model if it statistically outperforms [62]. In the ops service startup, they launch a `canary_evaluation_loop` (in `ops/canary_manager.py`) which continuously evaluates if the canary's results merit promotion [63]. If yes, it can adjust the weights or call the promotion logic. Promotion essentially means switching the "current" model tag in use. This process is logged in a strategy registry (`ops/strategy_registry.json` keeps track of all models, their stats, and promotion history) and can be triggered manually as well via an API endpoint. - **Dynamic Symbol Universe:** The **Universe service and scanner** ensure the set of symbols each strategy trades is not static. For instance, the trend strategy when loading config calls `StrategyUniverse(scanner).get("trend")` which retrieves a tailored list of symbols for trend trading [64] [65]. The Universe service classifies symbols (maybe by market cap, volatility, volume) into buckets, and can "promote" or "demote" symbols between buckets each time `/tick` is called (which likely happens on a schedule) [8] [66]. Thus, if a new symbol becomes hot (high volume or trend), it can get added to the trade list; if an old one becomes uninteresting, it can be dropped. This dynamic rebalancing of what to trade increases the system's adaptability to current market conditions.

In summary, the codebase achieves adaptability through a combination of *self-tuning strategies, continuous performance evaluation, and external intelligence (ML and scanners)*. The trading bot isn't locked to a fixed strategy or set of coins – it's designed to **continuously evolve**. This addresses the goal of autonomy: the system can "learn, evaluate, and deploy" new strategies on its own, as stated in the project's mission [67].

Risk Management and Governance Layer

Running in parallel to the strategies is a robust **governance and risk management layer** in the `ops/` module that oversees high-level decision making, capital allocation, and safety mechanisms:

Ops API and Control Endpoints: The Ops FastAPI (`ops_api.py`) not only aggregates data but also exposes control endpoints to manage the system. For example: - `GET /status` returns a snapshot of the overall system state (trading enabled flag, current balances, PnL, positions, etc., even the health of the ML service) [68] [69]. - `POST /kill` with a boolean will **enable or disable trading** system-wide [70]. This is effectively a remote kill-switch that updates an internal state flag. (The engines check this flag via RiskRails and will reject orders if trading is "killed".) It requires an auth token header for safety [7]. - `POST /retrain` can forward a request to the ML service to train the model on new data [71]. This allows triggering model updates via the Ops API (for example, the UI could offer a "Re-train model" button). - Endpoints under `/strategy/*` provide access to the strategy governance: - `GET /strategy/leaderboard` returns a live performance ranking of all strategy models [72]. - `GET /strategy/status` dumps the entire strategy registry and current leaderboard [73] [74]. - `POST /strategy/promote` allows manually promoting a specific model to be the active one (again requiring auth) [75] [76]. This will update the registry and notify engines of the new "current_model". - There are also endpoints for flushing guardrails counters, listing incidents, etc., which are used in recovery or monitoring scenarios [77] [78].

All these give a centralized **"control plane" API** to manage the bot without directly touching the engine processes. They are the hooks by which an operator or the front-end UI can adjust settings on the fly (more on the UI in the next section).

Strategy Router and Signal Distribution: In `ops/strategy_router.py`, we find the logic for **routing trade signals to different strategy models**. This is crucial in a multi-strategy, multi-model setup. Key points about the strategy router: - It loads a `strategy_weights.json` configuration which defines the weight of each model and which model is "current" (primary) [79] [80]. If the file doesn't exist, it defaults to a single model with weight 1.0 (no competition) [81]. - When a new trade signal comes in (likely from the strategy loop or an external source), `choose_model()` is called to probabilistically pick a model based on the weights [59] [82]. If any secondary model (canary) has a weight above the allowed max canary fraction, it gets capped to ensure the primary still dominates unless deliberately changed [61]. - The chosen model's name (tag) is attached to the signal, and then `route_signal()` will forward the order to one of the engine endpoints for execution [83] [84]. It tries each available engine in `ENGINE_ENDPOINTS` until one succeeds (for redundancy or multi-venue support) [85] [86]. The order is sent as a REST call to the engine's `/orders/market` endpoint with the signal data in JSON. - The router records metrics about how many signals went to each model and how many failed, enabling tracking of model usage [87] [88].

This component essentially implements **traffic shifting between strategies** – a hallmark of modern autonomous trading systems that do A/B testing of strategies. It ensures the system can run multiple strategies in parallel and lean more on the winning ones over time.

