```
import asyncio
import ccxt.async_support as ccxt  # async support
import torch
import torch.nn as nn
import numpy as np
import random
import time
from datetime import datetime
import csv
from stable_baselines3 import PPO
from transformers import pipeline
import tweepy
import praw

# ============ CONFIG & SECURITY ============
EXCHANGES = ["binance", "bybit", "okx"]
SYMBOLS = ["BTC/USDT", "ETH/USDT"]
STARTING_CAPITAL = 5
TRADE_LOG = "trade_log.csv"
API_KEYS = {"binance": {FX2zGYkO4PRrixwYjkQ5lY14vmqmu0vKcjIYmjYvbLqP6MNxrmCpiZF28gOOLE42}, "bybit": {}, "okx": {FX2zGYkO4PRrixwYjkQ5lY14vmqmu0vKcjIYmjYvbLqP6MNxrmCpiZF28gOOLE42}}  # Insert your keys securely
RISK_THRESHOLD = 0.7
MAX_DRAWDOWN = -0.2
MAX_LEVERAGE = 5

TWITTER_KEYS = {"bearer_token": "YOUR_TWITTER_BEARER_TOKEN"}
REDDIT_KEYS = {
    "client_id": "YOUR_REDDIT_CLIENT_ID",
    "client_secret": "YOUR_REDDIT_CLIENT_SECRET",
    "user_agent": "trading_bot"
}

def secure_key_access(exchange):
    return API_KEYS[exchange].get('apiKey', ''), API_KEYS[exchange].get('secret', '')

# ============ AI MODELS ============
class RLTradingAgent:
    def __init__(self, model_path="ppo_trader.zip"):
        try:
            self.model = PPO.load(model_path)
            self.active = True
            print("[RL] PPO model loaded.")
```

```
        except Exception as e:
            print(f"[RL] Failed to load PPO model: {e}")
            self.model = None
            self.active = False

    def predict_action(self, obs):
        if not self.active:
            return random.choice([0, 1, 2])  # hold, buy, sell
        action, _ = self.model.predict(obs)
        return action

class SimpleMLModel(nn.Module):
    def __init__(self, input_dim=5):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 1),
            nn.Tanh()
        )
    def forward(self, x):
        return self.net(x)

    def load_weights(self, path):
        try:
            self.load_state_dict(torch.load(path))
            self.eval()
            print("[ML] Loaded model weights.")
        except Exception as e:
            print(f"[ML] Failed to load model weights: {e}")

# ============ SENTIMENT DATA ============
class NewsSentimentAnalyzer:
    def __init__(self, model_name="distilbert-base-uncased-finetuned-sentiment"):
        self.model = pipeline("sentiment-analysis", model=model_name)

    def get_sentiment(self, text):
        try:
            result = self.model(text)
            return result[0]['score'] * (1 if result[0]['label'] == 'POSITIVE' else -1)
        except Exception as e:
            print(f"[Sentiment] Error analyzing text: {e}")
            return 0
```

```
class SentimentAggregator:
    def __init__(self, twitter_keys, reddit_keys):
        self.twitter = tweepy.Client(bearer_token=twitter_keys["bearer_token"])
        self.reddit = praw.Reddit(
            client_id=reddit_keys["client_id"],
            client_secret=reddit_keys["client_secret"],
            user_agent=reddit_keys["user_agent"]
        )

    async def twitter_sentiment(self, query, analyzer):
        try:
            tweets = self.twitter.search_recent_tweets(query=query, max_results=10)
            sentiments = [analyzer.get_sentiment(tweet.text) for tweet in tweets.data if hasattr(tweet, "text")]
            return sum(sentiments) / len(sentiments) if sentiments else 0
        except Exception as e:
            print(f"[Sentiment] Twitter API error: {e}")
            return 0

    async def reddit_sentiment(self, subreddit, analyzer):
        try:
            posts = self.reddit.subreddit(subreddit).hot(limit=10)
            sentiments = [analyzer.get_sentiment(post.title + " " + getattr(post, "selftext", "")) for post in posts]
            return sum(sentiments) / len(sentiments) if sentiments else 0
        except Exception as e:
            print(f"[Sentiment] Reddit API error: {e}")
            return 0

    async def aggregate_sentiment(self, queries, analyzer):
        twitter_score = await self.twitter_sentiment(queries.get("twitter", "bitcoin"), analyzer)
        reddit_score = await self.reddit_sentiment(queries.get("reddit", "bitcoin"), analyzer)
        return (twitter_score + reddit_score) / 2 if (twitter_score or reddit_score) else 0

# ============ EXCHANGE MANAGER & MARKET DATA ============
class ExchangeManager:
    def __init__(self, names):
        self.rest = {}
        for ex in names:
            apiKey, secret = secure_key_access(ex)
            self.rest[ex] = getattr(ccxt, ex)({
                'apiKey': apiKey,
                'secret': secret,
                'enableRateLimit': True,
            })
        self.prices = {ex: {sym: [] for sym in SYMBOLS} for ex in names}
```

```
async def fetch_ws(self, ex, symbol, callback):
    """
    Placeholder for real websocket streaming.
