IB Gateway Futures Charting: Critical Technical Answers

Building a TradingView-quality futures charting webapp requires understanding four key implementation details. This report provides definitive, code-ready answers based on official documentation, real user experiences, and production implementations.

IB Gateway's keepUpToDate doesn't stream ticks—it updates every 5 seconds

The most critical finding: **`keepUpToDate=True` does NOT provide sub-second updates**. Per Interactive Brokers' official documentation, even when requesting 1-minute bars with `keepUpToDate=True`, you receive updates of the most recent partial **five-second bar**—not tick-by-tick data. github This means approximately **12 updates per minute**, not multiple updates per second.

The minimum bar size for `keepUpToDate` is 5 seconds. You cannot request 1-second bars with real-time updates through this mechanism. For true tick-level updates on MNQ, MGC, and MES futures, you must use a hybrid approach:

```python
```

```
from ib_insync import *

class LiveCandlestickBuilder:
    def __init__(self, ib, contract, bar_size_minutes=1):
        self.ib = ib
        self.contract = contract
        self.bar_size = timedelta(minutes=bar_size_minutes)
        self.current_bar = {'open': None, 'high': float('-inf'),
                            'low': float('inf'), 'close': None, 'volume': 0}

    def start(self):
        # 1. Fetch historical bars ONCE (not keepUpToDate)
        self.bars = self.ib.reqHistoricalData(
            self.contract, endDateTime='', durationStr='1 D',
            barSizeSetting='1 min', whatToShow='TRADES',
            useRTH=False, keepUpToDate=False
        )

        # 2. Stream true tick-by-tick for live updates
        self.ticker = self.ib.reqTickByTickData(
            self.contract, 'AllLast',  # Captures all trade types
            numberOfTicks=0, ignoreSize=False
        )
        self.ticker.updateEvent += self._on_tick

    def _on_tick(self, ticker):
        for tick in ticker.tickByTicks:
            price, size = tick.price, tick.size
            if self.current_bar['open'] is None:
                self.current_bar['open'] = price
            self.current_bar['high'] = max(self.current_bar['high'], price)
            self.current_bar['low'] = min(self.current_bar['low'], price)
            self.current_bar['close'] = price
            self.current_bar['volume'] += size
            self._push_to_websocket(self.current_bar)  # Real-time chart update
```

Use `reqTickByTickData()` with `'AllLast'` rather than `reqMktData()`. The latter provides aggregated snapshots "several times per second" (~300ms intervals), while tick-by-tick gives you every trade. Expected latency from CME exchange to your application: **50-300ms** depending on network location, identical between paper and live accounts when market data sharing is enabled.

Critical caveat: IB documentation warns that `reqHistoricalData + keepUpToDate` can leave the entire API inoperable after network interruptions. For production systems, the tick-aggregation pattern above is more robust.

Three concrete alternatives for simpler indicator syntax

The user wants "like Pine Script but not literally Pine"—simpler than full Python classes. Here are three production-ready approaches, ranked from simplest to most flexible:

Option A: Expression-based DSL (most Pine-like, 1-2 lines per indicator)

python

```python
from simpleeval import SimpleEval

class IndicatorEvaluator:
    def __init__(self, df):
        self.eval = SimpleEval()
        self.eval.names = {
            'close': df['close'], 'open': df['open'],
            'high': df['high'], 'low': df['low'], 'volume': df['volume']
        }
        self.eval.functions = {
            'EMA': lambda s, p: s.ewm(span=p, adjust=False).mean(),
            'SMA': lambda s, p: s.rolling(p).mean(),
            'RSI': self._rsi, 'MACD': self._macd,
        }

    def calc(self, expr):
        return self.eval.eval(expr)

# Usage - Pine-like syntax
evaluator = IndicatorEvaluator(df)
df['trend'] = evaluator.calc("EMA(close, 20) - EMA(close, 50)")
df['signal'] = evaluator.calc("EMA(close, 9)")
```

Use `simpleeval` or `asteval` libraries—never raw `eval()`. Whitelist functions explicitly PyPI and set computation timeouts.

Option B: Minimal decorator pattern (5-10 lines, full Python power)

python

```
from functools import wraps

def indicator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    wrapper._is_indicator = True
    return wrapper

@indicator
def EMA(close, period=20):
    return close.ewm(span=period, adjust=False).mean()

@indicator
def MACD(close, fast=12, slow=26, signal=9):
    fast_ema = close.ewm(span=fast, adjust=False).mean()
    slow_ema = close.ewm(span=slow, adjust=False).mean()
    macd_line = fast_ema - slow_ema
    signal_line = macd_line.ewm(span=signal, adjust=False).mean()
    return {'macd': macd_line, 'signal': signal_line, 'hist': macd_line - signal_line}

# Usage
df['ema20'] = EMA(df['close'], 20)
```

Option C: YAML config for non-programmers

```yaml
indicators:
  - name: ema_20
    type: ema
    params: {period: 20, source: close}
  - name: trend
    type: expression
    formula: "EMA(close, 20) - EMA(close, 50)"
```

Best existing library: The **Jesse framework** has the cleanest API found—`ta.ema(candles, 20)` style with 300+ built-in indicators. For dict-based configuration, **pandas_ta's Strategy class** already works without custom parsing.