Capital Allocator (Dynamic Fund Management): A standout feature for autonomy is the dynamic capital allocation in `ops/capital_allocator.py`. This daemon monitors each strategy model's performance and allocates the capital budget accordingly: - It periodically fetches live metrics (either directly from Prometheus or via the Ops metrics endpoint) to get the current total equity and each model's Sharpe ratio and PnL [89] [90]. It parses metrics like `strategy_sharpe{model="X"}` and `pnl_realized_total{model="X"}` to gather stats per model [90] [91]. - Based on a **capital policy** defined in `ops/capital_policy.json`, it decides how to distribute capital. For example, the policy might say what fraction of equity is allocated to active strategies, minimum capital per strategy, etc. The allocator will calculate a target allocation for each model – for instance, if one model has a much higher Sharpe ratio, it should get a larger share of the total capital pool [92] [93]. - It then writes the decided allocations to `ops/capital_allocations.json` for transparency [94] [94]. These allocations can be used by strategies to scale their trade sizes (e.g., a model with a higher capital allocation can trade larger position sizes). In practice, this could mean adjusting the notional value or risk per trade each strategy is allowed, based on its allocation. - The capital allocator runs continuously (on a schedule, perhaps every N minutes) and includes safety like cooldowns (to not reallocate too rapidly) and persistence so that if the service restarts it doesn't forget past decisions [95] [96]. The design goal is to **"expand or contract per-model budgets automatically"** as mentioned in the changelog [97], essentially **rewarding profitable strategies with more capital and pulling back from underperformers**.

It's worth noting this is *internal fund management* – it doesn't physically transfer funds between accounts, but it governs how the bot's single account equity is conceptually split among strategies. All of this happens in real-time without human input.

Risk Monitors and Playbooks: The ops layer also includes **background risk monitoring** tasks and automated playbooks for certain scenarios: - A risk monitoring loop (`_risk_monitoring_loop` in ops_api) periodically calls an aggregate exposure endpoint and checks if any symbol's exposure exceeds a limit (the `RISK_LIMIT_USD_PER_SYMBOL` we saw, default $1000) [98] [99]. If a breach is found, it can send an alert (e.g., via a webhook to Slack/Telegram) and flag it in logs [100]. This acts as an additional guardrail for unexpected behavior (for example, if some logic bug started opening a too-large position). - **Governance/Guardian Scripts:** Files like `ops/m20_playbook.py` and `ops/m25_governor.py` are referenced as containing scripted mitigation actions [41]. The playbook might include steps to execute if certain alerts occur – for instance, if drawdown exceeds X%, then automatically cut all positions and disable trading (similar to a circuit breaker). The governor might enforce compliance rules or periodically sanity-check that everything is within limits. - On startup, the ops service launches tasks such as `strategy_tracker_loop` and `canary_evaluation_loop` (mentioned earlier) and possibly others like `exposure_collector_loop` and `pnl_collector_loop` [101] [102]. These continuously aggregate data across venues (e.g., combining exposures from multiple engines if trading on multiple exchanges) and track strategy performance over time.

Policy Configuration: Much of the governance behavior is driven by configuration files: - `ops/policies.yaml`**:** Likely defines various risk rules or conditions for the governance daemon. - `ops/strategy_weights.json`**:** Holds current and canary weights as discussed. - `ops/capital_policy.json`**:** Defines how the allocator should behave – e.g., base fraction of equity to allocate, maximum allocation per strategy, etc. - `ops/capital_allocations.json`**:** Output of allocator – shows current allocated amounts. - `ops/strategy_registry.json`**:** Persistent record of each model's stats, which model is current, promotion log (history of promotions with timestamps), etc. The UI and APIs use this to display the leaderboard and such [103] [104].

In summary, the ops/governance layer makes the system **self-regulating**. It not only automates trading decisions but also automates oversight: adjusting capital, selecting the best strategies, and halting or alerting on anomalies. This aligns with the project's principles of *"tight governance and always-on telemetry"* [32]. At this point, many traditional "human trader" decisions (like rebalancing capital or deciding when to switch strategies) have been encoded into the system.

Front-End Web UI Integration (Control Center)

The repository includes a front-end (found under the `frontend/` directory, built with React/TypeScript) that serves as a **web-based control center** for the trading bot. This UI is designed to improve usability by allowing real-time monitoring and manual adjustments to configurations through a friendly interface, rather than editing env files or curling APIs directly.

