ChatGPT

Institutional-Grade Live Signal Optimizer Prompt Pack

This prompt pack provides a comprehensive set of generation instructions to build a **live signal optimization system** that runs in parallel with an existing crypto trading bot. The solution leverages the v3.5 trading bot codebase and its configurable signal logic (e.g. `signal-weights.js`, `adjust-weights.js`, `SignalGenerator-configurable.js`, `WEIGHT_ADJUSTMENT_GUIDE.md`) as a foundation. It will dynamically test and tune strategy parameters **during live trading** without downtime. All prompts are structured for clarity, with usage notes and step-by-step instructions to ensure safe, high-performance, and modular code suitable for production.

Included Components:

- **Live Optimizer Controller:** Orchestrates real-time strategy variant testing alongside the active strategy.
- **Strategy Engine Integration:** Hooks the optimizer into the existing trading engine without disrupting live trades.
- **Optimizer Config Manager:** Manages optimization mode toggles, parameter constraints, and environment settings.
- **Telemetry Dashboard Feed:** Streams real-time performance metrics and status via an in-memory pub/sub for monitoring.
- **Scoring & Confidence Gating:** Computes composite scores (ROI, Sharpe, etc.) for experimental configs and gates promotion of new strategies based on confidence thresholds.
- **Signal Metadata Tagging:** Distinguishes experimental vs. production signals/trades with metadata flags.

Each section below describes the component's purpose, followed by a **prompt** to generate the required code or modifications. **Do NOT include historical backtesting** in any prompt; the optimizer uses real-time data only. All code should target **KuCoin perpetual futures** symbols (e.g. ETH, SOL, XRP) and conform to exchange rate limits and latency requirements. Charting (Chart.js) integration is optional and kept outside core logic (i.e. in the dashboard), so it is not included in these core prompts.

---

1. Live Optimizer Controller

Description: The Live Optimizer Controller runs alongside the main trading loop to continually experiment with different strategy configurations in real-time. It launches **parallel strategy instances** (or simulated sub-strategies) that use alternative indicator weights, signal threshold settings, dual timeframe combinations, and risk parameters. The controller ensures these trials run **without interfering** with the active trading strategy and without causing downtime. It systematically explores **all permutations** (or a configurable subset) of strategy parameters – including indicator on/off combinations, weight distributions, dual-timeframe logic (e.g. combining signals from a fast and a slow timeframe), entry/exit thresholds, and risk management settings – in order to discover better-performing configurations.

Key responsibilities and constraints for the controller include:

- **Parallel Execution:** Run multiple experimental strategies concurrently, each on the same live market data, using isolated state (positions, PnL, signals) for each variant.
- **No Downtime:** The main strategy continues executing normally. The optimizer must catch up gracefully on missed ticks if needed and never block the main event loop.
- **Real-Time Data Only:** Subscribe to the live feed (or reuse the bot's data stream) for price and indicator updates; do not perform any historical data backfills.
- **Performance Tracking:** For each strategy variant, track metrics like ROI, win rate, average P&L per trade, Sharpe ratio, and latency from signal to order execution.
- **Throttling:** Respect exchange API rate limits – reuse shared market data subscriptions and stagger any order placement or query to avoid bursts. **Maximize data ping rate without penalties** (e.g. prefer WebSocket or streaming data to polling).
- **Safety Mechanisms:** All experimental trades use small notional size or paper trading mode to avoid real risk. If using real orders, they must be minimal and immediately protected by stop-loss.

Prompt – Generate Live Optimizer Controller Module:

Use this prompt to have the AI generate a new **Node.js module** (e.g. `optimizerController.js`) that implements the Live Optimizer Controller logic. The code should be modular and integrate with the existing v3.5 architecture (which includes a WebSocket price feed and a signal generator). Include inline comments for clarity.

```
**System**:
You are a senior Node.js backend engineer working on a crypto trading bot
(v3.5). The bot currently trades KuCoin Futures (perpetual swaps) with a signal
generator that scores technical indicators from -100 to +100 [1] . We are adding a
live strategy optimization component that runs in parallel with the main
strategy.

**User**:
Develop a new module called `optimizerController.js`. This module will run
alongside the main trading loop to test multiple strategy configurations in real
time. **No downtime** or interference with current trades is allowed. The
optimizer should:
- Launch and manage multiple **experimental strategy instances** (each with its
own set of parameters and internal state).
- Subscribe to the same live market data feed as the main strategy (reuse
existing WebSocket price and indicator streams).
- For each tick or indicator update, feed the data to all active experimental
strategies to generate signals.
- Allow variations in strategy parameters for each instance, including:
  - Different combinations of technical indicators (enable/disable certain
indicators or use different weightings).
```

```
  - Different weight distributions for indicators (e.g. more weight on momentum vs trend) – use the configurations from `signal-weights.js` as a base [2] , but allow each instance to tweak these values or switch profiles.
  - Different signal threshold levels for buy/sell decisions (e.g. test threshold values around the defaults of 50/70 for buy, -50/-70 for sell [3] ).
  - Different timeframes (each instance can apply dual timeframe logic: e.g. combine a fast timeframe signal with a confirmation from a slower timeframe).
  - Different risk parameters (initial stop-loss %, take-profit %, trailing stop settings, position size factor, leverage).
- **Manage concurrency:** Use asynchronous operations or worker threads if needed so that multiple strategy instances can process ticks in parallel without slowing down the main thread.
- **Rate limiting:** Do not exceed KuCoin API limits when placing orders or querying data. The optimizer should ideally not make its own API calls per tick; instead, it can piggyback on the main feed and only place test orders if absolutely needed. If placing real orders for testing, ensure they are throttled and minimal.
- **Metrics Collection:** Each experimental instance should track performance metrics:
  - Net ROI (% gain/loss relative to a base capital for that instance).
  - Win/Loss percentage and number of trades taken.
  - Average P&L per trade.
  - Sharpe ratio or profit factor for the trades (if feasible to compute live).
  - Signal quality metrics (e.g. how often signals flip, stability of signals).
  - Execution latency (time between signal generation and order execution).
- **Logging/Telemetry:** The controller should emit events or logs containing performance metrics for each strategy variant, to be consumed by a dashboard or logging system (we will set up a pub/sub in another module).
- **Isolation:** Experimental trades should be clearly separated from real trades:
  - If using real orders, use a separate small sub-account or a safety mechanism to immediately close or limit size.
  - Alternatively, simulate orders (paper trade) within the optimizer for metrics, without actually placing on exchange.
- Provide functions within `optimizerController` to:
  - **start()** – initialize and begin running the optimizer (spawn strategy instances, subscribe to feeds).
  - **stop()** – gracefully stop/pause all experimental strategies.
  - **updateConfig(instanceId, newParams)** – (optional) dynamically adjust parameters of a running instance.
- The module should be configurable via a config or environment (for example, number of parallel instances, specific parameter sets to test, etc.).
- Ensure all asynchronous loops or timers are properly handled to avoid unhandled rejections or memory leaks.

Implement `optimizerController.js` with the above requirements. Use clear, self-documenting code and comments for each major section. Do not integrate into the
```

```
main server here (that will be done separately); just provide the module and its
API.
```

Usage Notes: After generation, save this module in the project (e.g. `engine/optimizerController.js`). Review that it properly imports any needed utilities (e.g., the signal generator, config files, or an event emitter for telemetry). Ensure the controller does not automatically start unless invoked (so it doesn't run when optimizer mode is off).

---

2. Strategy Engine Integration

Description: To tie the optimizer into the existing bot, modifications to the main strategy engine (likely in `server.js` or a similar orchestrator file) are needed. This integration ensures that when *optimizer mode* is enabled, the optimizer controller is started and that experimental signals/trades remain isolated from live trades. We will add hooks to initialize the optimizer, feed it data, and possibly execute or simulate trades based on its signals. Crucially, the integration must maintain the **safety and performance** of the main strategy:

- The main signal processing loop should broadcast market data updates to the optimizer controller (so it can update experimental instances). This can be done via an event emitter or by direct function calls to the optimizer with new tick data.
- The trading engine should listen for any **optimizer events** (like when an experimental configuration hits a new high ROI or triggers a potential promotion).
- If an experimental configuration meets the confidence criteria (handled in the scoring component), the integration layer could initiate a strategy switch or notify an operator.
- The integration must respect the **toggle**: if optimizer mode is off, none of these hooks should run. This likely corresponds to a config setting or environment variable.

