ChatGPT

Dual-Timeframe Screener Module Prompt Suite

Overview and Objectives

This prompt suite is designed to guide the creation of a **dual-timeframe screener** module for the KuCoin futures trading bot (v3.5). The screener scans multiple symbols across two timeframes and identifies when technical indicators align on both frames, generating trade candidate signals. Key goals include low-latency operation via WebSocket data, efficient incremental indicator calculations, and easy integration with the existing bot architecture.

Key Features to Implement: - **Multi-symbol, multi-timeframe scanning:** Monitor 20+ KuCoin Futures symbols on two timeframes each (configurable, e.g. 5m and 15m). - **Indicator alignment detection:** Evaluate 2–3 technical indicators per symbol (default: RSI, MACD, Williams %R, AO) and detect when they show consistent bullish or bearish signals on both timeframes. - **Real-time data feed:** Use KuCoin's WebSocket API for live OHLCV candle updates [1] [2] (with REST as optional fallback) and maintain a rolling buffer of recent candles for each timeframe. - **Incremental indicator calculations:** Implement indicator classes with a *nextValue* pattern so each new candle updates indicators without full recomputation (for performance). - **Output of signals:** When alignment criteria are met, output the candidate (symbol and brief reason) via multiple channels – append to a JSON lines log file, emit an in-memory event, and optionally print to console.

The following sections break down prompt instructions for each module/component needed, followed by testing scenarios. Each prompt should be given to a code generation assistant to produce the respective module code.

Screener Engine (`screenerEngine.js`)

Description: The screener engine orchestrates the entire screening process. It runs as an independent Node.js process or service, parallel to the main trading bot, and coordinates data intake, indicator updates, alignment checks, and signal output.

Prompt for `screenerEngine.js`:

- Initialize the screener as a standalone module that can start and stop via CLI or script. On start, load the configuration from `screenerConfig.js` (including the list of symbols, chosen timeframes, indicator parameters, and alignment criteria).
- Set up the KuCoin WebSocket **data feed**. Connect to the public futures WS endpoint and subscribe to candlestick channels for each symbol and timeframe in the config (e.g. topics like `/market/candles:BTC-USDT_5min`) [1]. Use a single connection for multiple subscriptions (KuCoin supports ~100 topics per connection) [3].

- Ensure the WebSocket connection stays alive with heartbeats: send ping frames at the interval recommended by KuCoin (every ~18 seconds) [4] and handle **pong** responses. Implement auto-reconnect on connection drops or errors.
- For each symbol and timeframe, maintain a **candle buffer** in memory (e.g. an array or queue). When a new candle message arrives:
- Parse the data (timestamp, open, close, high, low, volume) and convert numeric strings to numbers [1].
- Append it to the corresponding buffer, and if the buffer exceeds a configured length (e.g. 1000 candles), remove the oldest entry to cap memory usage.
- Update the **indicator engines** for that symbol/timeframe with the new candle's values (see indicator classes below for `next()` methods).
- After updating indicators for a given timeframe, check if the paired timeframe for the same symbol also has up-to-date indicator values. Use the `timeframeAligner` module to evaluate **alignment conditions** between the two timeframe readings:
- Determine if the configured indicators are **bullish**, **bearish**, or neutral on each timeframe (e.g. RSI <30 on both frames would be bullish oversold alignment [5], RSI >70 on both would be bearish overbought alignment).
- Identify if the required number of indicators are aligned in the same direction. For example, if config requires at least 2 of 3 indicators bullish on both frames, confirm this condition.
- The aligner should also consider trend/momentum conditions: e.g. MACD histogram >0 on both frames (bullish momentum) [6], Williams %R < -80 on both (bullish oversold) [7], AO >0 on both (bullish momentum) [8], etc.
- If an alignment signal is found (either **bullish** or **bearish**):
- Construct a signal object with details (symbol, timeframes, alignedIndicators list, timestamp, and signal direction).
- Pass the signal to the `signalEmitter` for output distribution (logging to file, emitting events, etc.).
- *(Optional:* You may also maintain an internal de-duplication or rate-limit if needed to avoid spamming signals if the same alignment persists across multiple candles).
- Provide a clean **shutdown** mechanism: unsubscribe from all WS channels and close the connection, flush any buffered data if necessary, and stop any timers (heartbeats or polling).
- Keep performance in mind: use asynchronous/non-blocking operations for network and file I/O. Rely on the incremental calculations to minimize CPU work per candle. The screener engine should handle ~40 channels (e.g. 20 symbols * 2 timeframes) in real-time without lag.

Timeframe Aligner (`timeframeAligner.js`)

Description: This module encapsulates the logic for checking indicator alignment between two timeframes for a given symbol. It takes the latest indicator values from the two timeframe contexts and determines if the alignment criteria for a bullish or bearish signal are met.