Volume bar coloring uses same-candle comparison by default

TradingView's default: `close >= open` of the **SAME candle** determines color. Green if the candle closed at or above its open (bullish), red if below (bearish).

This is confirmed by TradingView's official documentation, which describes an optional "Color Based On Previous Close" setting—the phrase "when enabled" proves the default uses same-candle comparison. GetSatisfaction

```javascript
function getVolumeBarColor(candle) {
  // TradingView default: same-candle comparison
  // Note: >= means doji candles (open === close) are GREEN
  return candle.close >= candle.open ? '#26a69a' : '#ef5350';
}

// With optional previous-close method
function getVolumeBarColorAlt(candle, prevCandle, usePrevClose = false) {
  if (usePrevClose && prevCandle) {
    return candle.close > prevCandle.close ? '#26a69a' : '#ef5350';
  }
  return candle.close >= candle.open ? '#26a69a' : '#ef5350';
}
```

Edge cases resolved:

- **Doji (open = close)**: Green bar (because `>=` is used)
- **Gap up with red body**: Red bar (only current bar's open/close matters)
- **Gap down with green body**: Green bar
- **First bar of session**: Uses same-candle logic; no previous close needed

Sierra Chart and NinjaTrader use identical logic. ThinkOrSwim is the exception, defaulting to previous-close comparison.

375,000 candles will crash mobile and degrade desktop performance

The claim that Lightweight Charts can "smoothly handle" 375K candles is **not realistic without aggressive optimization**. Real benchmarks from GitHub issues:

Dataset Size	Desktop Performance	Mobile Performance
~60,000 candles	Smooth scrolling/zooming	Borderline
15,000+ with markers	Lag begins	Unusable
1 million points	Works but heavy	Crashes iPhone 13 Pro "every third reload"

TradingView's own documentation says charts work with "thousands of bars"—notably not "hundreds of thousands." TradingView Their production platform limits free users to 5,000-10,000 candles.

Data conflation is real but insufficient alone

Version 5.1.0 introduced conflation, which merges data points when bar spacing falls below 0.5 pixels:

javascript

```javascript
const chart = createChart(container, {
  timeScale: {
    enableConflation: true,
    conflationThresholdFactor: 1.0,  // Performance-focused
    precomputeConflationOnInit: true  // +100-500ms load, 10-100x faster zoom
  }
});
```

This provides **10-100x improvement** for zoom operations but doesn't solve memory limits. The `uniformDistribution` option only affects time axis label rendering, not data performance.

The production solution: lazy loading + timeframe aggregation

No production TradingView-style app loads 375K candles at once. The standard pattern:

javascript

```
// Pre-compute multiple timeframes server-side
// Show appropriate resolution based on zoom level
chart.timeScale().subscribeVisibleLogicalRangeChange(range => {
  const barsVisible = range.to - range.from;

  let timeframe;
  if (barsVisible > 5000) timeframe = '1D';      // ~260 bars/year
  else if (barsVisible > 1000) timeframe = '1H'; // ~6,500 bars/year
  else if (barsVisible > 200) timeframe = '5m';  // ~78,000 bars/year
  else timeframe = '1m';                         // Full resolution

  fetchAndDisplayData(timeframe, range.from - 50, range.to + 50);
});
```

Load only the visible range plus a 50-bar buffer. When users scroll left (into history), fetch more data on demand using the infinite history pattern from Lightweight Charts' official tutorials.

Practical limits: Keep **under 60,000 candles** loaded at any time on desktop, **under 20,000** for reliable mobile support. iOS Safari has aggressive canvas memory limits that cause crashes with multiple charts or large datasets. github

Implementation architecture summary

For a production-quality futures charting webapp:

1. **Real-time data**: Use `reqTickByTickData('AllLast')` + manual candlestick aggregation—not `keepUpToDate`. Expect 50-300ms latency.
2. **Indicators**: Start with the expression DSL using `simpleeval` for user-defined indicators; fall back to decorated Python functions for complex logic.
3. **Volume bars**: `close >= open` for same-candle coloring. Doji = green.
4. **Data handling**: Pre-aggregate into 5m/15m/1H/1D bars server-side. Load max 60K candles at once. Implement lazy loading with `subscribeVisibleLogicalRangeChange`. Enable conflation in Lightweight Charts v5.1.0+.

This architecture delivers TradingView-quality responsiveness while avoiding the memory and performance pitfalls of naive full-dataset loading.