📊 DIAGNÓSTICO PROFESIONAL: BACKTESTING ENGINE

Basado en mi análisis exhaustivo del código actual y las tecnologías disponibles, te presento un diagnóstico completo con múltiples opciones técnicas.

🎯 ANÁLISIS DE REQUERIMIENTOS

Funcionalidades Core

1. **Datos históricos**: 3 años de datos en caché (sin llamadas en tiempo real)
2. **Timeframes**: 15m, 1h, 4h con subdivisiones para simular formación de velas
3. **Reproducción temporal**: Control play/pause/velocidad desde fecha seleccionada
4. **Sincronización**: Múltiples gráficos sincronizados (mismo símbolo, diferentes timeframes)
5. **Herramientas de dibujo**: Líneas, rectángulos, cajas TP/SL, texto
6. **Persistencia**: Guardar dibujos e indicadores entre sesiones
7. **Reutilización**: Mismos indicadores y visualización de la watchlist

Complejidad Estimada

- **Alta**: Sincronización multi-pestaña, simulación de velas en tiempo "real"
- **Media**: Sistema de dibujo, gestión de caché masivo (3 años de datos)
- **Baja**: Reutilización de indicadores existentes

📐 ARQUITECTURA ACTUAL (Fortalezas y Limitaciones)

✅ Fortalezas a Aprovechar

1. **Canvas 2D nativo**: Perfecto para dibujo - ya tienen experiencia rendering manual
2. **Sistema de indicadores modular**: Clase base `IndicatorBase.js` altamente reutilizable
3. **Backend FastAPI**: Fácil agregar endpoints para gestión de caché histórico
4. **localStorage**: Ya usado para fixed ranges, extendible para persistencia de dibujos

⚠ Limitaciones a Resolver

1. **WebSocket innecesario**: El backtesting no necesita datos en tiempo real

2. **Límites de días por timeframe**: Actualmente limitados (ej: 5min = 5 días) - necesitan 3 años
3. **No hay sistema de reproducción temporal**: Todo es en tiempo real
4. **Canvas sin capas**: Dibujos y velas en mismo canvas dificulta edición de objetos

---

🛠️ OPCIONES TÉCNICAS POR COMPONENTE

1️⃣ GESTIÓN DE DATOS HISTÓRICOS

Opción A: Caché JSON en Backend (RECOMENDADA)

Pros:

- Consistente con arquitectura actual
- Fácil implementación: extender `/backend/cache/`
- 3 años × 3 timeframes × subdivisiones ≈ 500MB-1GB (manejable)

Cons:

- Primer load lento (descargar 3 años de datos)

Implementación:

```
# Nuevo endpoint: /api/historical/bulk/{symbol}
# Retorna: {
#   "15m": [...],  # 3 años de datos
#   "5m": [...],   # Para subdivisiones
#   "1h": [...],
#   "15m_subdivisions": [...],
#   "4h": [...],
#   "1h_subdivisions": [...]
# }
```

Opción B: IndexedDB en Frontend

Pros:

- Almacenamiento local masivo (gigabytes disponibles)
- Sin límites de localStorage (5MB)
- Queries rápidas con índices

Cons:

- Complejidad adicional en frontend
- API asíncrona más compleja que localStorage

Opción C: SQLite en Backend

Pros:

- Base de datos real, queries optimizadas
- Escalable a múltiples símbolos

Cons:

- Sobre-ingeniería para este caso de uso
- Requiere migraciones, gestión de DB

💡 **RECOMENDACIÓN: Opción A + IndexedDB**

- Descargar datos desde backend
- Almacenar en IndexedDB del navegador
- Evita re-descargas en futuras sesiones

2️⃣ SIMULACIÓN DE FORMACIÓN DE VELAS

Opción A: Interpolación Temporal con Subdivisiones

Pros:

- Realista: usa datos reales de subdivisiones (5m para formar 15m)
- Crea sensación de mercado en vivo

Implementación:

```
// Para vela de 15min (13:00-13:15)
// Renderizar progresivamente:
// 1. Datos de 13:00-13:05 (primera subdivisión)
// 2. Datos de 13:05-13:10 (segunda subdivisión)
// 3. Datos de 13:10-13:15 (tercera subdivisión)
// 4. Vela completa de 15min
```

Cons:

- 3x-4x más datos a almacenar y procesar

Opción B: Animación Artificial (Sin Subdivisiones)