Prompt for `timeframeAligner.js`:

- Export a function or class (e.g. `checkAlignment`) that accepts two sets of indicator readings (one for the lower timeframe, one for the higher timeframe) and the alignment rules from config.
- The function should evaluate each configured indicator's state on both timeframes:

- Define what constitutes a bullish vs bearish signal for each indicator:
  - **RSI:** bullish if value is below oversold threshold (e.g. <30) or rising upward from a low; bearish if above overbought threshold (e.g. >70) [5].
  - **MACD:** bullish if MACD line > signal line (or histogram > 0) on both frames; bearish if MACD line < signal line (histogram < 0) [6].
  - **Williams %R:** bullish if %R is below -80 (oversold) on both frames; bearish if above -20 (overbought) [7].
  - **AO (Awesome Osc):** bullish if AO > 0 (momentum positive) on both; bearish if AO < 0 on both [8].
  - *(Extend logic for any additional indicators in config, e.g. EMA trend crosses, Stochastic, etc.)*
- Determine alignment for each indicator: it is "aligned bullish" if it indicates bullish on both timeframes, "aligned bearish" if bearish on both, otherwise not aligned (if one timeframe is bullish and the other neutral/bearish, that indicator doesn't count as aligned).
- Based on the config criteria, decide if a signal should be emitted:
- For example, if **at least 2 out of 3** monitored indicators are aligned bullish, return a result indicating a bullish alignment. The result could include which indicators met the condition and the overall signal direction.
- Similarly, check for bearish alignment (e.g. 2 out of 3 aligned bearish).
- If the conditions are not fulfilled, return a null/false or an object indicating "no signal".
- The alignment checker should be flexible: allow config to specify a strict requirement (e.g. *all* listed indicators must align) or a minimum count ("threshold of indicators") to trigger. Use the config's values for oversold/overbought thresholds where applicable (these might be included in the indicator parameters).
- **Output structure:** Design the output of `checkAlignment` to clearly convey:
- **direction**: `"bullish"`, `"bearish"`, or none.
- **indicators**: which indicators aligned (e.g. `["RSI","MACD"]`).
- Optionally, the measured values or any additional context (for logging/analysis purposes).
- Keep the function pure (no side effects). It should just evaluate conditions. The screener engine will use this to decide on emitting signals.

Indicator Engines (`indicatorEngines/`)

Description: Under this directory, implement specialized classes for each technical indicator (RSI, MACD, Williams %R, AO) with the ability to update values incrementally. Each class will be responsible for computing the current indicator value given new price data, while maintaining the state needed (such as previous averages or window of values).

General requirements for all indicator classes: - Each class should have a constructor that accepts configuration (like period lengths or other parameters). - Each class should have an `update(candle)` (or `next`) method that consumes a new candle (with at least close price, and high/low if needed) and updates the internal state, returning the latest indicator value. - Implement **incremental calculation** so that new values are computed efficiently: - Use previously stored intermediate values (like running averages, previous EMA, etc.) to avoid full recomputation on each update. - Only if not enough data has been accumulated (e.g., during startup), handle that (possibly by storing initial values until the first calculation is possible).

Below are prompts for each specific indicator class.

RSI Indicator Class Prompt (`RSIIndicator.js`)

- **Purpose:** Compute the Relative Strength Index for a given period (default 14).
- **Internal state:** Track the **average gain** and **average loss** over the period, updating these on each new close price:
- When a new close price comes in, calculate the change = current close - previous close.
- Compute the gain = max(change, 0) and loss = max(-change, 0).
- Update average gain and loss using Wilder's smoothing technique: \ `avgGain_new = ((avgGain_prev * (period-1)) + currentGain) / period` \ `avgLoss_new = ((avgLoss_prev * (period-1)) + currentLoss) / period` \ (This formula gives an exponential-like moving average of gains/losses [9] .)
- **Note:** For the very first calculation, you need an initial average gain/loss (e.g., the simple average over the first `period` candles). Until at least `period` data points are collected, you may not produce an RSI value.
- After updating avgGain and avgLoss, calculate **Relative Strength (RS)** = avgGain_new / avgLoss_new. Then compute RSI = 100 - (100 / (1 + RS)) [10] [9] . Handle edge cases: if avgLoss is 0 (no losses), RSI = 100; if avgGain is 0, RSI = 0 [11] .
- The class should store the last close price to compute the next change. Also store the current RSI value (or simply compute on the fly after updating averages).
- **Output:** The `update()` method returns the latest RSI value (a number between 0–100). This value can be used to assess conditions like oversold/overbought: e.g., RSI < 30 (bullish oversold signal) or > 70 (bearish overbought) by convention [5] .
- Optimize for incremental updates: once initialized, each `update` does constant work. Avoid recalculating over the whole window of closes. Use the stored averages as described.
- Provide a way to **initialize** the averages if a batch of historical data is supplied (for instance, if the screener loads the last 14 closes on startup). If no history is given, you might accumulate gains/losses until you have `period` samples then start producing RSI.
- Example behavior: If period = 14, and on a new candle the price jump yields a large gain, avgGain will gradually reflect this over time (smoothing). The RSIIndicator should accurately reflect quick momentum shifts but also smooth them as per formula.