Prompt – Integrate Optimizer with Main Engine:

Use this prompt to generate the necessary changes in the main server or engine code. This might be a diff or instructions to insert new code. The prompt assumes the main file is `server.js` (or similar) and that the optimizer should start when the system boots (if enabled). It should also cover how to separate experimental from live orders (e.g., by not sending experimental trades to the exchange or using a test flag).

```
**System**:
You are editing the main trading bot application (Node.js, v3.5 `server.js`). It
manages live trading via signals and orders. The code already calculates signals
using `SignalGenerator.generate()` each tick and executes trades for the active
strategy.

**User**:
Modify the main strategy engine to **integrate the new Optimizer** alongside
live trading. Implement the following in `server.js`:
1. **Configuration Toggle**: Check a config value or environment variable (e.g.
`OPTIMIZER_MODE`) at startup. If false, skip optimizer initialization entirely.
If true, proceed with optimizer setup.
2. **Import and Initialize**: Import the `optimizerController` module. During
server startup (after main components like data feed and exchange client are
ready), call `optimizerController.start()` to launch the optimizer in parallel,
but only if optimizer mode is enabled.
3. **Data Feed Hook**: Wherever the code receives new market data ticks or
indicator updates for the main strategy, add a hook to also feed this data to
the optimizer. For example, if there's a function `onMarketData(update)` or a
WebSocket message handler that computes indicators:
    - After computing the indicators for the main strategy, call something like
`optimizerController.onData(update, computedIndicators)`.
    - This will allow the optimizer to update all experimental strategies with
the same data **without extra API calls**.
4. **Order Execution Separation**: Ensure that any trade signals from
experimental strategies **do not directly execute real orders** on the main
exchange account. Possible approaches:
    - The optimizer might already be handling experimental trades internally
(e.g., paper trading). If so, confirm no conflict.
    - If the optimizer were to place real orders (not recommended unless on a
sandbox), use a separate API client or a flag to denote test orders. (In
production, prefer simulation for optimizer.)
5. **Event Listeners**: If the optimizerController emits events (e.g.,
`'strategyPromising'` when a config performs well, or `'metricsUpdate'` for
telemetry), set up listeners in the main engine:
    - e.g., `optimizerController.on('metricsUpdate', (data) => { /* forward to
dashboard or log */ })`
    - e.g., `optimizerController.on('promoteStrategy', (config) => { /* handle
promotion logic, see below */ })`.
6. **Strategy Promotion Logic** (Integration side): When the optimizer signals
that an experimental strategy has surpassed the confidence threshold (we'll
implement scoring later), integrate a mechanism to switch the bot’s active
strategy to the new configuration:
    - Pause or safely wind down current positions if needed (ensure no
conflict).
    - Apply the new strategy parameters to the main strategy (this might mean
updating the active profile or weights, thresholds, etc., e.g., by writing to
`signal-weights.js` active profile or a live config in memory).
    - Log this event and notify the dashboard.
    - This promotion should only happen if a config is clearly superior and
**only** once certain conditions are met (to avoid frequent switching).
7. **Cleanup on Shutdown**: On bot shutdown, if optimizer was running, call
`optimizerController.stop()` to gracefully stop background processes.

Make the code changes, ensuring they are well-documented and do not break normal
operations when optimizer mode is off. Provide the modified code segments (or
pseudo-code with clear markers where changes occur).
```

Usage Notes: This prompt is asking for modifications in an existing file. The AI's response might be a patch or instructions. After obtaining the output, manually merge these changes into the `server.js` (or equivalent) file. Double-check that: - The optimizer only runs when intended. - Data is passed correctly to the optimizer (correct function names and data format as defined in `optimizerController`). - No experimental trade logic interferes with real trading.