Pros:

- Menos datos (solo timeframes principales)
- Más simple de implementar

Cons:

- Menos realista: no simula volatilidad intra-vela
- Usuario no ve cómo se forma la mecha

💡 **RECOMENDACIÓN: Opción A**

- El realismo justifica el costo de datos adicionales
- Crítico para entrenamiento efectivo

3️⃣ CONTROL DE REPRODUCCIÓN TEMPORAL

Arquitectura Propuesta:

```
class BacktestingTimeController {
  constructor(startDate, endDate, speed) {
    this.currentTime = startDate;
    this.endTime = endDate;
    this.playbackSpeed = speed; // 1x, 2x, 5x, 10x, etc.
    this.isPaused = true;
  }

  play() {
    this.isPaused = false;
    this.interval = setInterval(() => {
      this.currentTime += this.getTimeIncrement();
      this.notifySubscribers(); // Actualiza todos los gráficos
    }, this.getUpdateInterval());
  }

  // Con speed=1x y timeframe=15m:
  // getUpdateInterval() = 1000ms (actualizar cada segundo)
  // getTimeIncrement() = 15min (avanzar 15min cada segundo)
}
```

Ventajas:

- Control centralizado del tiempo
- Fácil sincronizar múltiples gráficos
- Pausar/reanudar trivial

4️⃣ SINCRONIZACIÓN MULTI-PESTAÑA

Opción A: BroadcastChannel API (RECOMENDADA)

Pros:

- API nativa del navegador
- Comunicación entre pestañas del mismo origen
- Ligero y eficiente

```
// Pestaña A (Master)
const bc = new BroadcastChannel('backtesting_sync');
bc.postMessage({
  event: 'timeUpdate',
  currentTime: 1672531200000,
  isPaused: false
});

// Pestaña B (Slave)
bc.onmessage = (event) => {
  if (event.data.event === 'timeUpdate') {
    this.timeController.jumpTo(event.data.currentTime);
  }
};
```

Cons:

- No soportado en IE (irrelevante en 2025)

Opción B: localStorage + Storage Events

Pros:

- Mayor compatibilidad (navegadores antiguos)

Cons:

- Más hacky, menos eficiente
- Límite de 5MB

💡 **RECOMENDACIÓN: BroadcastChannel API**

5️⃣ HERRAMIENTAS DE DIBUJO SOBRE CANVAS

Opción A: Fabric.js (RECOMENDADA PARA INICIO RÁPIDO)

Pros:

- 27.8k estrellas en GitHub, muy maduro
- Sistema de objetos sobre canvas ("canvas dentro de canvas")
- Drag & drop, resize, rotate out-of-the-box
- Serialización JSON (perfecto para persistencia)
- Eventos de mouse/teclado ya manejados

```
const canvas = new fabric.Canvas('backtest-canvas');

// Dibujar línea
const line = new fabric.Line([50, 100, 200, 200], {
  stroke: 'red',
  strokeWidth: 2
});
canvas.add(line);

// Dibujar rectángulo (TP/SL box)
const rect = new fabric.Rect({
  left: 100,
  top: 100,
  fill: 'rgba(0,255,0,0.2)',
  width: 200,
  height: 50
});
canvas.add(rect);

// Serializar para persistencia
const json = canvas.toJSON();
localStorage.setItem('drawings', JSON.stringify(json));
```

Cons:

- 250KB minified (aumenta bundle size)
- Curva de aprendizaje para integrar con tu rendering actual

Opción B: Konva (ALTERNATIVA MODERNA)

Pros:

- 10.9k estrellas, más moderno que Fabric.js
- Arquitectura de capas nativa (Stage → Layer → Shapes)
- Excelente performance con animaciones
- TypeScript support

```
const stage = new Konva.Stage({
  container: 'container',
  width: window.innerWidth,
  height: window.innerHeight
});
```

```
const layer = new Konva.Layer();
const line = new Konva.Line({
  points: [50, 100, 200, 200],
  stroke: 'red',
  strokeWidth: 2,
  draggable: true
});
layer.add(line);
stage.add(layer);
```

Cons:

- Menos ecosistema de plugins que Fabric.js
- ~170KB minified

Opción C: Implementación Nativa (MÁXIMO CONTROL)

Pros:

- Cero dependencias adicionales
- Control total sobre rendering
- Bundle size mínimo