MACD Indicator Class Prompt (`MACDIndicator.js`)

- **Purpose:** Calculate the Moving Average Convergence Divergence indicator, which consists of the MACD line (difference between two EMAs) and a signal line (EMA of the MACD line), plus a histogram.
- **Parameters:** Typically uses `fastPeriod = 12`, `slowPeriod = 26`, and `signalPeriod = 9` (these are defaults, but allow config override).
- **Internal state:** Maintain the following:
- The latest **EMA_fast** (e.g. 12-period EMA of closing price).
- The latest **EMA_slow** (26-period EMA of closing price).
- The latest **MACD line** value = EMA_fast - EMA_slow.
- The latest **Signal line** (9-period EMA of the MACD line).
- The latest **Histogram** = MACD line - Signal line.
- **Initialization:** The first EMA values require seed data:
- One common approach: calculate simple moving averages for the first `fastPeriod` and `slowPeriod` data points to start EMAs. Alternatively, if a history is provided, use the official method (e.g., first EMA_fast = SMA of first 12 closes).

- If no history, you might wait until enough data comes in (but that delays output). A practical approach is to fetch initial candles from REST (e.g., last 30 candles) to bootstrap – this can be done outside the class and then passed in.
- **Update formula:** On each new close price:
- Update EMA_fast and EMA_slow using the classic EMA formula: \ `EMA_new = (NewPrice * α) + (EMA_prev * (1 - α))` \ where α = 2/(period+1) [12]. Compute α_fast for period 12, and α_slow for 26.
- Calculate new MACD_line = EMA_fast_new - EMA_slow_new [13] [14].
- Update the Signal line EMA similarly using MACD_line values with α_signal = 2/(signalPeriod+1).
- Compute Histogram = MACD_line - Signal_line.
- Return an object or tuple of the MACD-related values (MACD line, Signal line, Histogram). For alignment checks, typically:
- **Bullish** signal if MACD line is above Signal (histogram > 0) on both timeframes [6].
- **Bearish** if MACD line below Signal (histogram < 0).
- Ensure the class handles incremental updates efficiently (each update O(1)). Only store needed state (prev EMAs, etc.). The EMA formula inherently is incremental, no need to loop over all past data each time [12].
- Edge cases: Before signal line is established (requires ~signalPeriod MACD values), you might delay histogram calculation. Also, if using initial SMA as EMA start, the first few values will have less accuracy – acceptable for a screener or mention in documentation.

Williams %R Indicator Class Prompt (`WilliamsRIndicator.js`)

- **Purpose:** Compute Williams %R, an oscillator showing the current close relative to the high-low range over a look-back period (default 14 periods). %R outputs values from 0 to -100 (with 0 at high of range, -100 at low).
- **Internal state:** Likely maintain a rolling window of the recent **highs**, **lows**, and the last `period` closes (or at least the last close, though for calculation you need the min and max).
- **Calculation:** For each update:
- Add the new candle's high and low to a window (e.g., queue or array). Also track the new close.
- Remove data falling outside the look-back window (older than `period` candles ago) from the window.
- Determine the **highest high** and **lowest low** in the window (for the last N periods, including the current one).
- Compute %R = (highestHigh - latestClose) / (highestHigh - lowestLow) * -100 [15]. This yields a value between 0 and -100.
- A straightforward approach is to store the last N highs and lows and recalc min/max each time. Given N is typically small (14), this is efficient enough. For performance, one can use a deque structure to get max/min in O(1) time, but with N=14, O(N) per update is fine.
- **Output:** The `update()` returns the latest %R value (e.g. -85.0 means price is near the bottom of the range).
- Use this to detect oversold/overbought:
- %R < -80 is often considered **oversold** (bullish setup) [7].
- %R > -20 is **overbought** (bearish setup).
- Ensure window management is correct: when length > period, pop the oldest. Use the candle's data for high/low; some implementations use just close for %R, but standard is highest high and lowest low of the period.

- Example: If the highest high of last 14 is 105 and lowest low is 95, and current close is 96, then %R = (105 - 96)/(105 - 95)*-100 = 9/10 * -100 = -90 (quite oversold).
- The class should be able to handle the case where not enough data yet (e.g., less than N candles: you might wait or compute using whatever is available, noting that signals in that phase may be less reliable).

Awesome Oscillator Indicator Class Prompt (`AwesomeOscillator.js`)

