Complete TradingView-like Platform Implementation Guide

📋 Pre-Implementation Checklist

Environment Setup

- [ ] Python 3.9+ installed
- [ ] Node.js 16+ for frontend components
- [ ] Git repository cloned and accessible
- [ ] Virtual environment created
- [ ] Required system dependencies (build tools, etc.)

API Access

- [ ] Binance API keys obtained (if needed for authenticated endpoints)
- [ ] Binance WebSocket endpoint accessible
- [ ] Test connection to Binance streams

Development Tools

- [ ] IDE/Editor configured
- [ ] Database tools installed
- [ ] Browser developer tools familiar
- [ ] Testing framework ready

🏗️ Phase 1: Project Structure & Core Dependencies

1.1 Directory Structure Setup

```
zulubuntu/
├── apps/
│   ├── trading_platform/
│   │   ├── __init__.py
│   │   ├── models.py              # Data models
│   │   ├── websocket_manager.py   # WebSocket handling
│   │   ├── data_manager.py        # DuckDB operations
│   │   ├── indicators.py          # Technical indicators
│   │   ├── chart_manager.py       # Chart operations
│   │   └── utils/
│   │       ├── __init__.py
│   │       ├── logger.py          # Custom logging
│   │       ├── progress_bar.py    # Custom progress bars
│   │       ├── error_handler.py   # Error management
│   │       └── notifications.py   # Email/Telegram (commented)
├── templates/
│   ├── trading/
│   │   ├── base.html
│   │   ├── dashboard.html
│   │   └── chart.html
├── static/
│   ├── css/
│   ├── js/
│   └── assets/
├── config/
│   ├── settings.py
│   ├── database.py
│   └── websocket_config.py
├── data/                          # DuckDB files
├── logs/                          # Application logs
├── requirements.txt
└── main.py                        # Application entry point
```

1.2 Core Dependencies Installation

Why Each Library:

- `lightweight-charts`: Core charting engine, TradingView-compatible
- `duckdb`: Lightning-fast analytical database for OHLC storage
- `polars`: Blazing-fast DataFrame operations with Arrow backend
- `websockets`: Real-time Binance data streaming
- `fastapi`: High-performance async API framework
- `uvicorn`: ASGI server for FastAPI
- `tqdm`: Beautiful progress bars with customization
- `rich`: Enhanced terminal output and tables
- `yagmail`: Email notifications (commented out initially)
- `python-telegram-bot`: Telegram notifications (commented out)

bash

```bash
# Core trading platform
pip install lightweight-charts
pip install duckdb
pip install polars
pip install pyarrow  # Required for Polars-DuckDB integration

# Web framework and real-time communication
pip install fastapi
pip install uvicorn
pip install websockets
pip install jinja2  # For templates

# Data processing and networking
pip install httpx
pip install pandas  # Still useful for some operations
pip install numpy
pip install ta-lib  # Technical analysis library

# UI and progress visualization
pip install tqdm
pip install rich
pip install tabulate

# Error handling and notifications (initially commented)
pip install yagmail
pip install python-telegram-bot

# Development and testing
pip install pytest
pip install black
pip install flake8
```

🏗️ Phase 2: Data Management Layer (DuckDB + Polars)

2.1 Database Schema Design

Why This Approach:

- Each symbol+interval gets its own DuckDB file for optimal performance
- Columnar storage provides blazing-fast queries for backtesting
- Polars integration enables lightning-speed analytics

python

```python
# apps/trading_platform/models.py
"""
Data models for trading platform
Defines the structure for OHLC data and indicators storage
"""
from dataclasses import dataclass
from datetime import datetime
from typing import Optional, Dict, Any
import polars as pl
import duckdb

@dataclass
class OHLCData:
    """
    📊 OHLC Data Structure
    Represents a single candlestick with all associated indicators
    """
    timestamp: datetime
    symbol: str
    interval: str
    open: float
    high: float
    low: float
    close: float
    volume: float

    # Technical Indicators
    ema_12: Optional[float] = None
    ema_26: Optional[float] = None
    rsi: Optional[float] = None
    macd: Optional[float] = None
    macd_signal: Optional[float] = None
    macd_histogram: Optional[float] = None
    bb_upper: Optional[float] = None
    bb_middle: Optional[float] = None
    bb_lower: Optional[float] = None
    stoch_k: Optional[float] = None
    stoch_d: Optional[float] = None

class DatabaseSchema:
    """
    🗄️ Database Schema Manager
    Handles DuckDB table creation and management
    """

    @staticmethod
    def get_table_schema() -> str:
        """
        Returns the SQL schema for OHLC + indicators table
        Optimized for fast queries and minimal storage
        """
        return """
        CREATE TABLE IF NOT EXISTS ohlc_data (
            timestamp TIMESTAMP PRIMARY KEY,
            symbol VARCHAR NOT NULL,
            interval VARCHAR NOT NULL,
            open DOUBLE NOT NULL,
            high DOUBLE NOT NULL,
            low DOUBLE NOT NULL,
            close DOUBLE NOT NULL,
            volume DOUBLE NOT NULL,
```

```
            -- Technical Indicators
            ema_12 DOUBLE,
            ema_26 DOUBLE,
            rsi DOUBLE,
            macd DOUBLE,
            macd_signal DOUBLE,
            macd_histogram DOUBLE,
            bb_upper DOUBLE,
            bb_middle DOUBLE,
            bb_lower DOUBLE,
            stoch_k DOUBLE,
            stoch_d DOUBLE,

            -- Indexing for fast queries
            INDEX idx_timestamp (timestamp),
            INDEX idx_symbol_interval (symbol, interval)
        );
        """
```