Implementación básica:

```
class DrawingManager {
  constructor(canvas) {
    this.canvas = canvas;
    this.ctx = canvas.getContext('2d');
    this.drawings = []; // Array de objetos de dibujo
    this.activeDrawing = null;
    this.setupEvents();
  }

  setupEvents() {
    this.canvas.addEventListener('mousedown', this.handleMouseDown);
    this.canvas.addEventListener('mousemove', this.handleMouseMove);
    this.canvas.addEventListener('mouseup', this.handleMouseUp);
  }

  addLine(x1, y1, x2, y2) {
    this.drawings.push({
      type: 'line',
      x1, y1, x2, y2,
      color: '#ff0000',
      width: 2
    });
    this.redraw();
  }
```

```
  redraw() {
    // 1. Limpiar canvas
    this.ctx.clearRect(0, 0, this.canvas.width, this.canvas.height);

    // 2. Redibujar velas (desde tu código actual)
    this.drawCandlesticks();

    // 3. Redibujar dibujos encima
    this.drawings.forEach(drawing => {
      if (drawing.type === 'line') {
        this.ctx.beginPath();
        this.ctx.moveTo(drawing.x1, drawing.y1);
        this.ctx.lineTo(drawing.x2, drawing.y2);
        this.ctx.strokeStyle = drawing.color;
        this.ctx.lineWidth = drawing.width;
        this.ctx.stroke();
      }
    });
  }
}
```

Cons:

- Mucho desarrollo manual (drag & drop, resize, detección de hover)
- No hay serialización automática
- Meses de desarrollo para igualar Fabric.js/Konva

Opción D: Sistema Híbrido - Canvas Multicapa

Arquitectura:

```
<div style="position: relative">
  <!-- Capa 1: Candlesticks e indicadores (tu código actual) -->
  <canvas id="chart-layer" style="position: absolute; z-index: 1"></canvas>

  <!-- Capa 2: Dibujos (Fabric.js o Konva) -->
  <canvas id="drawing-layer" style="position: absolute; z-index: 2"></canvas>

  <!-- Capa 3: UI overlays (botones, tooltips) -->
  <div id="ui-layer" style="position: absolute; z-index: 3"></div>
</div>
```

Pros:

- Separación de responsabilidades
- Tu código de velas no se toca
- Dibujos independientes del chart
- Mejor performance (no re-render innecesario)

Cons:

- Coordenadas deben sincronizarse entre capas (zoom/pan)

💡 **RECOMENDACIÓN: Fabric.js con Canvas Multicapa**

- **Fase 1**: Implementar con Fabric.js (2-3 semanas)
- **Fase 2 (opcional)**: Migrar a implementación nativa si bundle size es problema (2-3 meses)

---

6 PERSISTENCIA DE DIBUJOS

Opción A: localStorage + JSON

Pros:

- Ya lo usan para fixed ranges
- Integración trivial con Fabric.js (`canvas.toJSON()`)

```
// Guardar
const state = {
  symbol: 'BTCUSDT',
  timeframe: '15m',
  drawings: canvas.toJSON(),
  indicators: this.getIndicatorConfig(),
  lastSavedDate: Date.now()
};
localStorage.setItem(`backtesting_${symbol}`, JSON.stringify(state));

// Cargar
const saved = JSON.parse(localStorage.getItem(`backtesting_BTCUSDT`));
canvas.loadFromJSON(saved.drawings);
```

Cons:

- Límite de 5MB por dominio (puede llenarse con muchos dibujos)

Opción B: IndexedDB

Pros:

- Sin límites prácticos de storage
- Queries por símbolo, timeframe, etc.

Cons:

- API más compleja (async)

💡 **RECOMENDACIÓN:**

- **Inicio:** localStorage (simple, rápido)
- **Si se llena:** Migrar a IndexedDB

🎨 COMPARACIÓN: CÓMO LO HACEN OTROS

TradingView

- **Tecnología**: Proprietary canvas engine
- **Dibujos**: ~50 herramientas (líneas, Fibonacci, pitchforks)
- **Persistencia**: Backend (cuenta de usuario)

Exocharts

- **Tecnología**: Canvas 2D (removieron WASM por performance issues)
- **Dibujos**: Brush, horizontal ray, text, magnet mode
- **Sincronización**: Magnet snap to price

GoCharting

- **Tecnología**: Canvas con objetos
- **Feature clave**: Draw Mode (dibujar múltiples sin re-seleccionar tool)
- **Persistencia**: Local + nube opcional