    For production, integrate actual WS clients from exchanges.
    Here we fallback to REST polling every 0.5 sec.
    """
    print(f"[WS] Starting data stream for {ex} {symbol}")
    while True:
        try:
            ticker = await self.rest[ex].fetch_ticker(symbol)
            price = ticker['last']
        except Exception as e:
            print(f"[WS] Error fetching ticker {ex} {symbol}: {e}")
            price = random.uniform(25000, 35000)
        self.prices[ex][symbol].append(price)
        await callback(ex, symbol, price)
        await asyncio.sleep(0.5)

async def place_order(self, ex, symbol, side, amount, leverage=1, stealth=False):
    """
    Replace this with real order placement logic.
    """
    try:
        if stealth:
            # Split orders to mimic stealth
            chunks = random.randint(5, 15)
            split = amount / chunks
            for i in range(chunks):
                print(f"[STEALTH EXEC] {ex} {side} {split:.4f} {symbol} lev={leverage}")
                if random.random() < 0.2:
                    print(f"[DECOY] {ex} {'sell' if side == 'buy' else 'buy'} {split*0.2:.4f} {symbol} lev={leverage}")
        else:
            print(f"[EXEC] {ex} {side} {amount:.4f} {symbol} lev={leverage}")
        # Example async call (uncomment & adapt when real keys available):
        # order = await self.rest[ex].create_order(symbol, 'market', side, amount, params={'leverage': leverage})
        # print(f"[EXEC] Order placed: {order}")
    except Exception as e:
        print(f"[Order] Failed to place order {ex} {symbol}: {e}")

async def fetch_sentiment(self, ex, symbol):
    # Placeholder for real on-chain or order book sentiment
    return random.uniform(-1, 1)

async def fetch_spread(self, ex, symbol):
```

```
        # Placeholder for real spread fetching
        return 0.0002 + random.uniform(0, 0.0005)

    async def fetch_order_book(self, ex, symbol):
        # Placeholder bids and asks
        bids = [random.uniform(25000, 35000) for _ in range(10)]
        asks = [random.uniform(25000, 35000) for _ in range(10)]
        return {"bids": bids, "asks": asks}

    async def close(self):
        for ex in self.rest.values():
            await ex.close()
        print("[EXCHANGE] All connections closed.")

# ============ ARBITRAGE ENGINE ============
class ArbitrageEngine:
    def __init__(self, exchanges):
        self.exchanges = exchanges

    async def detect_opportunity(self, symbol):
        prices = {}
        for ex in self.exchanges.rest:
            try:
                ticker = await self.exchanges.rest[ex].fetch_ticker(symbol)
                prices[ex] = ticker['last']
            except Exception:
                prices[ex] = random.uniform(25000, 35000)
        best_buy = min(prices.items(), key=lambda x: x[1])
        best_sell = max(prices.items(), key=lambda x: x[1])
        spread = best_sell[1] - best_buy[1]
        if spread > 50:  # Arbitrage threshold
            return best_buy[0], best_sell[0], spread, best_buy[1], best_sell[1]
        return None

    async def execute_arbitrage(self, symbol, position_size, leverage):
        opp = await self.detect_opportunity(symbol)
        if opp:
            buy_ex, sell_ex, spread, buy_price, sell_price = opp
             print(f"[ARBITRAGE] Buy {symbol} on {buy_ex} at {buy_price:.2f}, sell on {sell_ex} at {sell_price:.2f},
spread: {spread:.2f}")
            await self.exchanges.place_order(buy_ex, symbol, "buy", position_size, leverage=leverage)
            await self.exchanges.place_order(sell_ex, symbol, "sell", position_size, leverage=leverage)
            return spread
        return 0
```

```
# ============ ML STRATEGY ============
class MLStrategy:
    def __init__(self):
        self.model = SimpleMLModel()
        # Load pre-trained weights here if available:
        # self.model.load_weights('ml_model_weights.pth')
        self.model.eval()

    def predict(self, features):
        with torch.no_grad():
            x = torch.tensor(features, dtype=torch.float32)
            out = self.model(x)
            return out.item()

# ============ RISK ENGINE ============
def calc_atr(prices, period=14):
    if len(prices) < period:
        return 0.001
    high = np.array(prices[-period:]) + 0.0005
    low = np.array(prices[-period:]) - 0.0005
    atr = np.mean(high - low)
    return atr

def risk_score(volatility, sentiment, spread, position_size, equity, market_depth):
    score = (abs(volatility) + abs(sentiment) + spread + (1.0 / (market_depth + 1))) * (position_size / max(equity, 1))
    return min(score, 1.0)

class RiskEngine:
    def __init__(self, threshold):
        self.threshold = threshold

    def check(self, risk_score_val):
        return risk_score_val <= self.threshold

# ============ SECURITY MONITOR ============
class SecurityMonitor:
    def __init__(self):
        self.threat = False

    def check_anomaly(self, events):
        if random.random() < 0.001:
            self.threat = True
            print("[SECURITY] Threat detected! Trading halted.")