- **Purpose:** Calculate the Awesome Oscillator (AO), which is the difference between a fast and slow simple moving average of the median price (usually 5-period SMA minus 34-period SMA of (High+Low)/2) [16] [17].
- **Parameters:** Standard periods are 5 and 34.
- **Internal state:** Maintain rolling calculations for:
- The 5-period **SMA of median price**.
- The 34-period **SMA of median price**.
- The current AO value = SMA_fast5 - SMA_slow34.
- Each new candle:
- Compute the candle's median price = (high + low) / 2.
- Update the 5-period SMA with this new median:
    - Keep a queue of last 5 median prices. Add the new one, remove the oldest if more than 5 in queue.
    - Compute SMA_fast = sum(queue5) / 5.
- Update the 34-period SMA similarly with a queue of length 34 for median prices.
    - Compute SMA_slow = sum(queue34) / 34.
- Calculate AO = SMA_fast - SMA_slow [18].
- The first outputs will occur only after enough data (at least 5 for an initial fast SMA, and 34 for a full slow SMA). Before that, you might either wait or compute partial (though better to wait for full period for accuracy).
- Return the AO value on each update. Positive AO indicates momentum is bullish (short-term average above long-term) [8], negative indicates bearish momentum.
- Because SMA is a simple average, you can maintain the sum for each window:
- e.g., keep `sum5` of last 5 medians. On new update, do `sum5 += newMedian - oldestMedian` (if queue full) to get new sum in O(1), then SMA_fast = sum5/5. Same for sum34.
- This avoids recomputing sums from scratch each time.
- The class should handle window initialization: until 34 values collected, the 34 SMA is not fully valid. You might still compute AO with fewer values (it will be volatile) or choose to output only after 34th data point. Document this choice.
- Ensure to output AO as a single number. For alignment, we consider AO > 0 bullish, AO < 0 bearish [8].

Screener Configuration (`screenerConfig.js`)

Description: A configuration module that defines how the screener operates. This includes which symbols and timeframes to scan, which indicators to use and their parameters, and what constitutes an alignment signal.

Prompt for `screenerConfig.js`:

- Export an object (or `module.exports`) containing configuration fields such as:
- **symbols:** an array of symbol names (strings) to monitor (e.g. `['BTCUSDTM', 'ETHUSDTM', ...]` for KuCoin futures symbols).
- **timeframes:** an array or tuple of the two timeframes to align (e.g. `['5m', '15m']` or for flexibility, an array like `[['5m','15m'], ['15m','1h']]` if multiple pairs are allowed). In simplest form, use two keys: `primaryTimeframe` and `secondaryTimeframe`.
- **indicators:** list of indicators to evaluate alignment on. Could be an array like `['rsi', 'macd', 'williamsR', 'ao']` (the default set). Only these will be calculated and checked. (The screener can still calculate others if needed by main strategy, but alignment logic will focus on this list.)
- **indicatorParams:** optional sub-config for each indicator:
  - e.g. `{ rsi: { period: 14 }, macd: { fastPeriod:12, slowPeriod:26, signalPeriod:9 }, williamsR: { period:14 }, ao: { fast:5, slow:34 } }`. Include threshold levels if needed (though those could be fixed in code or derived from common defaults).
- **alignmentCriteria:** define the rules for triggering a signal:
  - This could be a number or structure, e.g. `minAligned: 2` (meaning at least 2 indicators of the set must align bullish/bearish).
  - Or a boolean setting for strict mode: `requireAll: false` vs true.
  - Possibly separate criteria for strong vs weak signals (not necessary, but could allow: e.g., if all 3 align, mark as "strong" signal).
- **outputs:** configuration for the signalEmitter outputs:
  - E.g. `{ logToFile: true, logFilePath: 'screener_matches.jsonl', console: false, emitEvents: true }`.
- **dataFeed:** config for data feed behavior:
  - e.g. `{ useWebSocket: true, useRestFallback: false, restPollInterval: 30000 }`. Also API credentials if needed (for private channels, though here only public market data is needed).
  - Perhaps a **volume/volatility filter**: if dynamic symbol filtering is desired, include thresholds like `minVolume` or `minVolatility` and the screenerEngine could skip symbols not meeting those. (This is optional advanced feature; mention if needed.)
- Provide sensible defaults. For example:
- timeframes: 5m and 15m (common intraday alignment).
- indicators: RSI, MACD, Williams %R, AO by default.
- alignmentCriteria: require at least 2 of those 4 to align (so we don't demand all four, reducing noise).
- log file path default to `screener_matches.jsonl` in the project directory.
- Ensure the config is clearly documented so a user can easily tweak symbol lists or change timeframes without code changes. For instance, add comments explaining that KuCoin futures symbols typically end with **"USDTM"** for perpetuals.
- If needed, include mapping or conversion for timeframe strings to the format required by KuCoin API (the config might use `5m` but WS topic expects `5min` or similar; KuCoin uses `1hour, 1day` etc. In the example from docs it was `1hour`, `15min` [19] – verify and ensure our code handles the format).
- Example config snippet:

```
module.exports = {
  symbols: ['BTC-USDT', 'ETH-USDT', 'XRP-USDT'],
  primaryTimeframe: '5min',
  secondaryTimeframe: '15min',
  indicators: ['rsi','macd','williamsR','ao'],
  indicatorParams: {
    rsi: { period: 14, oversold: 30, overbought: 70 },
    macd: { fast: 12, slow: 26, signal: 9 },
    williamsR: { period: 14 },
    ao: { fast: 5, slow: 34 }
  },
  alignmentCriteria: { minAligned: 2 },
  outputs: {
    logToFile: true,
    logFilePath: 'screener_matches.jsonl',
    console: true,
    emitEvents: true
  }
};
```

(Note: Use the exact symbol format KuCoin expects for subscription, e.g. `'BTC-USDT'` *vs* `'BTCUSDTM'` *depending on the API. KuCoin futures may use a suffix ".M" or similar; ensure it matches the actual exchange symbol.)*

Signal Emitter (`signalEmitter.js`)

Description: Handles outputting the screener's findings to various targets – file, console, and event bus. This module abstracts away the output logic so the screener engine just calls something like `signalEmitter.emit(signal)` and doesn't worry about the details.