2.2 Data Manager Implementation

python

```python
# apps/trading_platform/data_manager.py
"""
🔧 Data Manager
Handles all database operations using DuckDB + Polars
Provides blazing-fast data storage and retrieval
"""
import os
import duckdb
import polars as pl
from datetime import datetime, timedelta
from typing import List, Optional, Tuple
from pathlib import Path

from .models import OHLCData, DatabaseSchema
from .utils.logger import get_logger
from .utils.progress_bar import CustomProgressBar

logger = get_logger(__name__)

class DataManager:
    """
    💾 High-Performance Data Manager

    Features:
    - Individual DuckDB files per symbol+interval
    - Polars integration for lightning-fast queries
    - Automatic data compression and optimization
    - Seamless historical data loading
    """

    def __init__(self, data_dir: str = "data"):
        """
        Initialize data manager with storage directory
        Creates directory structure if it doesn't exist
        """
        self.data_dir = Path(data_dir)
        self.data_dir.mkdir(exist_ok=True)

        # 📊 Connection pool for database files
        self._connections: Dict[str, duckdb.DuckDBPyConnection] = {}

        logger.info(f"🚀 DataManager initialized with directory: {self.data_dir}")
        print(f"✅ Data storage ready at: {self.data_dir.absolute()}")

    def _get_db_path(self, symbol: str, interval: str) -> Path:
        """
        Generate database file path for symbol+interval combination
        Format: data/BTCUSDT_1m.duckdb
        """
        filename = f"{symbol}_{interval}.duckdb"
        return self.data_dir / filename

    def _get_connection(self, symbol: str, interval: str) -> duckdb.DuckDBPyConnection
        """
        Get or create database connection for symbol+interval
        Implements connection pooling for performance
        """
        key = f"{symbol}_{interval}"

        if key not in self._connections:
            db_path = self._get_db_path(symbol, interval)
```

```
            # 🔗 Create new connection
            conn = duckdb.connect(str(db_path))

            # 📋 Initialize schema
            conn.execute(DatabaseSchema.get_table_schema())

            self._connections[key] = conn
            logger.info(f"📊 New database connection: {key}")
            print(f"🔗 Connected to database: {db_path.name}")

        return self._connections[key]

    def store_ohlc_batch(self, data_batch: List[OHLCData]) -> None:
        """
        🚀 Store batch of OHLC data with indicators
        Uses Polars for maximum performance
        """
        if not data_batch:
            return

        # Group by symbol+interval for efficient storage
        grouped_data = {}
        for item in data_batch:
            key = (item.symbol, item.interval)
            if key not in grouped_data:
                grouped_data[key] = []
            grouped_data[key].append(item)

        # 📊 Process each group
        progress_bar = CustomProgressBar(
            total=len(grouped_data),
            desc="💾 Storing OHLC data"
        )

        for (symbol, interval), items in grouped_data.items():
            try:
                self._store_symbol_batch(symbol, interval, items)
                progress_bar.update(1)

            except Exception as e:
                logger.error(f"❌ Failed to store {symbol}_{interval}: {e}")
                print(f"⚠️  Storage error for {symbol}_{interval}: {e}")

        progress_bar.close()
        logger.info(f"✅ Stored {len(data_batch)} OHLC records")

    def _store_symbol_batch(self, symbol: str, interval: str, items: List[OHLCData]) -
        """
        Store batch of data for specific symbol+interval
        Uses Polars DataFrame for efficient bulk insert
        """
        # 🔄 Clean up existing connection if any
        await self.disconnect()

        # 📊 Store new configuration
        self.current_symbol = symbol
        self.current_interval = interval
        self.stream_url = self._build_stream_url(symbol, interval)

        try:
            # 🌐 Establish WebSocket connection
            self.websocket = await websockets.connect(
```

```
                self.stream_url,
                ping_interval=20,
                ping_timeout=10,
                close_timeout=10
            )

            self.is_connected = True
            self.reconnect_attempts = 0

            print(f"✅ Connected to {symbol} {interval} stream")
            logger.info(f"WebSocket connected: {self.stream_url}")

            # 🚀 Start data processing loop
            await self._process_data_stream()

            return True

        except Exception as e:
            self.error_handler.handle_websocket_error(e)
            print(f"❌ Connection failed: {e}")
            logger.error(f"WebSocket connection failed: {e}")
            return False

    async def disconnect(self):
        """
        🔌 Clean disconnection from WebSocket
        Ensures proper cleanup of resources
        """
        if self.websocket and not self.websocket.closed:
            print("🔌 Disconnecting WebSocket...")

            self.is_connected = False
            await self.websocket.close()

            print("✅ WebSocket disconnected cleanly")
            logger.info("WebSocket disconnected")

    async def _process_data_stream(self):
        """
        🔄 Main data processing loop
        Handles incoming WebSocket messages and processes them
        """
        print(f"🔄 Starting data stream processing for {self.current_symbol}...")

        try:
            async for message in self.websocket:
                if not self.is_connected:
                    break

                try:
                    # 📊 Parse incoming message
                    data = json.loads(message)
                    await self._handle_kline_data(data)

                except json.JSONDecodeError as e:
                    logger.warning(f"Invalid JSON received: {e}")
                    continue

                except Exception as e:
                    logger.error(f"Error processing message: {e}")
                    continue
```

```
            except websockets.exceptions.ConnectionClosed:
                print("🔌 WebSocket connection closed")
                logger.info("WebSocket connection closed")

                if self.should_reconnect:
                    await self._attempt_reconnection()

            except Exception as e:
                logger.error(f"Data stream error: {e}")
                print(f"❌ Stream error: {e}")

                if self.should_reconnect:
                    await self._attempt_reconnection()

    async def _handle_kline_data(self, data: Dict):
        """
        📈 Process kline (candlestick) data from Binance

        Binance sends kline data in this format:
        {
            "e": "kline",
            "E": 123456789,
            "s": "BNBBTC",
            "k": {
                "t": 123400000,
                "T": 123460000,
                "s": "BNBBTC",
                "i": "1m",
                "f": 100,
                "L": 200,
                "o": "0.0010",
                "c": "0.0020",
                "h": "0.0025",
                "l": "0.0015",
                "v": "1000",
                "n": 100,
                "x": false,
                "q": "1.0000",
                "V": "500",
                "Q": "0.500"
            }
        }
        """
        if data.get('e') != 'kline':
            return

        kline = data.get('k', {})

        # 📊 Extract OHLC data
        try:
            ohlc_data = OHLCData(
                timestamp=datetime.fromtimestamp(kline['t'] / 1000),
                symbol=kline['s'],
                interval=kline['i'],
                open=float(kline['o']),
                high=float(kline['h']),
                low=float(kline['l']),
                close=float(kline['c']),