Integration via Ops API: The front-end is tightly integrated with the Ops service: - The Ops FastAPI mounts a static files directory (`ops/static`) at the root URL, serving the compiled frontend bundle (HTML/JS/CSS) [105]. When the system is running, an operator can open `http://localhost:8002/` and load the dashboard interface [106] (the changelog explicitly notes the "OPS dashboard" URL). - CORS is enabled on the ops server to allow the React dev server (usually running on localhost:3000 during development) to interact with the API safely [107]. This indicates the UI communicates with the backend via HTTP calls (e.g., using REST endpoints or WebSockets provided by `ops_api.py`).

UI Features: Based on the code, the front-end provides several panels and controls: - **Performance Dashboard:** The UI displays key performance metrics of each strategy/venue combination. For instance, a “RightPanel” component shows **PnL, win rate, Sharpe ratio, and max drawdown** for a selected strategy model and exchange venue [108] [109]. It even plots an **equity curve** over time and a “confidence timeline” (which might reflect the model’s confidence or market regime probability) [110] [111]. These visualizations help the user see how each strategy is doing at a glance. Top symbols contributing to PnL are also listed [112]. - **Real-Time Updates:** The frontend likely uses periodic polling or perhaps server-sent events to get updates. The `pnl_dashboard.html` (a static HTML dashboard in ops) uses a JavaScript snippet to fetch `/dash/pnl` periodically and refresh the view [113]. The React app may do similarly via the Ops API endpoints like `/metrics_snapshot` or the specialized `/strategy/leaderboard` and `/account_snapshot`. A “refresh” indicator in the HTML shows when data is updating [114]. This real-time aspect aligns with the system’s real-time requirements – the UI reflects live data streaming from the bot. - **Control Toggles and Actions:** The web UI is intended to allow manual overrides. For example, there might be a toggle switch for “Trading Enabled” which, when clicked, sends a `POST /kill` (with enabled=false or true) to remotely pause or resume trading – effectively the UI kill-switch. The ops API supports this call with an auth token [70]. Similarly, the UI could present a button to **promote a canary model** (invoking `POST /strategy/promote` with the model tag) or to trigger a model retrain (`POST /retrain`). In the Ops API, these endpoints are ready to be called [115] [71]. The frontend likely collects an API token (X-OPS-TOKEN) for authorized actions. While we haven’t seen the exact UI code for these buttons, the backend plumbing and CORS setup strongly suggest the UI is meant as the primary interface for such controls. - **Configuration Management:** The prompt suggests the UI should allow adjusting “any and all configs.” This could mean the UI provides forms or fields to tweak environment variables or policy settings on the fly. For example, adjusting risk thresholds, switching allowed symbols, or tuning strategy parameters. Implementing full config editing might be a work in progress. However, the presence of a React form library (there are components like `Form`, `FormField`, `Checkbox` in `frontend/src/components/ui/`) indicates that forms exist in the UI [116] [117]. These could be used for a settings panel where the user can input new values (e.g. change `TRADE_SYMBOLS` list or risk caps) and submit them. If those connect to an endpoint (which might not be fully implemented yet, as we didn’t see a specific “update config” API), it’s something intended to **improve maintainability** by not requiring direct code edits for config changes.

- **Strategy Governance View:** The repository includes a `ops/strategy_ui.py` which dynamically generates an HTML page for strategy governance visualization [118]. It builds a table of all models with their performance stats (Sharpe, drawdown, realized PnL, trade count, last promotion time) and highlights the current model [103] [104]. This HTML is styled in a dark theme resembling an institutional trading dashboard [119] [120]. The Ops API serves this at `GET /strategy/ui`, so the React frontend may either embed it or replicate its functionality. In any case, the **user can view a leaderboard of strategies/models** and see which one is active and how others compare. This ties into making promotion decisions (manually or automatically).

Collectively, the front-end provides a **“single pane of glass”** for the operator. It aligns with the modular design by using the Ops API as the single integration point – the UI doesn’t talk directly to engines or databases, only to Ops, which in turn talks to everything else. By abstracting configs and controls into a UI, it vastly improves usability: one can monitor the bot’s performance in real-time and tweak settings without stopping containers or editing env files. This is crucial for maintainability, as non-developers (or the developer themselves) can manage the system in production through the UI.