Prompt for `signalEmitter.js`:

- Implement as a class (e.g. `SignalEmitter`) or a set of functions that can be initialized with the config outputs settings.
- **File output:** If `logToFile` is enabled in config:
- Open (or create) a file stream to the given `logFilePath`. Ideally append in text mode. Each signal should be written as a separate line in JSON format (JSON Lines):
  - e.g. `{"ts": 1672531199000, "symbol":"ETHUSDTM", "timeframes":["5m","15m"], "direction":"bullish", "indicatorsAligned":["RSI","MACD"]}` and then a newline.
- Using newline-delimited JSON allows easy parsing later. Make sure to flush or handle the stream properly (use `fs.appendFile` for simplicity or maintain a write stream).
- Consider file rotation or size management if this will run long (optional).
- **Console output:** If enabled, simply `console.log()` a human-readable message for each signal:

- e.g. `console.log("Screener match: ETHUSDTM RSI and MACD bullish on 5m & 15m")`. Craft the message to include symbol, the indicators aligned, and the timeframes or nature of alignment.
- Avoid console logging if not enabled to reduce noise (this should be optional for debugging).
- **Event emission:** If `emitEvents` is true, the module should extend or use an EventEmitter:
- For example, `class SignalEmitter extends EventEmitter { ... }` so that other parts of the bot can listen for `'signal'` or `'match'` events.
- After outputting to file/console, also do `this.emit('signal', signalData)` so that the main strategy engine (if running in same process) could capture it. Document this event name.
- If the screener runs as a separate process, an alternative could be to use IPC or a message queue, but that's beyond scope – here we assume either same process or file-based integration.
- **Initialization:** Provide a function or constructor that takes the config. For file writing, open the file in append mode at start (and perhaps write an initial header or timestamped start marker if desired).
- Ensure thread-safety if multiple signals come in quick succession: using Node's single thread nature and appendFile's internal queue is usually sufficient. If needed, queue writes to avoid concurrency issues (probably not an issue here).
- Close file descriptors on program exit (maybe listen for a shutdown event from screenerEngine to close the log file cleanly).

- Example usage in screenerEngine:

```
const emitter = new SignalEmitter(config.outputs);
...
if (alignmentResult) {
    emitter.handleSignal({
      symbol,
      timeframes: [tf1, tf2],
      direction: alignmentResult.direction,
      indicators: alignmentResult.indicators
    });
}
```

  Inside `handleSignal`, it would perform the three output actions as configured.

- **JSON logging note:** Use consistent keys and formats in the JSON. Include a timestamp (`ts` in UNIX ms or an ISO string) for record. This will help in later analysis. For human readability, the console message can be shorter.

Data Feed (`dataFeed.js`)

Description: Encapsulates the market data sourcing. In this case, primarily a WebSocket client for KuCoin candlestick data. It manages subscriptions and incoming data, and feeds candles to the screener engine. Optionally supports a REST polling fallback.

Prompt for `dataFeed.js`:

- Implement a class `DataFeed` that handles connecting to KuCoin's API:
- Use KuCoin's WebSocket public endpoint for market data. No authentication is required for public market feeds (candlesticks).
- The connection process on KuCoin involves obtaining a WebSocket **endpoint URL and token** via a REST call (`/api/v1/bullet-public` for Classic) [20] . *However*, to simplify, you may use the hard-coded endpoint if allowed (KuCoin docs provide wss URLs like `wss://ws-api.kucoin.com/endpoint` with the token).
    - For production-ready code, actually perform the REST call to get a fresh token and endpoint. The returned data will have an endpoint URL, a token, and a ping interval. Connect using that.
    - Since this is a prompt, you can outline that step or use a direct connection if that detail is too deep. (If using an NPM library like `@iofate/kucoin-ws` , it handles this under the hood.)
- Provide methods:
- `connect()` : Establish the WebSocket connection. Handle the handshake (including sending ping as needed).
- `subscribeCandles(symbol, timeframe)` : Subscribe to a candle feed topic:
    - Format the topic string as `"/market/candles:{symbol}_{type}"` where symbol is e.g. `"BTC-USDT"` and type is `"5min"` [21] [1] .
    - Send a subscription message as per KuCoin protocol:

      ```
      {
        "id": Date.now(),
        "type": "subscribe",
        "topic": "/market/candles:BTC-USDT_5min",
        "response": true
      }
      ```

      (The `id` can be any unique number, `response:true` asks for ack).
    - Repeat for each requested symbol/timeframe.
    - Ensure not to exceed subscription limits. For KuCoin futures, **no specific topic limit per connection is documented** (spot has 400) [22] , but if you had to subscribe hundreds, consider multiple connections. For ~40 topics, one connection is fine.
- Internally, set up the WebSocket `'message'` event handler to process incoming messages. KuCoin sends a JSON message with `topic` , `subject` , and `data` for each update [23] .
    - Filter for messages where `subject === "trade.candles.update"` (which are candlestick updates) [24] .
    - Parse the `data.candles` array. The array structure is: `[ startTime, open, close, high, low, volume, turnover ]` as strings [1] . Convert these to a structured candle object:

      ```
      {
        startTime: Number(data.candles[0]) * 1000, // if it's in seconds
      epoch
        open: Number(data.candles[1]),
        close: Number(data.candles[2]),
      ```

```
    high: Number(data.candles[3]),
    low: Number(data.candles[4]),
    volume: Number(data.candles[5])
    // turnover value is data.candles[6] if needed
  }
```

Note: KuCoin provides startTime in seconds (as string) [25] and a separate `data.time` as current timestamp in microseconds which can be ignored for our purposes.

  - Determine **candle completion**: Typically, KuCoin sends an update only when a candle closes (since push frequency is “real-time” on close) for that interval. We should confirm if they send interim updates. If they do send interim, use the `data.symbol` and perhaps the fact that a new candle with the same timestamp may update. To keep it simple, assume each message is a completed candle for the previous interval.

- Emit or callback the new candle to the screener engine:
  - DataFeed could be an EventEmitter that emits `'newCandle'` events with payload `{symbol, timeframe, candle}`. The screenerEngine can subscribe to these events.
  - Alternatively, provide a callback in the constructor (e.g. pass a reference to screenerEngine’s handler).
  - Using events is cleaner: e.g. `this.emit('newCandle', symbol, timeframe, candle)`.
- Implement ping/heartbeat:
  - Use the `pingInterval` from KuCoin’s handshake response (often ~15-20s). Set `setInterval` to send a JSON `{"id": pingId, "type":"ping"}` frame every interval [4].
  - On receiving a `pong` (KuCoin might respond with type `"pong"`), you can log or simply ignore beyond resetting connection timer.
- Reconnect logic:
  - Listen for `'close'` or `'error'` events on the WebSocket. If the connection drops, attempt to reconnect after a short delay.
  - On reconnect, you will need to fetch a new token (if using the official flow) and resubscribe to all topics. Keep track of what was subscribed.
  - Ensure to clear old timers and event handlers on reconnect to avoid duplicates.
- **REST fallback (optional):** Implement only if needed:
- If `useRestFallback` is true or if WS fails, allow fetching candles via REST API (`GET /api/v1/kline/query?symbol=&granularity=`) [26]. You could schedule an interval (like every X seconds) to poll the latest candle for each symbol/timeframe. Rate limit: KuCoin allows 3 requests/sec for this endpoint [27], so polling 40 pairs would need staggering or slower polling.
- Given WebSocket is preferred (real-time, no rate limit issues [2]), the fallback can be minimal or just a note in code that it exists.
- **Testing hooks:** Perhaps include a function to simulate a candle input (for testing without real connection) – this can feed dummy data to the event emitter, used in offline tests.
- **Performance considerations:** Using one connection for all topics reduces overhead. KuCoin’s backend can handle it. The dataFeed should not perform heavy computations – just parsing and emitting. That parsing is trivial JSON handling.
- Example structure:

```
class DataFeed extends EventEmitter {
  constructor() { ... }
  async connect() { ... }
  subscribeCandles(symbol, interval) { ... }
  // internal: on message -> parse and emit event
  // ping handling, reconnect handling
}
```

Then in screenerEngine:

```
const feed = new DataFeed();
feed.on('newCandle', (symbol, timeframe, candle) => {
    // integrate with buffers & indicators
});
feed.connect().then(() => {
  config.symbols.forEach(sym => {
    feed.subscribeCandles(sym, config.primaryTimeframe);
    feed.subscribeCandles(sym, config.secondaryTimeframe);
  });
});
```

- Use logging (console or debug) for important events: connection open, error, reconnect attempts, etc., to facilitate debugging of the live feed.

With all components above, the screener module will fetch live data, compute indicators on the fly, and identify cross-timeframe aligned signals for potential trades.

Testing Prompts

To ensure each part of the screener works correctly and the module as a whole meets the requirements, use the following testing scenarios. These prompts can be given to the AI or used as guidelines to create test code.

Testing: Buffer Handling & Indicator Accuracy

Objective: Verify that candle buffering and incremental indicator calculations work as expected.

- **Buffer rollover test:** Simulate a sequence of more candles than the buffer limit. For example, if buffer limit is 100, feed 120 dummy 5m candles for a symbol and ensure that after insertion, the buffer length is capped at 100 (oldest 20 dropped). Prompt the assistant to generate a test function that pushes incremental candle data into a dummy buffer and asserts the length.
- **RSI correctness test:** Create known scenarios to compare the RSIIndicator output with an expected value:
- Feed a rising price series where RSI should gradually increase above 70, or a flat series where RSI ~50. For instance, input 14 periods of incremental gains (price +1 each candle) – the RSI should end

up near 100 (extremely overbought) [11]. Then input a price drop and see RSI fall accordingly. Have the test compute RSI both via the class and via a straightforward full-period calculation to cross-check.