                volume=float(kline['v'])
            )

            # 🔥 Only process completed candles for storage
```

```
            if kline.get('x', False):  # 'x' indicates kline is closed
                await self._process_completed_candle(ohlc_data)

            # 📡 Send real-time updates to callbacks (for live chart updates)
            for callback in self.data_callbacks:
                try:
                    await callback(ohlc_data.symbol, ohlc_data.interval, ohlc_data)
                except Exception as e:
                    logger.error(f"Callback error: {e}")

            # 📊 Print live price updates
            print(f"📊 {ohlc_data.symbol} {ohlc_data.interval} | "
                  f"O: {ohlc_data.open:.4f} H: {ohlc_data.high:.4f} "
                  f"L: {ohlc_data.low:.4f} C: {ohlc_data.close:.4f} "
                  f"V: {ohlc_data.volume:.0f}")

        except (KeyError, ValueError) as e:
            logger.error(f"Invalid kline data: {e}")

    async def _process_completed_candle(self, ohlc_data: OHLCData):
        """
        🔥 Process completed candle with indicators

        This runs when a candle is fully formed and ready for storage
        """
        print(f"🔥 Processing completed candle: {ohlc_data.symbol} {ohlc_data.timestam

        try:
            # 📈 Get recent historical data for indicator calculation
            recent_data = self.data_manager.get_historical_data(
                ohlc_data.symbol,
                ohlc_data.interval,
                limit=100
            )

            # 🔄 Add new candle and calculate indicators
            if len(recent_data) > 0:
                # Convert to list for appending
                new_row = {
                    'timestamp': ohlc_data.timestamp,
                    'symbol': ohlc_data.symbol,
                    'interval': ohlc_data.interval,
                    'open': ohlc_data.open,
                    'high': ohlc_data.high,
                    'low': ohlc_data.low,
                    'close': ohlc_data.close,
                    'volume': ohlc_data.volume
                }

                # 📊 Add new candle to recent data
                import polars as pl
                updated_data = recent_data.vstack(pl.DataFrame([new_row]))

                # 🧮 Calculate indicators
                data_with_indicators = self.indicator_manager.calculate_all_indicators

                # 📈 Get the latest row with indicators
                latest_row = data_with_indicators.tail(1).to_dicts()[0]

                # 🔄 Update OHLCData with indicator values
                ohlc_data.ema_12 = latest_row.get('ema_12')
                ohlc_data.ema_26 = latest_row.get('ema_26')
```

```
                ohlc_data.rsi = latest_row.get('rsi')
                ohlc_data.macd = latest_row.get('macd')
                ohlc_data.macd_signal = latest_row.get('macd_signal')
                ohlc_data.macd_histogram = latest_row.get('macd_histogram')
                ohlc_data.bb_upper = latest_row.get('bb_upper')
                ohlc_data.bb_middle = latest_row.get('bb_middle')
                ohlc_data.bb_lower = latest_row.get('bb_lower')
                ohlc_data.stoch_k = latest_row.get('stoch_k')
                ohlc_data.stoch_d = latest_row.get('stoch_d')

            # 💾 Store completed candle with indicators
            self.data_manager.store_ohlc_batch([ohlc_data])

            print(f"✅ Stored candle with indicators: {ohlc_data.symbol} {ohlc_data.ti

        except Exception as e:
            logger.error(f"Error processing completed candle: {e}")
            print(f"⚠️ Candle processing error: {e}")

    async def _attempt_reconnection(self):
        """
        🔄 Attempt to reconnect to WebSocket
        Implements exponential backoff strategy
        """
        if self.reconnect_attempts >= self.max_reconnect_attempts:
            print(f"❌ Max reconnection attempts reached ({self.max_reconnect_attempts
            logger.error("Max reconnection attempts exceeded")
            return

        self.reconnect_attempts += 1
        delay = min(self.reconnect_delay * (2 ** self.reconnect_attempts), 60)

        print(f"🔄 Reconnection attempt {self.reconnect_attempts}/{self.max_reconnect_
        logger.info(f"Attempting reconnection in {delay}s")

        await asyncio.sleep(delay)

        if self.current_symbol and self.current_interval:
            await self.connect(self.current_symbol, self.current_interval)

    async def switch_symbol_interval(self, symbol: str, interval: str):
        """
        🔄 Switch to different symbol/interval combination
        Cleanly tears down current connection and establishes new one
        """
        print(f"🔄 Switching to {symbol} {interval}...")

        # 🔌 Disconnect current stream
        await self.disconnect()

        # 🔗 Connect to new stream
        success = await self.connect(symbol, interval)

        if success:
            print(f"✅ Successfully switched to {symbol} {interval}")
        else:
            print(f"❌ Failed to switch to {symbol} {interval}")

        return success

    def stop(self):
        """
```

```
        🛑 Stop WebSocket manager
        Disables reconnection and closes connections
        """
        print("🛑 Stopping WebSocket manager...")

        self.should_reconnect = False
        self.is_connected = False

        if self.websocket and not self.websocket.closed:
            asyncio.create_task(self.websocket.close())

        logger.info("WebSocket manager stopped")

class StreamController:
    """
    🎛️ Stream Controller
    High-level interface for managing multiple WebSocket streams
    """

    def __init__(self, data_manager: DataManager):
        """Initialize stream controller"""
        self.data_manager = data_manager
        self.active_streams: Dict[str, BinanceWebSocketManager] = {}

        print("🎛️ Stream Controller initialized")

    async def start_stream(self, symbol: str, interval: str) -> BinanceWebSocketManage
        """
        🚀 Start a new WebSocket stream

        Returns the WebSocket manager for this stream
        """
        stream_key = f"{symbol}_{interval}"

        if stream_key in self.active_streams:
            print(f"⚠️ Stream already active: {stream_key}")
            return self.active_streams[stream_key]

        # 🌐 Create new WebSocket manager
        ws_manager = BinanceWebSocketManager(self.data_manager)

        # 🔗 Start connection
        success = await ws_manager.connect(symbol, interval)

        if success:
            self.active_streams[stream_key] = ws_manager
            print(f"✅ Stream started: {stream_key}")
        else:
            print(f"❌ Failed to start stream: {stream_key}")

        return ws_manager

    async def stop_stream(self, symbol: str, interval: str):
        """
        🛑 Stop a specific WebSocket stream
        """
        stream_key = f"{symbol}_{interval}"

        if stream_key in self.active_streams:
            ws_manager = self.active_streams[stream_key]
            ws_manager.stop()
            await ws_manager.disconnect()
```

```
            del self.active_streams[stream_key]
            print(f"🛑 Stream stopped: {stream_key}")
        else:
            print(f"⚠️ Stream not found: {stream_key}")

    async def stop_all_streams(self):
        """
        🛑 Stop all active WebSocket streams
        """
        print("🛑 Stopping all streams...")

        for stream_key, ws_manager in self.active_streams.items():
            ws_manager.stop()
            await ws_manager.disconnect()
            print(f"🛑 Stopped: {stream_key}")

        self.active_streams.clear()
        print("✅ All streams stopped")

    def get_active_streams(self) -> List[str]:
        """
        📊 Get list of active stream keys
        """
        return list(self.active_streams.keys())
```