---

3. Optimizer Config Manager

Description: The Optimizer Config Manager is responsible for all configuration related to the optimization system. This includes toggling the optimizer on/off, setting global constraints, and defining ranges or sets of parameters to explore. It centralizes parameters such as: - **Enable/Disable** flag (possibly read from an environment variable or config file). - **Indicator and Weight Settings**: which indicators to include in experiments, baseline weight profiles, and ranges for weight adjustments. - **Threshold Ranges**: allowable values or increments for signal threshold parameters (for buys, sells, etc.). - **Timeframes**: list of primary and secondary timeframes for dual-timeframe testing. - **Risk Parameters**: like stop-loss %, take-profit %, trailing stop settings (step size, move size), position size scaling factors (perhaps relative to volatility or signal confidence), leverage mode (auto or fixed). - **Limits**: maximum concurrent experimental strategies, maximum trades per strategy, safe min/max position size, API rate limit safety margins, etc. - **Confidence Criteria**: the performance metrics thresholds for considering a strategy successful (e.g., minimum number of trades, minimum ROI, minimum Sharpe ratio, over a certain observation period).

By adjusting these in one place, the user can control how aggressive or exhaustive the optimizer is. This manager could simply be a JSON or JS module exporting a config object, or a more interactive module that can receive commands (e.g., via CLI or UI to adjust on the fly).

Prompt – Generate Optimizer Config Manager:

Use this prompt to create a configuration module (e.g. `optimizerConfig.js`) or an enhancement to an existing config. It should define all relevant settings and provide utility functions to enforce constraints (for example, ensure weights sum to a certain total, validate that at least one indicator is on, etc.). The code should be well-commented so that future adjustments are easy.

```
**System**:
You are creating a configuration component for the trading bot's optimizer. The
environment uses Node.js and already has config files like `signal-weights.js`
for indicator weights.

**User**:
Create a module `optimizerConfig.js` that centralizes all settings and
constraints for the live optimizer. The module should export an object or
functions for:
- **Enable Toggle**: A boolean flag `enabled` (default read from an environment
variable `OPTIMIZER_MODE` or a config constant).
- **Indicator Settings**: Define which indicators can be toggled in experiments
and any global rules (e.g., always include at least one momentum and one trend
indicator in a config). Optionally, list the indicator keys (matching those in `signal-weights.js` like `rsi`, `williamsR`, etc.) and default weight ranges for randomization.
- **Weight Distribution**: Possibly reuse the profiles from `signal-weights.js` as starting points. For example:
  - `baseProfiles`: copy of the profiles (default, conservative, aggressive, etc.).
  - `weightRange`: an allowable percentage change range for each indicator weight from the base profile (to avoid extreme out-of-bound values).
  - Ensure that for any generated weight set, the total is within a reasonable range (e.g. 100-120 total points [4] ).
- **Signal Threshold Ranges**: Provide default and allowed min/max for signal thresholds (for BUY, STRONG_BUY, SELL, etc.). For example, allow tweaking the buy threshold 50 up/down by 5 points, strongBuy 70 by 5, etc., but not crazy values.
- **Timeframes**: List of timeframes to test (e.g. `['1m','5m','15m','1h']` for short and `['15m','1h','4h','1d']` for long timeframe possibilities). Also a flag if dual timeframe logic is used or not.
- **Risk & Trade Parameters**:
  - `stopLossPercent`: starting SL % (e.g. 0.5% or ROI-based equivalent).
  - `takeProfitPercent`: starting TP %.
  - `trailingStop`: object with `step` and `move` (e.g. step 0.15%, move 0.05% as in current strategy).
  - `positionSizeFactor`: e.g. 0.5% of balance or dynamic based on volatility.
  - `maxLeverage`: default 10 (with auto-leverage enabled by exchange if possible).
  - `useAutoLeverage`: true/false, if false use `maxLeverage`.
  - `maxConcurrentTrades`: e.g. 5 (to match main bot limit).
- **Optimization Limits**:
  - `maxStrategies`: e.g. maximum number of parallel strategy variants to run (to avoid overload).
  - `maxTradesPerStrategy`: a limit of how many trades an experimental strategy can execute in its run.
  - `minRunTime`: minimum time (or number of ticks) an experiment should run before evaluation.
  - API rate limit settings: e.g. `minTickInterval`: if needed, ensure the optimizer processes at most one tick per X milliseconds to not overwhelm.
- **Confidence Criteria**: thresholds for promotion:
  - `minTrades`: e.g. require at least 10 trades.
  - `minROI`: e.g. require ROI > 5%.
  - `minWinRate`: e.g. > 60%.
  - `minSharpe`: e.g. > 1.5 or some figure if Sharpe is computed.
  - A `confidenceScoreFormula` (if using a composite score) or simply individual cut-offs.

Structure the config as a JavaScript object with nested fields as above. Include a function `validateConfig(config)` that checks consistency (e.g. if enabled and maxStrategies > 0, etc.) and possibly adjusts or throws errors for invalid
```

```
values.

Also provide comments and usage examples for how the optimizer and other modules
will use these settings (for instance, how an experimental strategy instance
might receive a random config drawn from these ranges).
```