- **MACD incremental vs batch test:** Generate a short array of prices and compute MACD manually or with a known formula, then compare with the MACDIndicator class outputs step by step. Ensure that the incremental EMA logic yields the same MACD line and signal line as a one-shot calculation (within a small epsilon error due to smoothing start).
- **Williams %R boundaries:** Test the WilliamsRIndicator with edge conditions:
- Create a scenario where the latest close is exactly the highest of the period (expected %R ≈ 0) and one where it's lowest (expected %R ≈ -100) [28]. For example, if period=4 and highs=[50,52,55,60], lows=[49,50,53,58], and current close = 58 (which is the low of window 58-60), %R should output ~-100. If close = 60 (the high), %R ~0. The test prompt can ask to simulate these and check outputs.
- **AO trend test:** Provide a short series of high/low values and verify AO:
- e.g. feed 34 candles with steadily increasing prices. The AO should be positive (fast SMA above slow SMA). Feed 34 candles decreasing – AO negative. Test that the class flips sign appropriately and that the magnitude roughly matches expectations (though not a specific value without full calc).
- Use **console assertions** or simple prints in the test prompts to confirm these conditions. The prompt should instruct to log intermediate values (like avgGain/avgLoss for RSI, or EMA values for MACD) for transparency.

(By verifying indicators in isolation, we ensure the building blocks are solid before integration.)

Testing: Alignment Detection Logic

Objective: Ensure the `timeframeAligner` correctly identifies alignment scenarios.

- **Bullish alignment scenario:** Simulate indicator values for two timeframes where alignment should trigger. For example:
- Timeframe1: RSI 25, MACD histogram +0.5, %R -85; Timeframe2: RSI 28, MACD hist +1.0, %R -90. Here all three indicators show bullish signals on both frames (RSIs <30, MACD >0, %R < -80). The aligner should report a bullish alignment with all three indicators listed.
- **Partial alignment scenario:** Simulate a case where only some indicators align:
- e.g. Timeframe1: RSI 40 (neutral), MACD >0 (bullish), %R -82 (bullish); Timeframe2: RSI 42 (neutral), MACD >0 (bullish), %R -50 (neutral). Here MACD is bullish on both, RSI is not oversold on either, %R is bullish on TF1 but not on TF2. If config requires 2 aligned and we only have MACD truly aligned, the result should be “no signal”. Test that the aligner does NOT produce a signal in this case.
- Then adjust one value (say Timeframe2 %R to -85) so that now MACD and %R are both aligned bullish, and ensure the aligner now flags a bullish signal with those two indicators.
- **Bearish alignment scenario:** Similar to above, but with bearish conditions:
- e.g. Timeframe1: RSI 75, MACD hist -0.4, %R -15; Timeframe2: RSI 80, MACD hist -1.2, %R -10. These indicate overbought RSI (>70) and negative MACD (downtrend) and %R near -0 (overbought) on both frames – aligner should detect bearish alignment.
- **Exact threshold handling:** Test edge values on thresholds (e.g. RSI exactly 30 or 70). Decide if 70 counts as overbought or not (likely >70 is overbought per config [5]). The aligner should consistently apply `>=` or `>` as defined. Write test cases for boundary values to ensure no off-by-one (for float this means just ensure the comparison logic is correct).

- The prompt for testing can direct to use a simple driver script where the aligner function is imported, fed with dummy indicator objects, and the output is printed to verify it matches expected outcome (e.g., “Expected: bullish alignment with [‘MACD’, ‘WilliamsR’]; Got: ...”).

Testing: End-to-End Integration with KuCoin WebSocket (Live Test)

Objective: Run the screener in a live or simulated environment to ensure it integrates properly with the WebSocket feed and emits signals.

- **WebSocket connection test:** Write a prompt to connect the DataFeed to a real (or test) KuCoin environment for a single symbol, single timeframe, for a short duration:
- Subscribe to a actively traded symbol (e.g. BTC-USDT) on 1-minute and 3-minute intervals. Let it run for a few minutes and observe console logs or collected data.
- Verify that candles are received and parsed correctly (print a few to ensure the format is as expected, matching the KuCoin data schema).
- Ensure pings are being sent and no disconnection occurs for that short test (if the connection remains stable for, say, 2+ minutes, ping/pong is working).
- **Full screener dry-run:** Using a small subset of symbols and fast timeframes:
- Configure `screenerConfig.js` for 2–3 symbols (perhaps test with known price patterns or use a stable coin for no signals).
- Run `screenerEngine` for a timeframe like 1min & 3min for a few cycles. If possible, manually create a scenario where an alignment might happen (this is tricky with live data – alternatively, modify alignment criteria temporarily to something easily met to force a signal, e.g., require only 1 indicator alignment).
- Check that when criteria are met, a log entry is written and console output appears (if enabled). If using a real feed, catching an alignment live is hit-or-miss; instead you might simulate by calling the alignment checker manually with fabricated indicator values.
- **Simulated feed test:** As an alternative to live testing, create a stub of DataFeed that emits pre-canned candle sequences designed to produce a signal. For example, generate a rising price sequence on both timeframes for one symbol such that RSI/MACD become bullish together. Feed those candles through the system (calling the same handlers) and verify a signal output occurs.
- Confirm that the `signalEmitter` writes the JSON line correctly. Perhaps open the `screener_matches.jsonl` after a test run and see that it’s well-formed JSON and contains the expected content.