(It's also worth noting that the Ops API requires an auth token for state-changing operations [7] *. This means the UI likely has a way to store this token (perhaps the user enters it or it's configured), ensuring that only authorized users can flip switches or promote models. This is a good practice for security and indicates the system is intended for real deployment use, not just local experiments.)*

Legacy Code Cleanup and Current Status

The repository appears to be in an advanced stage of development (v1.0.0 as of October 2025) where most core features are implemented and earlier placeholder/legacy code has been cleaned up. Notably: - **Deprecated Modules Removed:** The changelog shows that an old `strategies/hmm_policy/` directory (a skeleton from an earlier design) was removed in favor of the unified `engine/strategies/` implementations [121] . Also, an obsolete `ops/run_live.py` launcher was removed, since now `engine/app.py` along with Docker Compose handles launching the live system [121] . This suggests any confusion arising from duplicate or outdated files has been addressed – the codebase now has one clear way to run (via the engine FastAPI and ops service, not via scattered scripts). - **Documentation Updates:** The project maintainers put effort into updating documentation to match the code. The presence of `docs/` files like *SYSTEM_DESIGN.md*, *DEV_GUIDE.md*, *OPS_RUNBOOK.md*, and a step-by-step deployment guide indicates that the docs have been refreshed. Indeed, the changelog notes a "Documentation refresh" with comprehensive design and ops procedures included [122] . Thus, any discrepancies between code and docs should be minimal (the user mentioned the docs *might* be old, but as of this release they are up-to-date). We should rely on the code and the latest docs for truth. - **Dead Code Guard:** There is a GitHub workflow called `dead-code-guard.yml`, implying the project actively checks for unused code or references. This kind of CI step helps prevent legacy bits from lingering. For example, they noted removal of a `scripts/simulate_venue_errors.py` that was flagged as dead code [121] . This demonstrates a conscious effort to keep the repository clean and focused. - **Open Issues and Confusions:** The user hint "assume there are several issues" suggests there may be known issues or areas not fully resolved. Based on the project overview and code: - The **Bybit integration** is one such issue: it's not completed yet [35] . Running the Bybit engine in live mode would reuse Binance logic erroneously. So, until a proper adapter is written, Bybit trading is effectively non-functional (or at best, paper-trading with Binance API). This is on the roadmap to fix. - The **backtesting harness** is another gap. Earlier versions had placeholders that were removed to avoid confusion [123] . The team acknowledges that a production-grade backtester (with historical data replay, slippage modeling, etc.) is not yet included and is being worked on separately (tracked in `ROADMAP.md`) [123] [124] . So users wanting to historically test strategies must wait for those tools or roll their own for now. - **ML Service not in compose:** The ML microservice, while present, is not part of the default docker-compose stack (you must run it manually if needed) [57] . This could cause confusion for someone expecting full out-of-the-box AI integration. It's intentional (perhaps for resource reasons or because not everyone will retrain models frequently), but in terms of autonomy, it means model updates aren't happening automatically yet in the live system. - **Potential Configuration Pitfalls:** There are many configuration knobs. If these are not aligned properly, the system might not behave as expected. For example, if `TRADE_SYMBOLS` is set too restrictive, some strategies might not trade at all. Or if `EXCHANGE_MODE` vs actual account types aren't configured, the engine might error. The documentation likely covers these, and the `.env.example` is generated from code to ensure accuracy [125] . The user should double-check the env settings (especially risk limits) before live trading to avoid surprises. - **Concurrent Execution Issues:** Although not explicitly stated as an issue, running a multi-venue, multi-strategy system is complex. Race conditions or synchronization issues could arise (for instance, the ops service collecting data while engines update it). The project does use async tasks and even file locks to ensure singletons for certain tasks (see the strategy tracker lock to ensure only one process runs it in a multi-worker scenario) [126] [127]. This indicates awareness of concurrency concerns and an attempt to handle them. Nonetheless, one should be vigilant for any edge-case bugs when multiple strategies and components run simultaneously.