🏗️ Phase 5: Chart Manager & Frontend Integration

5.1 Chart Manager Implementation

python

```python
# apps/trading_platform/chart_manager.py
"""
📊 Chart Manager for Lightweight Charts Integration
Handles multi-pane charts, indicators, and real-time updates
"""
import json
from datetime import datetime
from typing import Dict, List, Optional, Any
import polars as pl

from .data_manager import DataManager
from .utils.logger import get_logger

logger = get_logger(__name__)

class ChartManager:
    """
    📈 Advanced Chart Manager

    Features:
    - Multi-pane chart support
    - Real-time data updates
    - Indicator toggles and customization
    - Historical data loading
    - Time synchronization across panes
    """

    def __init__(self, data_manager: DataManager):
        """Initialize chart manager"""
        self.data_manager = data_manager

        # 📊 Chart configuration
        self.chart_config = {
            'main_pane': {
                'series': ['candlestick', 'ema_12', 'ema_26', 'bollinger_bands'],
                'visible': True
            },
            'rsi_pane': {
                'series': ['rsi'],
                'visible': True,
                'height': 150
            },
            'macd_pane': {
                'series': ['macd', 'macd_signal', 'macd_histogram'],
                'visible': True,
                'height': 150
            },
            'stoch_pane': {
                'series': ['stochastic'],
                'visible': False,
                'height': 150
            }
        }

        # 🎨 Indicator visibility toggles
        self.indicator_visibility = {
            'ema_12': True,
            'ema_26': True,
            'rsi': True,
            'macd': True,
            'bollinger_bands': True,
            'stochastic': False
```

```
        }

        print("📊 Chart Manager initialized")
        logger.info("ChartManager ready for charting operations")

    def get_chart_data(self, symbol: str, interval: str, limit: int = 300) -> Dict[str
        """
        📈 Get formatted chart data for frontend

        Returns data in lightweight-charts compatible format
        """
        print(f"📈 Preparing chart data for {symbol} {interval}...")

        # 📊 Get historical data
        df = self.data_manager.get_historical_data(symbol, interval, limit)

        if len(df) == 0:
            print(f"⚠️ No data available for {symbol} {interval}")
            return self._get_empty_chart_data()

        # 🔄 Convert to chart format
        chart_data = self._convert_to_chart_format(df)

        print(f"✅ Chart data prepared: {len(df)} candles")
        logger.info(f"Chart data prepared for {symbol}_{interval}: {len(df)} candles")

        return chart_data

    def _convert_to_chart_format(self, df: pl.DataFrame) -> Dict[str, Any]:
        """
        🔄 Convert Polars DataFrame to lightweight-charts format
        """
        # 📊 Main candlestick data
        candlestick_data = []
        ema_12_data = []
        ema_26_data = []
        rsi_data = []
        macd_data = []
        macd_signal_data = []
        macd_histogram_data = []
        bb_upper_data = []
        bb_middle_data = []
        bb_lower_data = []
        stoch_k_data = []
        stoch_d_data = []

        for row in df.to_dicts():
            timestamp = int(row['timestamp'].timestamp())

            # 🕯️ Candlestick data
            candlestick_data.append({
                'time': timestamp,
                'open': row['open'],
                'high': row['high'],
                'low': row['low'],
                'close': row['close']
            })

            # 📈 EMA data
            if row.get('ema_12') is not None:
                ema_12_data.append({
                    'time': timestamp,
```

```
            'value': row['ema_12']
        })

    if row.get('ema_26') is not None:
        ema_26_data.append({
            'time': timestamp,
            'value': row['ema_26']
        })

    # 📊 RSI data
    if row.get('rsi') is not None:
        rsi_data.append({
            'time': timestamp,
            'value': row['rsi']
        })

    # 📈 MACD data
    if row.get('macd') is not None:
        macd_data.append({
            'time': timestamp,
            'value': row['macd']
        })

    if row.get('macd_signal') is not None:
        macd_signal_data.append({
            'time': timestamp,
            'value': row['macd_signal']
        })

    if row.get('macd_histogram') is not None:
        macd_histogram_data.append({
            'time': timestamp,
            'value': row['macd_histogram']
        })

    # 📊 Bollinger Bands
    if row.get('bb_upper') is not None:
        bb_upper_data.append({
            'time': timestamp,
            'value': row['bb_upper']
        })

    if row.get('bb_middle') is not None:
        bb_middle_data.append({
            'time': timestamp,
            'value': row['bb_middle']
        })

    if row.get('bb_lower') is not None:
        bb_lower_data.append({
            'time': timestamp,
            'value': row['bb_lower']
        })

    # 📈 Stochastic
    if row.get('stoch_k') is not None:
        stoch_k_data.append({
            'time': timestamp,
            'value': row['stoch_k']
        })

    if row.get('stoch_d') is not None:
```

```
                stoch_d_data.append({
                    'time': timestamp,
                    'value': row['stoch_d']
                })

        return {
            'candlestick': candlestick_data,
            'ema_12': ema_12_data,
            'ema_26': ema_26_data,
            'rsi': rsi_data,
            'macd': macd_data,
            'macd_signal': macd_signal_data,
            'macd_histogram': macd_histogram_data,
            'bb_upper': bb_upper_data,
            'bb_middle': bb_middle_data,
            'bb_lower': bb_lower_data,