Usage Notes: The generated config should be saved (e.g., `config/optimizerConfig.js`). After generation, verify that default values make sense and that it imports or references any existing configs (like weight profiles from `signal-weights.js`) correctly. This config will be imported by the optimizer controller and the scoring module to guide their behavior.

---

4. Telemetry Dashboard Data Feed

Description: To monitor the optimizer's performance and status in real-time, the system needs to feed data to a telemetry dashboard or logging system. Rather than directly integrating Chart.js (which is a front-end library) into the backend, we will implement a **pub/sub or event emitter** mechanism in the Node.js backend. This allows the optimizer to publish events (like metrics updates, signals, errors) that a dashboard client (possibly via WebSocket or another process) can subscribe to. The dashboard can then use Chart.js or other visualization tools to display this data.

Key points for the telemetry feed: - Use an in-memory pub/sub (e.g., Node's `EventEmitter` or a lightweight pubsub class) to broadcast events. This avoids heavy external dependencies and keeps latency low. - Events to publish: - **Metrics Update**: e.g., periodically (every N seconds or after every trade) for each strategy instance, an event like `{ type: 'metrics', instanceId, roi, winRate, avgPnL, sharpe, timestamp }`. - **Signal Events**: when an experimental signal triggers a trade entry or exit, maybe log an event `{ type: 'trade', instanceId, action: 'BUY'/'SELL', price, size, pnlIfClosed, etc. }`. These help in debugging and analysis. - **Status/Heartbeat**: e.g., event that optimizer is running and how many strategies active, maybe memory usage or any warnings. - **Promotion Event**: if an experiment gets promoted or if one is terminated early. - The EventEmitter can be a singleton that other modules import, or part of `optimizerController`. The main server (or a WebSocket server) can subscribe and forward to front-end. - Ensure that emitting events is non-blocking and doesn't slow down critical paths (use `setImmediate` or process.nextTick if needed when broadcasting to many listeners).

Prompt – Generate Telemetry Pub/Sub Module:

Use this prompt to create a module (e.g. `telemetry.js` or `optimizerEvents.js`) that sets up the pub/sub system. It should expose an interface for other modules to publish events and for the dashboard to subscribe (likely via the main server forwarding messages). Also, integrate usage of this event bus in the optimizer controller (i.e., optimizer should emit events at appropriate points).

```
**System**:
The project is a Node.js trading bot with an express or WS server for a
dashboard. We want to add real-time telemetry events from the optimizer to the
dashboard.

**User**:
1. Create a module `telemetry.js` that will function as a pub/sub event bus for
optimizer telemetry:
   - Use Node.js `EventEmitter` class (from the `events` module) to create an
event emitter instance.
   - Export functions: `publish(eventName, data)` and `subscribe(eventName,
handler)`. These should internally use a singleton EventEmitter.
   - Consider eventName constants or an enum object (like `OptimizerEvents`) for
standard events (`metrics`, `trade`, `status`, `promotion`, etc.).
   - Ensure thread safety and that subscribers can safely be added/removed.
   - If necessary, implement a simple buffering or rate-limit inside publish for
very frequent events (to avoid flooding).
2. Update the `optimizerController` to use this telemetry module:
   - After each significant action or on a timer, emit a `metrics` event with
performance data.
   - On each experimental trade signal or execution, emit a `trade` event
(include whether it's simulated or real, the strategy ID, and relevant details).
   - When a strategy meets promotion criteria or ends, emit a `promotion` or
`complete` event.
   - Also emit a periodic `status` event (could include number of strategies
running, maybe best current ROI).
3. In the main server (or wherever the WebSocket/HTTP server is), subscribe to
these events and forward to connected dashboard clients:
   - For example, if using WebSocket, on telemetry event, do
`wsServer.send(JSON.stringify({ event: eventName, data }))` to all dashboard
subscribers.
   - If using an Express endpoint for metrics, you might accumulate or serve the
latest metrics on request instead.
   - Document how the front-end can consume these events (but do not write
front-end code).
4. Make sure the telemetry system is optional or at least doesn't crash if no
listeners are attached. If `optimizerConfig.enabled` is false, it's okay if no
events are emitted.
5. Use robust error handling (e.g., try/catch around event processing in case a
listener throws, so one bad dashboard client doesn't break the optimizer).

Provide the code for `telemetry.js` and the changes in `optimizerController`
(and server if needed) to integrate publishing. Include comments explaining each
part.
```