(Note: When running against live data, ensure the API keys or endpoints are correctly configured if needed. Since we are using public data, no auth is required. Watch out for network errors or data spikes if the market is volatile.)

Testing: Performance and Stability

Objective: Evaluate that the screener can handle the load and stays within rate limits.

- **Throughput test:** If possible, simulate a high-frequency scenario (though candle feeds are limited by timeframe). One way is to use very low timeframe (like 1min) with many symbols. Configure 20 symbols on 1min & 3min timeframes and let it run for an hour. Monitor:
- CPU usage (should be low, mostly waiting on I/O).
- Memory usage (should not grow unbounded – the buffers should cap, and no leaks in indicator calculations).

- Check that the number of open WebSocket connections is 1 (or as expected) and subscriptions are all active. There should be no unintended multiple reconnects or duplicate subscriptions (which could be inferred if you see duplicate data or by adding logging in subscription calls).
- **Rate limit safety:** Although using WS, ensure that if the code ever falls back to REST (or uses REST for initial history), it doesn't spam requests:
- Test the REST path by deliberately disabling WS (set `useWebSocket:false` in config) and enabling `useRestFallback:true` with a small poll interval. Possibly throttle it to a safe value (like polling each symbol every 10 seconds). Run for a short time and verify no HTTP 429 errors from KuCoin (which would indicate hitting rate limits).
- Check that on WS, the number of outgoing messages is reasonable: initial subscribe messages (2 per symbol) plus periodic pings. KuCoin allows 100 messages per 10s [29] – our usage ~40 subs + ping – well below that. If testing with logs, you shouldn't see any warnings about "exceeds max subscription count".
- **Concurrent operations:** If possible, run the screener in parallel with the main trading bot (assuming the bot has its own process or thread). This is more of an integration test:
- Ensure that both can run without interfering (e.g. if they share event loops or outputs). If the screener is writing the log file, the main bot could be configured to read it (in a real integration scenario). That could be tested by checking the file contents mid-run by the main bot.
- If using event emitter integration and the screener is in the same process, attach a temporary listener in the main bot to the screener's events and verify it receives them.
- You can prompt the AI to generate a summary report of performance, for example by instrumenting the code with counters (count candles processed per minute) and then outputting those stats. Or simply instruct to observe that with 20 symbols * 2 timeframes, you expect ~40 candles per minute on 1min timeframe, which is trivial for Node.js to handle.

By running these tests and verifying the outcomes, you will ensure the dual-timeframe screener is robust, accurate, and efficient. The final product will be a production-ready module that can feed high-quality trade signals to your v3.5 trading dashboard based on cross-timeframe indicator alignment. Use the prompts above to systematically build and verify each component. Good luck, and happy screening!

Sources:

- KuCoin WebSocket API Docs – candle subscription format and data schema [1] [24]
- KuCoin API Rate Limit and WebSocket guidelines [30] [2]
- Technical indicator formulas and thresholds: RSI calculation (Wilder's method) [9], EMA update rule [12], Williams %R formula [15], Awesome Oscillator definition [17], standard overbought/oversold levels [5] [7].

---

[1] [19] [21] [23] [24] [25] Klines - KUCOIN API - AU
https://www.kucoin.com/en-au/docs-new/3470071w0

[2] [22] [29] [30] Rate Limit - KUCOIN API
https://www.kucoin.com/docs-new/rate-limit?lang=en_US&

[3] [20] Introduction - KUCOIN API
https://www.kucoin.com/docs-new/websocket-api/base-info/introduction

[4] Introduction - KUCOIN API
https://www.kucoin.com/docs-new/websocket-api/base-info/introduction?lang=en_US&

[5] [6] [7] [8] WEIGHT_ADJUSTMENT_GUIDE.md
file://file_000000007c5c71fdaf7e5effb2ff8603

[9] [10] [11] Relative Strength Index (RSI) | ChartSchool | StockCharts.com
https://chartschool.stockcharts.com/table-of-contents/technical-indicators-and-overlays/technical-indicators/relative-strength-index-rsi

[12] Exponential Moving Average (EMA): Definition, Formula, and Usage
https://www.investopedia.com/terms/e/ema.asp

[13] [14] What Is MACD?
https://www.investopedia.com/terms/m/macd.asp

[15] [28] Williams %R - Wikipedia
https://en.wikipedia.org/wiki/Williams_%25R

[16] [17] [18] Awesome Oscillator | TrendSpider Learning Center
https://trendspider.com/learning-center/awesome-oscillator/

[26] [27] Get Klines - KUCOIN API
https://www.kucoin.com/docs-new/rest/futures-trading/market-data/get-klines?lang=en_US&