            'stoch_k': stoch_k_data,
            'stoch_d': stoch_d_data,
            'config': self.chart_config,
            'visibility': self.indicator_visibility
        }

    def _get_empty_chart_data(self) -> Dict[str, Any]:
        """
        📊 Return empty chart data structure
        """
        return {
            'candlestick': [],
            'ema_12': [],
            'ema_26': [],
            'rsi': [],
            'macd': [],
            'macd_signal': [],
            'macd_histogram': [],
            'bb_upper': [],
            'bb_middle': [],
            'bb_lower': [],
            'stoch_k': [],
            'stoch_d': [],
            'config': self.chart_config,
            'visibility': self.indicator_visibility
        }

    def update_indicator_visibility(self, indicator: str, visible: bool):
        """
        👁️ Toggle indicator visibility
        """
        if indicator in self.indicator_visibility:
            self.indicator_visibility[indicator] = visible
            print(f"👁️ {indicator} visibility: {'ON' if visible else 'OFF'}")
            logger.info(f"Indicator visibility updated: {indicator} -> {visible}")
        else:
            print(f"⚠️ Unknown indicator: {indicator}")

    def update_pane_visibility(self, pane: str, visible: bool):
        """
        📊 Toggle pane visibility
        """
        if pane in self.chart_config:
            self.chart_config[pane]['visible'] = visible
            print(f"📊 {pane} pane visibility: {'ON' if visible else 'OFF'}")
            logger.info(f"Pane visibility updated: {pane} -> {visible}")
```

```
            else:
                print(f"⚠️ Unknown pane: {pane}")

    def get_real_time_update(self, symbol: str, interval: str, ohlc_data) -> Dict[str,
        """
        🔄 Format real-time data update for frontend

        This is called by WebSocket callbacks to send live updates
        """
        timestamp = int(ohlc_data.timestamp.timestamp())

        update_data = {
            'candlestick': {
                'time': timestamp,
                'open': ohlc_data.open,
                'high': ohlc_data.high,
                'low': ohlc_data.low,
                'close': ohlc_data.close
            }
        }

        # 📈 Add indicator values if available
        if ohlc_data.ema_12 is not None:
            update_data['ema_12'] = {'time': timestamp, 'value': ohlc_data.ema_12}

        if ohlc_data.ema_26 is not None:
            update_data['ema_26'] = {'time': timestamp, 'value': ohlc_data.ema_26}

        if ohlc_data.rsi is not None:
            update_data['rsi'] = {'time': timestamp, 'value': ohlc_data.rsi}

        if ohlc_data.macd is not None:
            update_data['macd'] = {'time': timestamp, 'value': ohlc_data.macd}

        if ohlc_data.macd_signal is not None:
            update_data['macd_signal'] = {'time': timestamp, 'value': ohlc_data.macd_s

        if ohlc_data.macd_histogram is not None:
            update_data['macd_histogram'] = {'time': timestamp, 'value': ohlc_data.mac

        return update_data

class ReplayManager:
    """
    ⏯️ Replay Manager for Historical Data Playback
    Simulates live trading with historical data
    """

    def __init__(self, data_manager: DataManager):
        """Initialize replay manager"""
        self.data_manager = data_manager
        self.is_playing = False
        self.current_position = 0
        self.replay_data = []
        self.replay_speed = 1.0  # 1x speed

        print("⏯️ Replay Manager initialized")

    async def start_replay(
        self,
        symbol: str,
        interval: str,
```

```
        start_time: datetime,
        end_time: datetime,
        speed: float = 1.0
    ):
        """
        ▶️ Start historical data replay
        """
        print(f"▶️ Starting replay: {symbol} {interval} from {start_time} to {end_time

        # 📊 Load historical data for replay
        df = self.data_manager.get_historical_data(
            symbol, interval, limit=None
        ).filter(
            (pl.col('timestamp') >= start_time) &
            (pl.col('timestamp') <= end_time)
        )

        if len(df) == 0:
            print(f"⚠️ No data available for replay period")
            return

        self.replay_data = df.to_dicts()
        self.replay_speed = speed
        self.current_position = 0
        self.is_playing = True

        print(f"✅ Replay ready: {len(self.replay_data)} candles at {speed}x speed")

    def pause_replay(self):
        """⏸️ Pause replay"""
        self.is_playing = False
        print("⏸️ Replay paused")

    def resume_replay(self):
        """▶️ Resume replay"""
        self.is_playing = True
        print("▶️ Replay resumed")

    def stop_replay(self):
        """⏹️ Stop replay"""
        self.is_playing = False
        self.current_position = 0
        print("⏹️ Replay stopped")

    def set_speed(self, speed: float):
        """⚡ Set replay speed"""
        self.replay_speed = speed
        print(f"⚡ Replay speed set to {speed}x")

    def get_next_candle(self) -> Optional[Dict]:
        """
        ➡️ Get next candle in replay sequence
        """
        if not self.is_playing or self.current_position >= len(self.replay_data):
            return None

        candle = self.replay_data[self.current_position]
        self.current_position += 1

        return candle

    def get_replay_progress(self) -> Dict[str, Any]:
```

```
        """
        📊 Get current replay progress
        """
        total = len(self.replay_data)
        current = self.current_position

        return {
            'current': current,
            'total': total,
            'progress_percent': (current / total * 100) if total > 0 else 0,
            'is_playing': self.is_playing,
            'speed': self.replay_speed
        }
```