Usage Notes: Once generated, integrate this by saving `telemetry.js` (and importing it in optimizer and server). After launching the bot, verify that: - When the optimizer is running, events are being published (you might log them to console for testing). - The dashboard (if connected) receives data and is able to display or log it. - The event emitter approach does not introduce memory leaks (ensure to remove listeners on client disconnect if applicable).

Remember that Chart.js or actual charting is handled in the front-end; the back-end just provides data streams. The prompts above avoid direct use of Chart.js to keep the core logic clean.

---

5. Optimizer Scoring Algorithm & Confidence Gating

Description: This component evaluates each experimental strategy's performance and decides if/when a new strategy is good enough to **promote** to become the primary trading strategy. It involves computing a **score or ranking** for each configuration based on the metrics collected, and comparing against a confidence threshold. This could be as simple as checking if all individual criteria are met (e.g., ROI > X%, win rate > Y%, etc.), or a more sophisticated formula combining metrics (for example a weighted score or Sharpe ratio). A **confidence score** might be calculated to quantify how likely it is that the observed performance is not due to luck (taking into account number of trades, consistency, etc.).

Key tasks: - After enough data (e.g., min trades or time), compute metrics like: - **ROI** (cumulative returns, possibly fee-adjusted). - **Win/Loss ratio**. - **Average P&L per trade**. - **Volatility of returns** (for Sharpe). - **Sharpe Ratio** (perhaps using a risk-free rate ~0 for simplicity). - **Max drawdown** (to penalize risky strategies). - Possibly a custom **stability score** for signals (did the strategy flip positions too often or behave erratically?). - Combine these into a single score or into a decision rule. - Compare the top configuration's stats to the `optimizerConfig.confidenceCriteria`. Only if beyond thresholds, trigger a promotion event. - Ensure statistical significance: for example, require at least `N` trades or a certain runtime to avoid promoting a lucky short-term winner. - The scoring algorithm should run periodically (e.g., every few minutes or after each new trade closed by any experiment). - If multiple configurations qualify, choose the best one (highest score or ROI) to promote. If none qualify, continue running. - Once a strategy is promoted, optionally you might want to stop or reset the optimizer (or at least mark that config as promoted and maybe continue searching for even better).

Prompt – Generate Scoring & Gating Functionality:

This prompt will create a module or set of functions (e.g. `optimizerScoring.js`) responsible for evaluating strategies. It should integrate with the optimizer controller by consuming the metrics each strategy generates. It might emit events (like `promoteStrategy`) when criteria met. Ensure it references thresholds from the config manager.

```
**System**:
The optimizer is running multiple strategy instances and collecting metrics. We
have a config with confidence thresholds (in `optimizerConfig`).

**User**:
Implement the **optimizer scoring and confidence gating** mechanism:
- Create a module `optimizerScoring.js` that exports:
  - A function `evaluateStrategies(strategies)` that takes a list/array of
strategy performance objects (each containing metrics like ROI, winRate, sharpe,
etc.). It should calculate a **score** for each strategy (if not already
provided).
  - A function `checkPromotion(strategies)` that returns the best strategy
```

```
config to promote if criteria are met, or `null` if none qualify.
  - Possibly a function `computeMetrics(trades)` that given a list of trades
(with P&L, timestamps, etc.) computes ROI, win rate, average P&L, max drawdown,
Sharpe ratio, etc., for that strategy.
- Use the following guidance for scoring:
  - Use **Sharpe Ratio** as a primary metric for ranking (higher is better).
Sharpe can be computed as (mean return / std dev of returns) * sqrt(N) for
normalization. If too few trades or zero std dev, handle gracefully.
  - Also consider ROI and win rate. For example, the score could be a weighted
sum: `score = Sharpe * 50 + ROI(%)*1 + winRate(%)*0.5` (weights are
illustrative).
  - Ensure that a strategy with < `minTrades` (from config) cannot be promoted
regardless of score.
  - If a strategy’s ROI is negative or win rate < 50%, probably its score should
be very low even if Sharpe is weirdly high (include safety checks).
- The `checkPromotion` logic:
  - Identify the current best strategy (highest score or highest ROI if scores
similar).
  - Check it against the **confidence thresholds** from config (e.g., require
ROI >= minROI, winRate >= minWinRate, Sharpe >= minSharpe, etc.).
  - Also ensure it has at least `minTrades` and perhaps has been running for a
minimum duration.
  - If all conditions are satisfied, return that strategy’s config (or an
identifier for promotion). Otherwise return null.
- Integrate this with `optimizerController`:
  - After each experiment concludes a trade or on a set interval (e.g., every
minute), call `evaluateStrategies` to update scores.
  - Then call `checkPromotion` to see if we have a winner.
  - If a promotion candidate exists, emit an event via the telemetry system
(like `telemetry.publish('promotion', {...})`) and also emit an event or call a
callback in the main system to handle the strategy switch (e.g.,
`optimizerController.emit('promoteStrategy', config)` as mentioned before).
  - Potentially, stop other experiments if one is promoted, or mark it so it
doesn’t get promoted again.
- Write the code for `optimizerScoring.js` with these features. Include robust
handling for division-by-zero or cases with too few data points.
- Add comments to explain the formula and thresholds used, and make it clear
where it pulls criteria from `optimizerConfig`.

In summary, produce the scoring module and show an example of how
`optimizerController` would use it (you can simulate a call in code comments or
a small snippet).
```

Usage Notes: Save the resulting module as `optimizerScoring.js`. After integrating, test the logic with dummy data to ensure that: - It correctly identifies when a strategy should be promoted (try feeding one strategy with metrics above thresholds and others below). - It does not promote prematurely with too few trades. - The score makes sense (e.g., consistent winners get higher score). You might need to adjust the weighting formula based on testing, which is fine – the prompt gives one approach but it's adjustable.

Finally, confirm the `optimizerController` uses this module. Likely, you will call `const { checkPromotion } = require('./optimizerScoring');` in the optimizer, and whenever a trade closes or every X ticks, do something like:

```
const bestConfig = checkPromotion(strategyStats);
if (bestConfig) {
    this.emit('promoteStrategy', bestConfig);
}
```

Make sure this aligns with the earlier integration prompt.

---

6. Signal Metadata Tagging

Description: In order to clearly distinguish between normal trading signals and the optimizer's experimental signals, the system will tag each signal (and any resulting orders or trades) with metadata indicating its source. This is crucial for logging, telemetry, and ensuring that experimental activity doesn't confuse reporting or risk management. We will implement a tagging convention such as adding a field like `signal.origin = 'experimental' | 'production'` or a boolean flag in the signal object or trade record. Similarly, any orders could carry a comment or clientOrderId that identifies them as test/optimizer orders.

Implementation considerations: - The `SignalGenerator` or wherever signals are created can be extended to accept a parameter indicating mode. However, since the main and experimental strategies may both use the same SignalGenerator logic, tagging might be done at a higher level (e.g., in optimizerController when it receives a signal, it attaches metadata before processing). - We should ensure that the metadata flows through: - When an experimental signal is generated and perhaps logged or sent to the dashboard, it should be marked. - When an experimental trade is executed (or simulated), mark it so that P&L tracking knows it's from optimizer. - Possibly maintain separate logs or files for optimizer trades vs normal trades. - If using simulation, the trades might not hit the exchange at all, but tagging still helps if we log them or display in the UI. - The main strategy's signals/trades should default to 'production' tags.

Prompt – Implement Signal/Trade Tagging:

This prompt will guide modifying relevant parts of the code to add the metadata tagging. It might involve small changes in multiple places: - The signal generation flow. - The order execution flow (for assigning tags to orders or using special IDs). - The data structures for trades.