Overall, **the repository has largely achieved a stable architecture** – as of v1.0.0 it's labeled "Stable Architecture Release" [128]. All major building blocks for autonomous trading are in place and functioning: - Execution engines (Binance, Kraken, IBKR) are **implemented and stable** [129]. - Strategy runtime with MA+HMM and others is **implemented** [130]. - Risk management is **hardened** (all rails in place) [131]. - Ops control plane, governance, and capital allocation are **autonomous** [131]. - Observability (metrics, dashboards) is **complete** [131]. - Documentation is **comprehensive** [131]. - The system is said to operate *"end-to-end with no human intervention required during normal operation"* [132]. This is a strong statement indicating that, technically, the bot can run by itself: it generates signals, executes trades, monitors itself, and even adjusts itself (to a degree) without needing manual commands.

What remains are mostly **extensions and refinements** rather than fundamental features. We will analyze these next.

Achievements vs Remaining Work (Autonomy Progress)

Given the ultimate goal of a fully autonomous, **profit-generating** trading organism, let's assess how much has been achieved and what is left:

Achieved So Far

The project has made significant progress toward complete autonomy: - **Multi-venue execution & portfolio:** The engine supports multiple exchanges (Binance, Kraken futures, IBKR equities), with a unified interface. It handles all order management, fills, and reconciliation robustly [129]. The in-memory portfolio and persistent storage means it can recover from restarts without manual intervention [39]. - **Integrated Strategy Ensemble:** A built-in **strategy scheduler** runs an MA + HMM ensemble strategy by default, including management of take-profit and stop-loss brackets (so it doesn't require human trade management) [42] [18]. Additional strategies (trend-following, momentum, etc.) can be enabled to diversify the trading approaches – all under the same framework. - **Policy-Based Risk Management:** Comprehensive risk rails ensure the bot won't violate predefined safety rules. This prevents reckless trades and implements a form of self-protection (pausing on too many errors, not oversizing orders, etc.) [27]. - **Ops Governance & Autonomy:** The **strategy router** and **canary deployment** system allow the bot to test new strategies and pick the best performers automatically [62]. The **capital allocator** gives the bot a way to dynamically optimize capital distribution among strategies based on performance metrics (e.g. increasing capital to a strategy with a high Sharpe ratio) [97]. These are advanced features typically requiring human portfolio managers, but here they are algorithmic. - **Observability & Self-Monitoring:** With Prometheus and Grafana integrated, plus alerting rules, the system monitors itself in real-time [12] [13]. There are even automated playbook scripts to react to certain conditions (like an incident response). This means the system not only knows what it's doing, but can also **raise flags or take action** if something goes wrong, increasing reliability for unattended operation. - **End-to-End Workflow Implemented:** The feedback loop is essentially in place – *"Signal → Execution → Metrics → Governance → Capital → (back to) Signal"* is functioning [133]. For example, a model generates a trade signal, trade happens, PnL is measured, the governance logic sees that Model A is performing better than Model B, then governance shifts more weight/capital to A, meaning future signals are more likely from A, thus closing the loop. This continuous learning loop is the core of an autonomous system. - **Documentation & Tooling:** Up-to-date guides and runbooks are available, meaning new developers or operators can understand and run the system more easily [134]. Also, extensive pytest test coverage (80+ tests) exists to ensure stability of components [135]. This lowers the risk of unknown bugs in critical logic and makes the system more robust for 24/7 operation.

In short, the project at this stage already operates as a self-regulating trading bot. It can theoretically run for long periods, adapting to market changes via its HMM and trend modules, and guarding against major risks without human help. This is a **huge milestone** towards the mission of a "fully autonomous trading organism" [67].

Remaining Work and Gaps

However, a few important pieces are still in progress or yet to be implemented before the vision is fully realized:

- **Complete Bybit Integration:** As noted, the Bybit exchange adapter is still a to-do. Implementing Bybit's REST and WebSocket clients (in `engine/core/bybit.py`) and testing them is needed to truly support multi-venue beyond the current ones [136]. Once done, the system can trade on Bybit and potentially other venues, increasing diversification and profit opportunities.
- **Automate ML Service Integration:** Right now, the ML model updates require manual steps (running the `ml_service` and triggering training). The roadmap suggests containerizing this service and wiring it into the Docker Compose, possibly with Role-Based Access Control and automatic reloading of new models [58]. Achieving this means the bot could **continuously retrain/improve its HMM models on new data** in the background, leading to better adaptation to market regime changes without human data-science intervention.
- **Expand Automation (Guardian & Notifications):** While some alerting exists, integrating a notification module (like `ops/m22_notify.py` for Slack/Telegram) will alert humans of important events [137] – e.g., if the bot encounters an exchange outage or hits a drawdown limit and halts, the team should be notified. Likewise, the "guardian" playbook could be expanded to automatically handle more scenarios (for example, auto-reboot a stuck service or dynamically hedge if needed). These make the system more **resilient**, a key part of consistent profitability (since downtime or unhandled incidents can be costly).
- **Robust Backtesting Suite:** Consistent profits can only be achieved if strategies are sound. The lack of a production-grade backtester is a notable gap. The team plans to implement a deterministic backtest harness with historical data replay, slippage modeling, and result reproducibility [123] [138]. This will allow extensive offline testing/tuning of strategies and models. Once in place, it will help validate and refine strategies before (and while) they run live, thereby improving their profit consistency.
- **Further Strategy/Model Development:** The current strategies (HMM+MA, trend, etc.) are a strong starting set. But markets evolve, and strategies might need enhancement:
- The "Meme Sentiment" strategy, for instance, is mentioned but likely experimental. Integrating real sentiment data feeds (Twitter, etc.) could enhance it.
- Additional strategies like arbitrage or mean-reversion could be introduced to diversify the strategy portfolio.
- The HMM model could be upgraded (say HMM v4 or replacing it with a deep learning model) – the framework allows it, and indeed the canary mechanism encourages trying new models. Ensuring the model training pipeline is streamlined will facilitate continuous improvement of the core algorithm.

- **Performance Tuning for Profit:** Autonomy alone doesn't guarantee profits – the strategies must have an edge. There may be remaining work in **calibrating risk-reward parameters** (e.g., ATR multipliers, RSI thresholds in trend strategy) to optimal levels. The auto-tuners help, but it might require more data or smarter algorithms. Also, ensuring the capital allocator's policy is sound (so it doesn't allocate too much to a lucky but risky strategy, for example) is important for long-term profit stability.
- **Scaling and Deployment Hardening:** For a production deployment that consistently earns profit, things like:
- **Secret management:** avoiding plain API keys in `.env` (maybe using a secret vault or Kubernetes secrets) for security [139] .
- **Horizontal scaling:** possibly running multiple engine instances per venue or in multiple regions for redundancy [140] .
- **Latency optimization:** Minimizing latency in data feed and order execution (important for high-frequency parts like scalping).

These are mentioned as future considerations (e.g. multi-region observability, Prometheus federation for scaling) [140] . While not directly affecting "profit", they affect reliability and uptime, which indirectly affect profit (you can't profit if the system is down or compromised).

In summary, **most of the core autonomy features are in place**, and the remaining tasks are about broadening support (exchanges, more automation) and deepening capabilities (backtesting, strategy improvements). The gap between the current state and a fully realized "profit-churning autonomous bot" is mostly about **quality and reliability of strategies** – the infrastructure is largely there.

Roadmap: Steps to Achieve a Consistently Profitable Autonomous Trading System

To reach the end-goal of a self-sustaining, **consistently profitable** trading system, the following step-by-step plan is recommended. This plan addresses the remaining implementation work and also emphasizes continuous improvement of trading performance:

1. Finalize and Test All Exchange Adapters: Complete the implementation of the Bybit client (`engine/core/bybit.py`) and thoroughly test it in both testnet and live environments [136] . After Bybit, evaluate if other exchanges (Coinbase, etc.) are beneficial and consider adding them. Multi-venue support increases opportunities for profit and spreads risk. Ensure each venue's specifics (rate limits, min order sizes, funding rates for futures, etc.) are correctly handled in the adapter.

2. Integrate the ML Service for Continuous Learning: Dockernize the `ml_service` and include it in the `docker-compose` workflow (with appropriate resource limits). Enable the Ops API to automatically call training routines on a schedule or when enough new data has accumulated. For example, the system could retrain the HMM model daily or weekly with recent data and then deploy the new model as a canary. Implement authentication and safe deployment for this (perhaps only promote the new model if it passes certain backtest or paper trade criteria). This will allow the strategy models to **evolve over time on their own**, ideally improving profitability as they learn [58] .

3. Enhance Real-time Monitoring and Notifications: Connect the ops alert webhook to actual notification channels (e.g. Slack, Telegram) using the planned `m22_notify.py` or similar [137]. Set up alerts for conditions like: trading halted, large drawdown, exchange API failures, etc. This ensures that if the autonomous system encounters a situation it can't handle automatically, human overseers are promptly alerted to intervene. It's an important safety net for maintaining account health.

