```
import os
import requests
import redis
import tweepy
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
import MetaTrader5 as mt5
import pandas as pd
import numpy as np
import talib
import hmmlearn.hmm
import tensorflow as tf
from tensorflow.keras import layers
import dash
import dash_core_components as dcc
import dash_html_components as html
from telebot import TeleBot
import time
from datetime import datetime
import logging

# --- Genel Ayarlar ve Başlatma ---

if not mt5.initialize():
    print("MT5 başlatılamadı, lütfen bağlantıyı kontrol edin.")
    exit()

# API anahtarları ve token'lar (kendi değerlerinizle değiştirin)
FRED_API_KEY = "YOUR_FRED_API_KEY"
TWITTER_BEARER_TOKEN = "YOUR_TWITTER_BEARER_TOKEN"
TELEGRAM_BOT_TOKEN = "YOUR_TELEGRAM_BOT_TOKEN"

# Redis önbellekleme
cache = redis.Redis(host='localhost', port=6379, db=0)

# Günlükleme
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s
- %(message)s')

# --- Veri Entegrasyonu ---

class MacroDataFetcher:
    """FRED API'den makroekonomik verileri çeker."""
    def __init__(self, api_key):
        self.api_url = "https://api.stlouisfed.org/fred/series/observations"
        self.api_key = api_key
```

```
    def fetch_inflation(self):
        """Son CPI enflasyon verisini çeker."""
        try:
            params = {"series_id": "CPIAUCSL", "api_key": self.api_key, "file_type":
"json"}
            response = requests.get(self.api_url, params=params)
            response.raise_for_status()
            data = response.json()
            inflation = float(data["observations"][-1]["value"])
            cache.set('inflation', inflation, ex=3600)
            return inflation
        except Exception as e:
            logging.error(f"Enflasyon verisi çekme hatası: {e}")
            cached = cache.get('inflation')
            return float(cached) if cached else 0

class FuturesDataLoader:
    """Binance Futures API'den vadeli işlem verilerini çeker."""
    def __init__(self):
        self.api_url = "https://fapi.binance.com/fapi/v1/openInterest"

    def fetch_open_interest(self, symbol="BTCUSDT"):
        """Vadeli kontratlar için açık faizi çeker."""
        try:
            response = requests.get(self.api_url, params={"symbol": symbol})
            response.raise_for_status()
            data = response.json()
            open_interest = float(data["openInterest"])
            cache.set(f'open_interest_{symbol}', open_interest, ex=3600)
            return open_interest
        except Exception as e:
            logging.error(f"Açık faiz çekme hatası: {e}")
            cached = cache.get(f'open_interest_{symbol}')
            return float(cached) if cached else 0

class MultiLingualTwitterSentiment:
    """Twitter'dan çok dilli duygu analizi yapar."""
    def __init__(self, bearer_token):
        self.client = tweepy.Client(bearer_token=bearer_token)
        self.analyzer = SentimentIntensityAnalyzer()
        self.hashtags = ["#USD", "#Fed", "#forex", "#EURUSD"]
        self.langs = ["en", "tr", "fr", "de"]

    def fetch_and_analyze(self):
        """Son tweet'lerden duygu analizi yapar."""
        scores = []
```

```
        for tag in self.hashtags:
            for lang in self.langs:
                q = f"{tag} lang:{lang} -is:retweet"
                try:
                    res = self.client.search_recent_tweets(query=q, max_results=50)
                    if res and res.data:
                        for t in res.data:
                            vs = self.analyzer.polarity_scores(t.text)
                            scores.append(vs["compound"])
                except Exception as e:
                    logging.error(f"Twitter analizi hatası: {e}")
                    continue
        sentiment = sum(scores) / len(scores) if scores else 0.0
        cache.set('twitter_sentiment', sentiment, ex=3600)
        return sentiment

# --- Özellik Mühendisliği ---

class EconomicCalendar:
    """Forex Factory'den ekonomik takvim verilerini çeker."""