We focus on ensuring minimal disruption: ideally just adding extra fields.

```
**System**:
We have a `SignalGenerator.generate(indicators)` that returns a signal score
(and maybe breakdown). The trading engine then decides to BUY/SELL based on
that. We also have the optimizer running parallel signals.

**User**:
Enhance the system to **tag signals and trades** from the optimizer as
experimental:
1. **Signal Tagging**: Modify the signal generation or handling so that each
signal object includes an `origin` (or `source`) property. For example, signals
produced by the main strategy get `origin: 'live'` (or 'production'), and
signals from optimizer instances get `origin: 'optimizer'` or even the specific
instance ID.
   - If the current implementation of SignalGenerator just returns a numeric
score, consider wrapping it in an object like `{ score, breakdown, origin }`.
   - The optimizer controller can also create signals; ensure it attaches origin
metadata when forwarding signals internally or when logging them.
2. **Trade Tagging**: When executing trades, tag the order or trade record:
   - If using a function like `executeTrade(signal)` or similar, update it to
accept an optional `origin`.
   - In the code that logs or records a trade (e.g., adding to a `trades` array
or database), include a field `strategyType: 'production' | 'experimental'` or
similar.
   - If placing real orders via API for experimental trades (not recommended on
main account), consider using a unique `clientOid` with a prefix like `"OPTIM_"`
for identification.
3. **Data Segregation**: Ensure that any P&L or performance calculations in the
main bot exclude experimental trades. For example, if there’s a dashboard
display for total P&L, it should probably not mix optimizer results. Possibly
maintain separate counters.
   - One approach: have separate trade logs: `productionTrades` vs
`experimentalTrades`.
   - Or each trade entry carries the tag and the front-end can filter.
4. **Dashboard Telemetry**: Update the telemetry events (from section 4) to
include the origin. For instance, a metrics update event for an experimental
strategy should clearly indicate it's not the main strategy (maybe by using the
instanceId or a flag).
5. **Minimal Invasiveness**: Make these changes such that if optimizer mode is
off, the origin is basically always 'live' and everything works as before. The
extra fields should be handled gracefully by any code reading them (e.g., front-
end should ignore unknown fields, or you update the front-end if needed to use
them).

Provide the code modifications:
- Possibly the updated `SignalGenerator.generate` function signature or return
structure.
- Changes in trade execution (maybe in a `tradeExecutor` or `orderManager`
function).
- Example of how an experimental signal is created and tagged in the optimizer
flow.

Comment each change to explain its purpose.
```

Usage Notes: After applying these changes, test that: - The main strategy still generates signals and trades normally (with origin 'live'). - In optimizer mode, experimental signals and trades are tagged properly (check logs or telemetry output to confirm). - The front-end or logging doesn't break due to the new fields (you might need to adjust the dashboard to display or filter by origin). This prompt pack focuses on back-end, so front-end changes (if any) would be separate. - These tags will greatly help in analyzing results and ensuring isolation between real and test trades.

Conclusion & Next Steps

By following this structured prompt pack, you can generate all the necessary components for a robust **live optimizer system** that enhances your crypto trading bot. The prompts ensure that the code integrates with the v3.5 architecture, respects real-time constraints, and remains maintainable.

After implementing, carefully test each part in a safe environment: - Run the bot with optimizer disabled to verify nothing has changed in normal operation. - Then enable optimizer with a small number of strategy variants and watch the telemetry to ensure it's exploring and not interfering. - Check that promotion logic only triggers under the right conditions and that switching strategies works smoothly on the fly. - Monitor resource usage (CPU/memory) since running many experiments in parallel can be heavy; adjust `maxStrategies` or tick processing if needed.

This solution will give an **institutional-grade** capability to continuously self-optimize the trading strategy based on live market performance, potentially increasing adaptability and profitability over time. Remember that all new code should be reviewed and tuned for your specific use-case and risk tolerance. Good luck with your implementation!

---

1 4 WEIGHT_ADJUSTMENT_GUIDE.md
file://file_000000007c5c71fdaf7e5effb2ff8603

2 3 signal-weights.js
file://file_00000000b17c71fdb8754ff1808f09bd