4. Expand the Governance Playbooks: Build out the "guardian" and "governor" scripts to handle more scenarios automatically. For example, if an alert for high drawdown triggers, the playbook could systematically **scale down positions or reduce all strategy allocations** by some factor to reduce risk. If a certain strategy triggers too many risk rule violations, the governance could auto-disable that strategy for a time. Document these behaviors in the OPS_RUNBOOK so it's clear what the bot will do on its own. The goal is to let the system correct or mitigate its own problems whenever possible, further reducing the need for human intervention.

5. Develop a Comprehensive Backtest Harness: Complete the backtesting framework as outlined in the `ROADMAP.md` [138]: - Implement data ingestion for historical market data into the engine (without affecting live state) – e.g., a module to feed stored candlestick data as if they were real-time ticks. - Implement a deterministic clock so strategies experience the backtest with correct timing and sequence, enabling reproducibility of results. - Model the exchange execution environment: simulate order fills with realistic delays, slippage, and fees. This could involve coding a "virtual exchange" that the OrderRouter can send orders to during backtest mode. - Ensure the portfolio updates and metrics work in backtest mode, writing results to a separate output (so as not to mix with live trading data). - Provide a CLI or script interface (like a resurrected `run_backtest.py`) to run backtests for given strategies over specified date ranges. - Create a suite of regression tests using this harness, for example to verify that the HMM strategy yields expected behavior over known historical scenarios.

Backtesting will allow tuning parameters (stop-loss, thresholds, etc.) for **optimal performance**. It will also help in discovering any bugs or edge cases when strategies handle unusual market conditions. By iterating on backtest results, one can significantly improve the consistency of profit in live runs.

6. Refine Strategies and Parameters Continuously: Armed with backtesting and live results: - **Optimize risk-reward settings** for each strategy. For instance, adjust the ATR multipliers in trend strategy to maximize return for a given max drawdown, or tweak the HMM ensemble's confidence thresholds to avoid false signals. Use grid search or optimization algorithms off-line with the backtester. - Incorporate new data features into strategies. The HMM could possibly benefit from more features (volatility regime, volume changes, funding rates, etc.). The momentum/scalp strategies might use order book imbalance from the Screener service for better timing. Adding these can improve edge. - **Expand the universe logic**: ensure the Universe service's criteria for symbol inclusion truly captures the best opportunities (volume, volatility, etc.). Perhaps include an automated check for correlation to ensure diversity of traded symbols. - Test the **event-driven strategies** (listing sniper, etc.) in live or simulated environments to fine-tune their triggers and ensure they are profitable (these can have high upside if done right, but also need careful risk limits). - Keep the strategy codebase clean of legacy approaches – now that `engine/strategies/` is the canonical place, any new strategy should fit into the unified pattern (take advantage of event bus, RiskRails, etc.).

7. Strengthen Testing and Reliability: Increase the automated test coverage further: - Write integration tests that spin up a minimal system (perhaps using a simulated exchange) to test the end-to-end flow: from a strategy signal to a portfolio update, ensuring no part fails. - Test failure scenarios: e.g., simulate an exchange not responding and see that circuit breakers kick in; simulate a high exposure and see that risk monitor alerts and reduces trading. - If possible, run a long-lived paper-trading instance and let it log results to see how stable the system is over weeks, addressing any memory leaks or performance degradations. - Implement robust logging and maybe use the Loki logging stack to retrospectively analyze incidents (the observability stack already has log shipping, which is great for debugging issues after the fact).

8. Deploy Incrementally and Monitor Performance: When rolling out these improvements, do it gradually: - First run the system in **paper trading or testnet mode** (the engine supports a testnet flag for Kraken, and for Binance one can use a test account or trade on a small amount with `TRADING_ENABLED=false` to dry-run) [141] . Observe if the strategies would have been profitable. Use the telemetry to identify any inefficiencies. - Then start with a small amount of capital live, perhaps on one venue (e.g. Binance with a limited set of symbols) to validate real-world performance and behavior. - Ramp up capital allocations as confidence in consistency grows. The dynamic capital allocator will help here by not overshooting any one strategy.