    def fetch_high_impact(self):
        """Yüksek etkili olayları çeker."""
        try:
            # Gerçek bir API ile entegre edilmeli; burada simüle edilmiş veri
            events = [{"time": "2023-10-01 14:30", "event": "Non-Farm Payrolls"}]
            cache.set('high_impact_events', str(events), ex=3600)
            return events
        except Exception as e:
            logging.error(f"Ekonomik takvim hatası: {e}")
            cached = cache.get('high_impact_events')
            return eval(cached) if cached else []

class COTLoader:
    """CFTC'den Commitment of Traders (COT) verilerini çeker."""
    def __init__(self):
        self.url = "https://www.cftc.gov/sites/default/files/files/dea/history/
fut_fin_txt_2023.zip"

    def fetch(self):
        """COT verilerini indirir ve işler."""
        try:
            # Gerçek uygulamada ZIP indirilip işlenmeli; burada simüle edilmiş veri
            cot_data = pd.DataFrame({"net_position": [1000, 1200, 1100]},
index=[0, 1, 2])
            cache.set('cot_data', cot_data.to_json(), ex=3600)
            return cot_data
        except Exception as e:
```

```
            logging.error(f"COT veri çekme hatası: {e}")
            cached = cache.get('cot_data')
            return pd.read_json(cached) if cached else pd.DataFrame()

class RegimeClassifier:
    """Pazar rejimlerini Gizli Markov Modeli ile sınıflandırır."""
    def __init__(self):
        self.hmm = hmmlearn.hmm.GaussianHMM(n_components=3,
covariance_type="full")

    def fit(self, data):
        """HMM modelini eğitir."""
        self.hmm.fit(data)

    def predict(self, data):
        """Rejimleri tahmin eder."""
        return self.hmm.predict(data)

class FeatureEngineer:
    """Trading botu için özellikler oluşturur."""
    def get_bars(self, symbol, timeframe, count):
        """MT5'ten tarihsel barları çeker."""
        rates = mt5.copy_rates_from_pos(symbol, timeframe, 0, count)
        df = pd.DataFrame(rates) if rates is not None else pd.DataFrame()
        if not df.empty:
            df['time'] = pd.to_datetime(df['time'], unit='s')
        return df

    def validate_data(self, df):
        """Veriyi doğrulayarak aykırı değerleri ve sıfır hacimli barları filtreler."""
        if df.empty:
            return df
        atr = talib.ATR(df['high'], df['low'], df['close'], timeperiod=14)
        price_jumps = df['close'].diff().abs() > 5 * atr
        zero_volume = df['volume'] == 0
        return df[~(price_jumps | zero_volume)]

    def compute_multitimeframe_features(self, symbol="EURUSD"):
        """Çoklu zaman dilimlerinde özellikler hesaplar."""
        timeframes = {
            'M1': mt5.TIMEFRAME_M1,
            'H1': mt5.TIMEFRAME_H1,
            'H4': mt5.TIMEFRAME_H4,
            'D1': mt5.TIMEFRAME_D1,
            'W1': mt5.TIMEFRAME_W1
        }
        features = {}
```

```
        for tf_name, tf in timeframes.items():
            df = self.get_bars(symbol, tf, 500)
            if not df.empty:
                features[f'{tf_name}_RSI'] = talib.RSI(df['close'],
timeperiod=14).iloc[-1]
                features[f'{tf_name}_MACD'] = talib.MACD(df['close'], fastperiod=12,
slowperiod=26, signalperiod=9)[2].iloc[-1]
                if tf_name in ['D1', 'W1']:
                    features[f'{tf_name}_ADX'] = talib.ADX(df['high'], df['low'],
df['close'], timeperiod=14).iloc[-1]
                    features[f'{tf_name}_Ichimoku'] = talib.ICHIMOKU(df['high'],
df['low'], df['close'])[0].iloc[-1]
        return features

    def compute_features(self, symbol="EURUSD"):
        """Harici verilerle birlikte tüm özellikleri birleştirir."""