ATAS

- **Tecnología**: DirectX/OpenGL (app desktop)
- **No aplicable**: Diferente stack (no web)

📋 STACK TECNOLÓGICO RECOMENDADO

Backend

```
FastAPI (actual)
+ SQLite (opcional, para múltiples símbolos)
+ Endpoints nuevos:
  - POST /api/backtesting/cache/{symbol}  # Cachear 3 años
  - GET  /api/backtesting/data/{symbol}   # Obtener datos cacheados
  - GET  /api/backtesting/drawings/{symbol}  # Persistencia en backend
(opcional)
```

Frontend

```
React 18 (actual)
+ Fabric.js 5.x (drawing tools)
+ IndexedDB API (caché masivo de datos)
+ BroadcastChannel API (sync multi-pestaña)
+ Tu código actual de MiniChart.jsx (rendering de velas)
```

🚀 PLAN DE IMPLEMENTACIÓN SUGERIDO

Fase 1: MVP (2-3 semanas)

1. ✅ Endpoint backend para descargar 3 años de datos
2. ✅ Componente `BacktestingChart.jsx` (fork de `MiniChart.jsx`)
3. ✅ `TimeController` para reproducción temporal
4. ✅ Integración básica de Fabric.js (líneas y rectángulos)
5. ✅ Persistencia en localStorage

Fase 2: Features Avanzadas (2-3 semanas)

6. ✅ Subdivisiones de velas (animación realista)
7. ✅ Control de velocidad (1x, 2x, 5x, 10x)
8. ✅ Sincronización multi-pestaña (BroadcastChannel)
9. ✅ Más herramientas de dibujo (texto, Fibonacci)

Fase 3: Optimización (1-2 semanas)

10. ✅ Migrar caché a IndexedDB
11. ✅ Lazy loading de datos (solo cargar velas visibles)
12. ✅ Optimización de rendering (requestAnimationFrame)

💰 ANÁLISIS DE COSTOS/BENEFICIOS

Aspecto	Opción Recomendada	Alternativa	Trade-off
Dibujo	Fabric.js (+250KB)	Nativo (0KB)	Tiempo vs Bundle
Caché	IndexedDB (ilimitado)	localStorage (5MB)	Complejidad vs Límites
Sync	BroadcastChannel (moderno)	localStorage events	Performance vs Compatibilidad
Datos	3 años × 3 TF (~1GB)	Solo TF principal (~350MB)	Realismo vs Storage

---

RIESGOS Y MITIGACIONES

Riesgo 1: Bundle Size Excesivo

- **Mitigación**: Code splitting, lazy load de Fabric.js solo en módulo de backtesting

Riesgo 2: Performance con 3 Años de Datos

- **Mitigación**: Virtualización (solo renderizar velas visibles en viewport)

Riesgo 3: Sincronización Desfasada

- **Mitigación**: Heartbeat cada 500ms para verificar estado entre pestañas

---

🎯 RECOMENDACIÓN FINAL

Stack Óptimo:

1. **Datos**: Backend (FastAPI) → IndexedDB (frontend)
2. **Dibujo**: Fabric.js + Canvas multicapa
3. **Reproducción**: `TimeController` custom con subdivisiones
4. **Sincronización**: BroadcastChannel API
5. **Persistencia**: localStorage (inicio) → IndexedDB (escalamiento)

Justificación:

- ✅ Reutiliza 80% del código actual (`MiniChart.jsx`, indicadores)
- ✅ Implementación rápida (4-6 semanas para producto completo)
- ✅ Escalable y mantenible
- ✅ No depende de TradingView ni APIs propietarias
- ✅ Performance adecuado para 3 años de datos

🤝 PRÓXIMOS PASOS

¿Qué enfoque te gustaría tomar?

Opción A: Comenzar con MVP (Fase 1) usando stack recomendado **Opción B**: Explorar alternativa específica (ej: implementación nativa de dibujo) **Opción C**: Profundizar en algún aspecto técnico antes de decidir

¿Qué te parece este análisis? ¿Hay algún aspecto que quieras que profundice o alguna opción técnica adicional que quieras explorar?

Update Todos

- Analizar requerimientos del backtesting engine
- Explorar arquitectura actual de la watchlist
- Investigar tecnologías de dibujo sobre canvas
- Crear diagnóstico profesional con opciones técnicas