        return not self.threat
```

```
# ============ PORTFOLIO MANAGER ============
class PortfolioManager:
    def __init__(self, capital, leverage=1):
        self.capital = capital
        self.equity = capital
        self.position_size = max(0.01, capital * 0.05)
        self.max_drawdown = MAX_DRAWDOWN
        self.leverage = leverage
        self.equity_peak = capital

    def update(self, pnl):
        self.equity += pnl
        self.equity_peak = max(self.equity_peak, self.equity)
        self.position_size = max(0.01, self.equity * 0.05)

    def scale_trade(self, volatility):
        if volatility > 0.02:
            self.leverage = max(1, self.leverage - 1)
        else:
            self.leverage = min(MAX_LEVERAGE, self.leverage + 1)
        return self.position_size, self.leverage

    def drawdown_exceeded(self):
        drawdown = (self.equity - self.equity_peak) / self.equity_peak
        return drawdown < self.max_drawdown

# ============ TRADE LOGGING ============
def log_trade(trade):
    try:
        with open(TRADE_LOG, "a", newline="") as f:
            writer = csv.writer(f)
            writer.writerow([
                datetime.utcnow().isoformat(),
                trade["exchange"],
                trade["symbol"],
                trade["side"],
                trade["size"],
                trade["price"],
                trade["signal"],
                trade["risk_score"],
                trade["leverage"],
                trade["stealth"]
            ])
    except Exception as e:
        print(f"[Log] Failed to write trade log: {e}")
```

```
# ============ SLIPPAGE SIMULATION ============
def simulate_slippage(price, spread=0.0002):
    slip = price * spread * np.random.uniform(0.8, 1.2)
    return price + slip if np.random.rand() < 0.5 else price - slip

# ============ MAIN BOT ============
class UltimateDisruptiveBot:
    def __init__(self):
        self.exchanges = ExchangeManager(EXCHANGES)
        self.strategy = MLStrategy()
        self.rl_agent = RLTradingAgent()
        self.sentiment_analyzer = NewsSentimentAnalyzer()
        self.sentiment_aggregator = SentimentAggregator(TWITTER_KEYS, REDDIT_KEYS)
        self.risk = RiskEngine(RISK_THRESHOLD)
        self.portfolio = PortfolioManager(STARTING_CAPITAL)
        self.security = SecurityMonitor()
        self.arbitrage = ArbitrageEngine(self.exchanges)
        self.trade_history = []
        self.active = True

    async def handle_tick(self, ex, symbol, price):
        if not self.active:
            return

        prices = self.exchanges.prices[ex][symbol]
        atr = calc_atr(prices) if len(prices) > 14 else 0.001
        sentiment = await self.exchanges.fetch_sentiment(ex, symbol)
        spread = await self.exchanges.fetch_spread(ex, symbol)
        order_book = await self.exchanges.fetch_order_book(ex, symbol)
        market_depth = np.mean(order_book["bids"] + order_book["asks"])
        features = [price, atr, sentiment, spread, market_depth]

        ml_signal = self.strategy.predict(features)
        rl_obs = np.array(features)
        rl_action = self.rl_agent.predict_action(rl_obs)

        social_sentiment = await self.sentiment_aggregator.aggregate_sentiment(
            {"twitter": symbol.split("/")[0], "reddit": symbol.split("/")[0].lower()},
            self.sentiment_analyzer
        )

        signal = (ml_signal + (rl_action - 1) + social_sentiment) / 3

        position_size, leverage = self.portfolio.scale_trade(atr)
```

```
        risk_val = risk_score(atr, sentiment + social_sentiment, spread, position_size, self.portfolio.equity, market_depth)

    arb_spread = await self.arbitrage.execute_arbitrage(symbol, position_size, leverage)
    if arb_spread:
        pnl = arb_spread * position_size
        self.portfolio.update(pnl)
        self.trade_history.append(pnl)
        return

    if not self.risk.check(risk_val):
        print(f"[RISK] Trade too risky: {risk_val:.2f}")
        return

    stealth = random.random() < 0.5
    trade_side = "buy" if signal > 0.5 else "sell" if signal < -0.5 else "hold"
    if trade_side != "hold" and self.security.check_anomaly({"exchange": ex, "symbol": symbol}):
        exec_price = simulate_slippage(price, spread)
        await self.exchanges.place_order(ex, symbol, trade_side, position_size, leverage=leverage, stealth=
```