        mtf_features = self.compute_multitimeframe_features(symbol)
        ec = EconomicCalendar()
        cot = COTLoader()
        high_impact_events = ec.fetch_high_impact()
        cot_data = cot.fetch()

        event_impact_score = len(high_impact_events) * 0.5 if
high_impact_events else 0
        net_position_ratio = cot_data["net_position"].iloc[-1] if not
cot_data.empty else 0

        macro_fetcher = MacroDataFetcher(FRED_API_KEY)
        inflation = macro_fetcher.fetch_inflation()

        twitter_sentiment =
MultiLingualTwitterSentiment(TWITTER_BEARER_TOKEN).fetch_and_analyze()

        return {
            **mtf_features,
            "event_impact_score": event_impact_score,
            "net_position_ratio": net_position_ratio,
            "inflation": inflation,
            "twitter_sentiment": twitter_sentiment
        }

# --- Ödül Fonksiyonu ve Geri Test ---

class Position:
    def __init__(self, entry_price, direction, lot, entry_time):
        self.entry_price = entry_price
        self.direction = direction  # 1: long, -1: short
```

```
        self.lot = lot
        self.entry_time = entry_time

    def close(self, exit_price, exit_time):
        holding_time = (exit_time - self.entry_time).total_seconds() / 3600  # saat
cinsinden
        return holding_time

class Backtester:
    def __init__(self, symbol, timeframe, data_dir, start_date, end_date):
        self.symbol = symbol
        self.timeframe = timeframe
        self.data_dir = data_dir
        self.start_date = pd.to_datetime(start_date)
        self.end_date = pd.to_datetime(end_date)
        self.position = None

    def load_historical(self):
        """Tarihsel verileri parçalar halinde yükler."""
        fname = f"{self.symbol}_{self.timeframe}.csv"
        path = os.path.join(self.data_dir, fname)
        if not os.path.exists(path):
            logging.error(f"Veri dosyası bulunamadı: {path}")
            return
        for chunk in pd.read_csv(path, chunksize=100ik00):
            chunk['time'] = pd.to_datetime(chunk['time'])
            yield chunk[(chunk['time'] >= self.start_date) & (chunk['time'] <=
self.end_date)]

    def compute_atr(self, df):
        return talib.ATR(df['high'], df['low'], df['close'], timeperiod=14).iloc[-1]

    def compute_sharpe_sortino(self, pnl, df):
        atr = self.compute_atr(df)
        sharpe = pnl / atr if atr > 0 else 0
        returns = pd.Series([pnl])
        downside_dev = returns[returns < 0].std()
        sortino = pnl / downside_dev if downside_dev > 0 else 0
        return sharpe, sortino

    def close_reward(self, exit_price, pip_value, df, holding_time,
correct_direction):
        if not self.position:
            return 0
        pnl = (exit_price - self.position.entry_price) * self.position.direction *
self.position.lot * pip_value
        sharpe, sortino = self.compute_sharpe_sortino(pnl, df)
```

```
        reward = sharpe + sortino
        if holding_time > 4:
            reward -= 0.1  # Swap maliyeti
        if correct_direction:
            reward += 0.5  # Doğru yön bonusu
        return reward

    def run_backtest(self):
        try:
            df = next(self.load_historical())
            if df.empty:
                logging.error("Geri test için veri yok.")
                return
        except StopIteration:
            logging.error("Veri yüklenemedi.")