9. Continuously Iterate on Model Governance: As live data comes in, continuously refine the governance parameters: - Check the **promotion criteria** for new models – ensure that the threshold for auto-promotion balances caution and agility (you don't want to switch to a new model on just a fluke lucky streak, but also don't want to lag on adopting improvements). This could involve requiring a minimum number of trades or time for canaries (the code uses `min_trades_for_eval` and `min_live_hours` to that effect) [142] . - Update the **capital_policy.json** if needed to align with risk appetite (e.g., maybe only allocate up to 50% equity to any single strategy until it has a long track record, etc.). - Utilize the **leaderboard** to foster a bit of internal competition: perhaps keep a few canary models running at all times (different parameter sets or even different strategy logics) and let the system pick winners. This can drive innovation in strategy and keep profits stable by always gravitating to what works best in current market conditions.

10. User Interface & UX Improvements: As the backend becomes more capable, ensure the front-end keeps up: - Expose new controls in the UI for any newly tunable parameters or actions (for example, if backtest harness can be triggered via an API, allow the user to start a backtest from the UI with specified dates and show the results). - Provide richer visualizations on the dashboard, such as equity curves per strategy, bar charts of PnL contribution by symbol, or distribution of returns. The more insight the user (or developer) has, the better they can make high-level decisions or notice issues. - Make the UI mobile-friendly if possible (so one can check on the bot from a phone). - Implement user authentication for the UI if deploying in a remote server (to prevent unauthorized access to the control panel).

Following these steps will incrementally move the project from a solid autonomous trading framework to a **polished, battle-tested automated trading system that can operate indefinitely and adapt itself for optimal performance**. The emphasis is on continuous learning (through ML and strategy experimentation), rigorous testing (through backtests and simulation), and robust operation (through risk governance and monitoring).

By completing the remaining integrations and continuously refining the strategies using both live feedback and offline testing, the system stands a strong chance of achieving **consistent profitability** – it will effectively be able to "rack in profits consistently" on its own, while transparently showing its status and allowing oversight through the UI.

---

1 10 12 13 25 30 31 32 35 41 57 58 67 129 130 134 136 137 139 140 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/f0d9c45c122da1530b922f3088e0fd7dd6b5ed26/docs/PROJECT_OVERVIEW.md

2 17 18 20 26 27 37 38 39 40 46 52 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/0dc47fb52d90d6880797c548d6f5cd5a3c68e83e/docs/SYSTEM_DESIGN.md

3 4 5 11 14 15 16 19 21 23 24 33 34 43 55 56 123 125 141 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/f0d9c45c122da1530b922f3088e0fd7dd6b5ed26/README.md

6 7 28 29 36 47 48 49 50 63 68 69 70 71 72 73 74 75 76 77 78 98 99 100 101 102 105 107 115 118 126 127 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/f0d9c45c122da1530b922f3088e0fd7dd6b5ed26/ops/ops_api.py

8 9 66 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/0dc47fb52d90d6880797c548d6f5cd5a3c68e83e/universe/service.py

22 44 45 53 54 64 65 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/0dc47fb52d90d6880797c548d6f5cd5a3c68e83e/engine/strategies/trend_follow.py

42 51 62 94 97 106 121 122 128 131 132 133 135 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/f0d9c45c122da1530b922f3088e0fd7dd6b5ed26/CHANGELOG.md

59 60 61 79 80 81 82 83 84 85 86 87 88 142 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/0dc47fb52d90d6880797c548d6f5cd5a3c68e83e/ops/strategy_router.py

89 90 91 92 93 95 96 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/0dc47fb52d90d6880797c548d6f5cd5a3c68e83e/ops/capital_allocator.py

103 104 119 120 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/f0d9c45c122da1530b922f3088e0fd7dd6b5ed26/ops/strategy_ui.py

108 109 110 111 112 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/f0d9c45c122da1530b922f3088e0fd7dd6b5ed26/frontend/src/components/RightPanel.tsx

113 114 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/0dc47fb52d90d6880797c548d6f5cd5a3c68e83e/ops/static/pnl_dashboard.html

116 117 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/f0d9c45c122da1530b922f3088e0fd7dd6b5ed26/frontend/src/components/ui/form.tsx

124 138 GitHub
https://github.com/mhdrizwan95-netizen/NAUTILUS_BINANCE_STARTER_PACK/blob/0dc47fb52d90d6880797c548d6f5cd5a3c68e83e/ROADMAP.md