🏗️ Phase 6: Utility Classes & Error Handling

6.1 Custom Progress Bar Implementation

python

```python
# apps/trading_platform/utils/progress_bar.py
"""
🎨 Custom Progress Bar with Beautiful Styling
Enhanced tqdm with colors, animations, and custom styles
"""
import time
import random
from typing import Optional, List
from tqdm import tqdm
import colorama
from colorama import Fore, Style

# Initialize colorama for cross-platform color support
colorama.init()

class CustomProgressBar:
    """
    🌈 Beautiful Custom Progress Bar

    Features:
    - 5 different colors that change every 20%
    - Multiple bar styles (blocks, stars, pipes)
    - Inline progress display
    - Smooth animations
    - Custom descriptions
    """

    def __init__(
        self,
        total: int,
        desc: str = "Processing",
        bar_style: Optional[str] = None
    ):
        """
        Initialize custom progress bar

        Args:
            total: Total number of items to process
            desc: Description text
            bar_style: Custom bar style ('blocks', 'stars', 'pipes', 'random')
        """
        self.total = total
        self.desc = desc
        self.current = 0

        # 🎨 Color progression (changes every 20%)
        self.colors = [
            Fore.RED,      # 0-20%
            Fore.YELLOW,   # 20-40%
            Fore.BLUE,     # 40-60%
            Fore.MAGENTA,  # 60-80%
            Fore.GREEN     # 80-100%
        ]

        # 🎭 Bar styles
        self.bar_styles = {
            'blocks': '█',
            'stars': '★',
            'pipes': '|',
            'dots': '●',
            'arrows': '→'
        }
```

```
        # 🎲 Select bar style
        if bar_style == 'random':
            self.bar_char = random.choice(list(self.bar_styles.values()))
        elif bar_style in self.bar_styles:
            self.bar_char = self.bar_styles[bar_style]
        else:
            self.bar_char = random.choice(list(self.bar_styles.values()))

        # 📊 Initialize tqdm
        self.pbar = tqdm(
            total=total,
            desc=desc,
            bar_format='{l_bar}{bar}| {n_fmt}/{total_fmt} [{elapsed}<{remaining}, {rat
            ncols=100,
            dynamic_ncols=True,
            leave=True
        )

        print(f"🎨 Progress bar initialized: {desc} (style: {self.bar_char})")

    def update(self, n: int = 1):
        """
        🔄 Update progress bar

        Args:
            n: Number of items to add to progress
        """
        self.current += n

        # 🌈 Determine current color based on progress
        progress_percent = (self.current / self.total) * 100
        color_index = min(int(progress_percent // 20), 4)
        current_color = self.colors[color_index]

        # 🎨 Update with color
        self.pbar.set_postfix_str(f"{current_color}{progress_percent:.1f}%{Style.RESET
        self.pbar.update(n)

        # 🎉 Special message at milestones
        if progress_percent in [20, 40, 60, 80]:
            milestone_msg = f"🎯 {progress_percent:.0f}% Complete!"
            self.pbar.setConvert to Polars DataFrame
        df_data = []
        for item in items:
            df_data.append({
                'timestamp': item.timestamp,
                'symbol': item.symbol,
                'interval': item.interval,
                'open': item.open,
                'high': item.high,
                'low': item.low,
                'close': item.close,
                'volume': item.volume,
                'ema_12': item.ema_12,
                'ema_26': item.ema_26,
                'rsi': item.rsi,
                'macd': item.macd,
                'macd_signal': item.macd_signal,
                'macd_histogram': item.macd_histogram,
                'bb_upper': item.bb_upper,
                'bb_middle': item.bb_middle,
```

```
                'bb_lower': item.bb_lower,
                'stoch_k': item.stoch_k,
                'stoch_d': item.stoch_d,
            })

        df = pl.DataFrame(df_data)

        # 💫 Store using DuckDB-Polars integration
        conn = self._get_connection(symbol, interval)

        # Use UPSERT to handle duplicates gracefully
        conn.execute("""
            INSERT OR REPLACE INTO ohlc_data
            SELECT * FROM df
        """)

        print(f"✨ Stored {len(items)} records for {symbol}_{interval}")

    def get_historical_data(
        self,
        symbol: str,
        interval: str,
        limit: int = 300,
        start_time: Optional[datetime] = None,
        end_time: Optional[datetime] = None
    ) -> pl.DataFrame:
        """
        ⚡ Retrieve historical OHLC data with lightning speed

        Args:
            symbol: Trading symbol (e.g., 'BTCUSDT')
            interval: Time interval (e.g., '1m', '1h')
            limit: Maximum number of records
            start_time: Optional start timestamp
            end_time: Optional end timestamp

        Returns:
            Polars DataFrame with OHLC + indicator data
        """
        try:
            conn = self._get_connection(symbol, interval)

            # 🔍 Build query based on parameters
            query = "SELECT * FROM ohlc_data WHERE 1=1"
            params = []

            if start_time:
                query += " AND timestamp >= ?"
                params.append(start_time)

            if end_time:
                query += " AND timestamp <= ?"
                params.append(end_time)

            query += " ORDER BY timestamp DESC"

            if limit:
                query += f" LIMIT {limit}"

            # ⚡ Execute query and return as Polars DataFrame
            result = conn.execute(query, params).fetch_arrow_table()
            df = pl.from_arrow(result)
```

```
            logger.info(f"📈 Retrieved {len(df)} records for {symbol}_{interval}")
            print(f"📊 Loaded {len(df)} historical candles for {symbol}_{interval}")

            return df.sort('timestamp')

        except Exception as e:
            logger.error(f"❌ Failed to retrieve data for {symbol}_{interval}: {e}")
            print(f"⚠️  Query failed for {symbol}_{interval}: {e}")
            return pl.DataFrame()

    def get_latest_candle(self, symbol: str, interval: str) -> Optional[OHLCData]:
        """
        🔥 Get the most recent candle for symbol+interval
        Used for seamless live data continuation
        """
        try:
            conn = self._get_connection(symbol, interval)

            result = conn.execute("""
                SELECT * FROM ohlc_data
                ORDER BY timestamp DESC
                LIMIT 1
            """).fetchone()

            if result:
                return OHLCData(*result)

        except Exception as e:
            logger.error(f"❌ Failed to get latest candle for {symbol}_{interval}: {e}

        return None

    def cleanup_connections(self):
        """
        🔒 Clean up database connections
        Call this when shutting down the application
        """
        for key, conn in self._connections.items():
            conn.close()
            print(f"🔐 Closed connection: {key}")

        self._connections.clear()
        logger.info("✅ All database connections closed")
```

🏗️ Phase 3: Technical Indicators Engine

3.1 Indicator Calculations

Why This Implementation:

- Pure Python calculations for transparency and customization
- Vectorized operations using Polars for speed
- Modular design allows easy addition of new indicators
- Real-time calculation capability for live data

python

```python
# apps/trading_platform/indicators.py
"""
📊 Technical Indicators Engine
Implements popular trading indicators with high performance
All calculations are vectorized using Polars for speed
"""
import polars as pl
import numpy as np
from typing import Tuple, Optional
from .utils.logger import get_logger

logger = get_logger(__name__)

class TechnicalIndicators:
    """
    🔧 High-Performance Technical Indicators

    Features:
    - Vectorized calculations using Polars
    - Real-time indicator updates
    - Memory-efficient operations
    - Easy parameter customization
    """

    @staticmethod
    def calculate_ema(df: pl.DataFrame, column: str = 'close', period: int = 12) -> pl
        """
        📈 Exponential Moving Average

        EMA gives more weight to recent prices, making it more responsive
        Formula: EMA = (Close * α) + (Previous_EMA * (1 - α))
        where α = 2 / (period + 1)
        """
        print(f"🔄 Calculating EMA({period}) for {len(df)} candles...")

        closes = df[column].to_numpy()
        alpha = 2.0 / (period + 1)
        ema_values = np.zeros_like(closes)

        # Initialize first EMA as first close price
        ema_values[0] = closes[0]

        # Calculate EMA for each subsequent period
        for i in range(1, len(closes)):
            ema_values[i] = (closes[i] * alpha) + (ema_values[i-1] * (1 - alpha))

        logger.info(f"✅ EMA({period}) calculated successfully")
        return pl.Series(ema_values)

    @staticmethod
    def calculate_rsi(df: pl.DataFrame, column: str = 'close', period: int = 14) -> pl
        """
        📊 Relative Strength Index

        RSI measures the speed and magnitude of price changes
        Values range from 0-100, with 70+ indicating overbought, 30- oversold
        """
        print(f"🔄 Calculating RSI({period}) for {len(df)} candles...")

        closes = df[column].to_numpy()
        deltas = np.diff(closes)
```

```python
        # Separate gains and losses
        gains = np.where(deltas > 0, deltas, 0)
        losses = np.where(deltas < 0, -deltas, 0)

        # Calculate initial averages
        avg_gain = np.mean(gains[:period])
        avg_loss = np.mean(losses[:period])

        rsi_values = np.zeros(len(closes))
        rsi_values[:period] = np.nan

        # Calculate RSI for each period
        for i in range(period, len(closes)):
            if i == period:
                # First RSI calculation
                rs = avg_gain / avg_loss if avg_loss != 0 else 0
            else:
                # Smooth the averages (Wilder's smoothing)
                avg_gain = ((avg_gain * (period - 1)) + gains[i-1]) / period
                avg_loss = ((avg_loss * (period - 1)) + losses[i-1]) / period
                rs = avg_gain / avg_loss if avg_loss != 0 else 0

            rsi_values[i] = 100 - (100 / (1 + rs))

        logger.info(f"✅ RSI({period}) calculated successfully")
        return pl.Series(rsi_values)

    @staticmethod
    def calculate_macd(
        df: pl.DataFrame,
        column: str = 'close',
        fast_period: int = 12,
        slow_period: int = 26,
        signal_period: int = 9
    ) -> Tuple[pl.Series, pl.Series, pl.Series]:
        """
        📈 MACD (Moving Average Convergence Divergence)

        MACD shows the relationship between two EMAs
        Returns: (MACD Line, Signal Line, Histogram)
        """
        print(f"🔄 Calculating MACD({fast_period},{slow_period},{signal_period})...")

        # Calculate fast and slow EMAs
        fast_ema = TechnicalIndicators.calculate_ema(df, column, fast_period)
        slow_ema = TechnicalIndicators.calculate_ema(df, column, slow_period)

        # MACD line = Fast EMA - Slow EMA
        macd_line = fast_ema - slow_ema

        # Signal line = EMA of MACD line
        macd_df = pl.DataFrame({'macd': macd_line})
        signal_line = TechnicalIndicators.calculate_ema(macd_df, 'macd', signal_period

        # Histogram = MACD - Signal
        histogram = macd_line - signal_line

        logger.info("✅ MACD calculated successfully")
        return macd_line, signal_line, histogram

    @staticmethod
    def calculate_bollinger_bands(
```

```python
        df: pl.DataFrame,
        column: str = 'close',
        period: int = 20,
        std_dev: float = 2.0
    ) -> Tuple[pl.Series, pl.Series, pl.Series]:
        """
        📊 Bollinger Bands

        Bands that expand and contract based on market volatility
        Returns: (Upper Band, Middle Band/SMA, Lower Band)
        """
        print(f"🔄 Calculating Bollinger Bands({period}, {std_dev})...")

        # Calculate Simple Moving Average (Middle Band)
        sma = df[column].rolling_mean(window_size=period)

        # Calculate standard deviation
        std = df[column].rolling_std(window_size=period)

        # Calculate bands
        upper_band = sma + (std * std_dev)
        lower_band = sma - (std * std_dev)

        logger.info("✅ Bollinger Bands calculated successfully")
        return upper_band, sma, lower_band

    @staticmethod
    def calculate_stochastic(
        df: pl.DataFrame,
        high_col: str = 'high',
        low_col: str = 'low',
        close_col: str = 'close',
        k_period: int = 14,
        d_period: int = 3
    ) -> Tuple[pl.Series, pl.Series]:
        """
        📈 Stochastic Oscillator

        Compares closing price to price range over time
        Returns: (%K, %D)
        """
        print(f"🔄 Calculating Stochastic({k_period}, {d_period})...")

        # Calculate %K
        lowest_low = df[low_col].rolling_min(window_size=k_period)
        highest_high = df[high_col].rolling_max(window_size=k_period)

        k_percent = ((df[close_col] - lowest_low) / (highest_high - lowest_low)) * 100

        # Calculate %D (SMA of %K)
        d_percent = k_percent.rolling_mean(window_size=d_period)

        logger.info("✅ Stochastic calculated successfully")
        return k_percent, d_percent

class IndicatorManager:
    """
    🎛️ Indicator Management System
    Handles real-time indicator calculations and updates
    """

    def __init__(self):
```

```python
        """Initialize indicator manager with default parameters"""
        self.indicators = TechnicalIndicators()

        # 📊 Default indicator parameters (easily customizable)
        self.params = {
            'ema_12': {'period': 12},
            'ema_26': {'period': 26},
            'rsi': {'period': 14},
            'macd': {'fast': 12, 'slow': 26, 'signal': 9},
            'bollinger': {'period': 20, 'std_dev': 2.0},
            'stochastic': {'k_period': 14, 'd_period': 3}
        }

        print("🎛️ Indicator Manager initialized with default parameters")
        logger.info("IndicatorManager ready for calculations")

    def calculate_all_indicators(self, df: pl.DataFrame) -> pl.DataFrame:
        """
        🚀 Calculate all indicators for a DataFrame

        This is the main function that adds all indicator columns
        to your OHLC data for storage and charting
        """
        if len(df) < 50:  # Need sufficient data for indicators
            logger.warning(f"⚠️ Insufficient data for indicators: {len(df)} candles")
            return df

        print(f"🔧 Computing all indicators for {len(df)} candles...")
        result_df = df.clone()

        try:
            # 📈 EMAs
            result_df = result_df.with_columns([
                self.indicators.calculate_ema(df, 'close', 12).alias('ema_12'),
                self.indicators.calculate_ema(df, 'close', 26).alias('ema_26')
            ])

            # 📊 RSI
            result_df = result_df.with_columns([
                self.indicators.calculate_rsi(df, 'close', 14).alias('rsi')
            ])

            # 📈 MACD
            macd, signal, histogram = self.indicators.calculate_macd(df)
            result_df = result_df.with_columns([
                macd.alias('macd'),
                signal.alias('macd_signal'),
                histogram.alias('macd_histogram')
            ])

            # 📊 Bollinger Bands
            bb_upper, bb_middle, bb_lower = self.indicators.calculate_bollinger_bands(
            result_df = result_df.with_columns([
                bb_upper.alias('bb_upper'),
                bb_middle.alias('bb_middle'),
                bb_lower.alias('bb_lower')
            ])

            # 📈 Stochastic
            stoch_k, stoch_d = self.indicators.calculate_stochastic(df)
            result_df = result_df.with_columns([
                stoch_k.alias('stoch_k'),
```

```
                stoch_d.alias('stoch_d')
            ])

            print("✅ All indicators calculated successfully!")
            logger.info(f"✅ Indicators calculated for {len(df)} candles")

            return result_df

        except Exception as e:
            logger.error(f"❌ Indicator calculation failed: {e}")
            print(f"⚠️ Indicator calculation error: {e}")
            return df

    def update_parameters(self, indicator: str, **kwargs):
        """
        🎛️ Update indicator parameters

        Example:
        manager.update_parameters('rsi', period=21)
        manager.update_parameters('macd', fast=10, slow=21, signal=7)
        """
        if indicator in self.params:
            self.params[indicator].update(kwargs)
            print(f"🔧 Updated {indicator} parameters: {kwargs}")
            logger.info(f"Indicator parameters updated: {indicator} -> {kwargs}")
        else:
            print(f"⚠️ Unknown indicator: {indicator}")
```