            return

        regime_classifier = RegimeClassifier()
        regime_classifier.fit(df[['close']].values)
        regimes = regime_classifier.predict(df[['close']].values)

        for idx in range(500, len(df)-1):
            entry_time = df['time'].iloc[idx-1]
            self.position = Position(df['close'].iloc[idx-1], 1, 0.1, entry_time)
            exit_price = df['close'].iloc[idx]
            exit_time = df['time'].iloc[idx]
            holding_time = self.position.close(exit_price, exit_time)
            correct_direction = (exit_price > self.position.entry_price) if
self.position.direction == 1 else (exit_price < self.position.entry_price)
            reward = self.close_reward(exit_price, 10000, df, holding_time,
correct_direction)
            if regimes[idx] == 0:  # Trend rejimi
                reward += 0.5
            elif regimes[idx] == 1:  # Konsolidasyon rejimi
                reward -= 0.2
            logging.info(f"Adım {idx}: Ödül = {reward}")

# --- RL Algoritması ---

class NoisyDense(layers.Layer):
    def __init__(self, units):
        super(NoisyDense, self).__init__()
        self.units = units

    def build(self, input_shape):
        self.w_mu = self.add_weight(shape=(input_shape[-1], self.units),
initializer='random_normal')
```

```
        self.w_sigma = self.add_weight(shape=(input_shape[-1], self.units),
initializer='random_normal')
        self.b_mu = self.add_weight(shape=(self.units,), initializer='zeros')
        self.b_sigma = self.add_weight(shape=(self.units,), initializer='zeros')

    def call(self, inputs, training=False):
        if training:
            w_noise = tf.random.normal(self.w_mu.shape)
            b_noise = tf.random.normal(self.b_mu.shape)
            w = self.w_mu + self.w_sigma * w_noise
            b = self.b_mu + self.b_sigma * b_noise
        else:
            w = self.w_mu
            b = self.b_mu
        return tf.matmul(inputs, w) + b

class DuelingRainbowDQN(tf.keras.Model):
    def __init__(self, state_size, action_size):
        super(DuelingRainbowDQN, self).__init__()
        self.lstm = layers.LSTM(128, return_sequences=False)
        self.fc1 = NoisyDense(128)
        self.value_stream = layers.Dense(1)
        self.advantage_stream = layers.Dense(action_size)

    def call(self, x, training=False):
        x = tf.expand_dims(x, axis=1)
        x = self.lstm(x)
        x = tf.nn.relu(self.fc1(x, training=training))
        value = self.value_stream(x)
        advantage = self.advantage_stream(x)
        q_values = value + (advantage - tf.reduce_mean(advantage, axis=1,
keepdims=True))
        return q_values

class RainbowAgent:
    def __init__(self, state_size, action_size):
        self.model = DuelingRainbowDQN(state_size, action_size)
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)
        self.model.compile(optimizer=self.optimizer, loss='mse')

    def load_pretrained_model(self, model_path):
        if os.path.exists(model_path):
            self.model.load_weights(model_path)
            logging.info(f"Model yüklendi: {model_path}")
        else:
            logging.warning("Önceden eğitilmiş model bulunamadı.")
```

```
    def fine_tune(self, states, actions, rewards, next_states, dones,
batch_size=32):
        states = np.array(states)
        next_states = np.array(next_states)
        targets = self.model.predict(states)
        next_q_values = self.model.predict(next_states)
        for i in range(len(states)):
            if dones[i]:
                targets[i, actions[i]] = rewards[i]
            else:
                targets[i, actions[i]] = rewards[i] + 0.99 * np.max(next_q_values[i])
        self.model.fit(states, targets, batch_size=batch_size, epochs=1,
verbose=0)

# --- İşlem Yürütme ---

class OrderExecutor:
    def open_long(self, symbol, lot, atr, deviation=20, magic=123456,
comment="Bot Long"):
        tick = mt5.symbol_info_tick(symbol)
        if not tick:
            logging.error(f"{symbol} için tick verisi alınamadı.")