🏗️ Phase 4: WebSocket Manager & Real-time Data

4.1 WebSocket Implementation

Why This Architecture:

- Clean separation between WebSocket handling and data processing
- Automatic reconnection and error recovery
- Efficient data streaming with minimal latency
- Easy symbol/interval switching without connection issues

python

```python
# apps/trading_platform/websocket_manager.py
"""
🌐 WebSocket Manager for Real-time Market Data
Handles Binance WebSocket streams with automatic reconnection
"""
import asyncio
import json
import websockets
from datetime import datetime
from typing import Dict, Callable, Optional, List
import logging

from .models import OHLCData
from .indicators import IndicatorManager
from .data_manager import DataManager
from .utils.logger import get_logger
from .utils.error_handler import ErrorHandler

logger = get_logger(__name__)

class BinanceWebSocketManager:
    """
    🚀 High-Performance WebSocket Manager

    Features:
    - Automatic reconnection on failures
    - Dynamic symbol/interval switching
    - Real-time indicator calculations
    - Efficient data batching and storage
    - Clean teardown and initialization
    """

    def __init__(self, data_manager: DataManager):
        """
        Initialize WebSocket manager

        Args:
            data_manager: DataManager instance for storing OHLC data
        """
        self.data_manager = data_manager
        self.indicator_manager = IndicatorManager()
        self.error_handler = ErrorHandler()

        # 🔗 Connection management
        self.websocket: Optional[websockets.WebSocketServerProtocol] = None
        self.is_connected = False
        self.should_reconnect = True

        # 📊 Current streaming configuration
        self.current_symbol = None
        self.current_interval = None
        self.stream_url = None

        # 📈 Data callbacks (for updating charts)
        self.data_callbacks: List[Callable] = []

        # 🔄 Reconnection settings
        self.reconnect_delay = 5  # seconds
        self.max_reconnect_attempts = 10
        self.reconnect_attempts = 0

        print("🌐 WebSocket Manager initialized")
```

```
        logger.info("BinanceWebSocketManager ready for connections")

    def add_data_callback(self, callback: Callable):
        """
        📡 Add callback function for real-time data updates

        Callbacks will be called with (symbol, interval, ohlc_data)
        Perfect for updating charts in real-time
        """
        self.data_callbacks.append(callback)
        print(f"📡 Added data callback: {callback.__name__}")

    def _build_stream_url(self, symbol: str, interval: str) -> str:
        """
        🔗 Build Binance WebSocket stream URL

        Format: wss://stream.binance.com:9443/ws/btcusdt@kline_1m
        """
        symbol_lower = symbol.lower()
        stream_name = f"{symbol_lower}@kline_{interval}"
        return f"wss://stream.binance.com:9443/ws/{stream_name}"

    async def connect(self, symbol: str, interval: str) -> bool:
        """
        🔌 Connect to Binance WebSocket stream

        Args:
            symbol: Trading symbol (e.g., 'BTCUSDT')
            interval: Time interval (e.g., '1m', '1h', '1d')

        Returns:
            True if connection successful, False otherwise
        """
        print(f"🔌 Connecting to {symbol} {interval} stream...")

        # 🔄
```