            return None
        price = tick.ask
        sl = price - 1.5 × atr
        tp = price + 3 × atr
        request = {
            "action": mt5.TRADE_ACTION_DEAL,
            "symbol": symbol,
            "volume": lot,
            "type": mt5.ORDER_TYPE_BUY,
            "price": price,
            "sl": sl,
            "tp": tp,
            "deviation": deviation,
            "magic": magic,
            "comment": comment,
            "type_time": mt5.ORDER_TIME_GTC,
            "type_filling": mt5.ORDER_FILLING_IOC,
        }
        result = mt5.order_send(request)
        if result.retcode != mt5.TRADE_RETCODE_DONE:
            logging.error(f"İşlem başarısız: {result.comment}")
        return result

# --- Risk Yönetimi ---
```

```
class RiskManager:
    def calculate_lot_size(self, risk_per_trade, atr, pip_value):
        lot = risk_per_trade / (atr / pip_value)
        return max(0.01, min(lot, 10.0))

# --- Canlı Alım Satım ---

class LiveTrading:
    def __init__(self):
        self.mt5 = mt5
        self.reconnect()

    def reconnect(self):
        for attempt in range(3):
            if self.mt5.initialize():
                logging.info("MT5 bağlantısı başarılı.")
                return
            logging.warning(f"MT5 bağlantı hatası, tekrar deneniyor ({attempt+1}/
3)...")
            time.sleep(5)
        logging.error("MT5 bağlantısı sağlanamadı.")
        exit()

    def run(self, symbol, agent, feature_engineer, executor, risk_manager):
        while True:
            features = feature_engineer.compute_features(symbol)
            state = np.array(list(features.values()))
            action = np.argmax(agent.model.predict(np.array([state]))[0])
            df = feature_engineer.get_bars(symbol, mt5.TIMEFRAME_M15, 100)
            atr = talib.ATR(df['high'], df['low'], df['close'], timeperiod=14).iloc[-1]
            lot = risk_manager.calculate_lot_size(0.01, atr, 10000)
            if action == 1:  # Long
                executor.open_long(symbol, lot, atr)
            time.sleep(60)

# --- Gösterge Paneli ---

app = dash.Dash(__name__)

app.layout = html.Div([
    html.H1("FTMO Trading Bot Dashboard"),
    dcc.Graph(id='equity-curve'),
    dcc.Slider(id='risk-slider', min=0, max=5, step=0.1, value=1),
    dcc.Interval(id='interval-component', interval=60*1000, n_intervals=0)
])

# --- Telegram Botu ---
```

```
bot = TeleBot(TELEGRAM_BOT_TOKEN)

@bot.message_handler(commands=['set_lot'])
def set_lot(message):
    try:
        lot = float(message.text.split()[1])
        bot.reply_to(message, f"Lot büyüklüğü {lot} olarak ayarlandı.")
    except Exception as e:
        bot.reply_to(message, f"Hata: {e}")

@bot.message_handler(commands=['start'])
def start_bot(message):
    bot.reply_to(message, "Trading botu başlatıldı!")

# --- Ana Fonksiyon ---

def main():
    live_trading = LiveTrading()
    feature_engineer = FeatureEngineer()
    backtester = Backtester("EURUSD", "M15", "data", "2003-01-01",
"2023-12-31")  # 20 yıllık veri
    executor = OrderExecutor()
    risk_manager = RiskManager()
    agent = RainbowAgent(state_size=10, action_size=3)

    print("Özellikler hesaplanıyor...")
    features = feature_engineer.compute_features("EURUSD")
    print("Özellikler:", features)

    print("Geri test başlatılıyor...")
    backtester.run_backtest()

    print("Canlı alım satım başlatılıyor...")
    live_trading.run("EURUSD", agent, feature_engineer, executor, risk_manager)

if __name__ == "__main__":
    import threading
    threading.Thread(target=bot.polling, daemon=True).start()
    threading.Thread(target=lambda: app.run_server(debug=False, port=8050),
daemon=True).start()
    main()
```