main.py

```
# library imports
import os
import asyncio
import aiofiles
import aiocsv
import threading
import pandas as pd
import csv
import matplotlib.pyplot as plt
import json
import time
import sys
import websockets

# local imports
from classes.interface_classes import Interface, DataStore, SymbolsManagerBase

# Huobi imports
import huobi_interface as huobi_interface
import api_keys as api_keys
from huobi.model.market import *
from huobi.constant import *

# Kucoin imports
import kucoin_interface

# Binance imports
import binance_interface

# ---------------------------- CONSTANTS ----------------------------
#EXCLUDED_COINS = ["btcusdt","ethusdt"]
EXCLUDED_COINS = []
INTERVAL = 20 # in seconds, how often to collect data
KLINE_INTERVAL = "1hour" # interval for kline data
KLINE_INTERVAL_SECONDS = 3600 # interval for kline data in seconds
DURATION = 999999 # in seconds, how long to collect data for
SIMULTANEOUS_REQUESTS = 5 # number of requests to make at once - prevents rate limiting
SLEEP_BETWEEN_THREAD_GEN = 10 # in seconds, how long to wait between generating threads
SLEEP_BETWEEN_HISTORY_REQUESTS = 1 # in seconds, how long to wait between history requests
DELAY_AFTER_TIMEOUT = 20 # in seconds, how long to wait after a timeout

# ---------------------------- CLASSES ----------------------------
# Factory classes for exchanges
class APIFactory:
    """
    Returns API instance for input exchange
    """
    def __init__(self, exchange: str):
        self.exchange = exchange

    def get_api(self):
        if self.exchange == "huobi":
            return huobi_interface.HuobiAPI(api_keys.hb_api_key, api_keys.hb_secret_key)
        elif self.exchange == "kucoin":
            return kucoin_interface.KucoinAPI(api_keys.kc_api_key, api_keys.kc_secret_key)
```

```
        elif self.exchange == "binance":
            return binance_interface.BinanceAPI("", "") # no binance api keys
        else:
            raise Exception("Exchange not supported")

class SymbolManagerFactory:
    """
    Returns SymbolManager instance for input exchange
    """
    def __init__(self, exchange: str, interface: Interface):
        self.exchange = exchange
        self.interface = interface

    def get_symbol_manager(self):
        if self.exchange == "huobi":
            return huobi_interface.HuobiSymbolsManager(self.interface)
        elif self.exchange == "kucoin":
            return kucoin_interface.KucoinSymbolsManager(self.interface)
        elif self.exchange == "binance":
            return binance_interface.BinanceSymbolsManager(self.interface)
        else:
            raise Exception("Exchange not supported")

class KlineIntervals:
    """
    Defines kline intervals for each exchange
    Gets either list of intervals or dictionary of seconds to interval
    """
    def __init__(self, exchange: str):
        self.exchange = exchange
        self.intervals = self._set_intervals()

    def _set_intervals(self):
        kline_intervals = {}
        if self.exchange == "binance":
            kline_intervals = {
                "1m": 60,
                "3m": 180,
                "5m": 300,
                "15m": 900,
                "30m": 1800,
                "1h": 3600,
                "2h": 7200,
                "1d": 86400,
                "1w": 604800,
                "1M": 2592000
            }
        elif self.exchange == "kucoin":
            kline_intervals = {
                "1min": 60,
                "3min": 180,
                "5min": 300,
                "15min": 900,
                "30min": 1800,
                "1hour": 3600,
                "2hour": 7200,
                "1d": 86400,
                "1week": 604800,
                "1mon": 2592000
```

```
            }
        elif self.exchange == "huobi":
            kline_intervals = {
                "1min": 60,
                "5min": 300,
                "15min": 900,
                "30min": 1800,
                "60min": 3600,
                "1day": 86400,
                "1week": 604800,
                "1mon": 2592000
            }
        else:
            raise Exception("Exchange not supported")
        return kline_intervals

    def get_intervals(self):
        return list(self.intervals.keys())

    def get_interval_seconds(self, interval: str):
        return self.intervals[interval]

class ErrorCodes():
    """
    Defines error codes for each exchange
    """
    def __init__(self, exchange: str):
        self.exchange = exchange
        self.error_codes = self._set_error_codes()

    def _set_error_codes(self):
        error_codes = {}
        if self.exchange == "binance":
            error_codes = {
                "TOO_MANY_REQUESTS": -1003,
                "TIMEOUT": -1001
            }
        elif self.exchange == "kucoin":
            error_codes = {
                "TOO_MANY_REQUESTS": 429,
                "TIMEOUT": 504
            }
        elif self.exchange == "huobi":
            error_codes = {
                "TOO_MANY_REQUESTS": 429,
                "TIMEOUT": 504
            }
        else:
            raise Exception("Exchange not supported")
        return error_codes

    def get_error_code(self, error_name: str):
        return self.error_codes[error_name]

class HistoricalKlines:
    """
    Gets historical klines for input symbol
    """
```

```
    def __init__(self, exchange: str, symbol: str, interval_seconds: int, interface:
Interface):
        self.exchange = exchange
        self.symbol = symbol
        # convert interval to exchange specific interval
        self.interval = self._get_exchange_interval(interval_seconds)
        self.interface = interface
        self.store = DataStore(self.exchange, self.symbol, metric="klines",
csv_name="data/{exchange}/kline_history
/{symbol}_{interval}.csv".format(exchange=self.exchange, symbol=self.symbol,
interval=self.interval))
        self.error_codes = ErrorCodes(self.exchange)

    def _get_exchange_interval(self, interval_seconds: int):
        kline_intervals = KlineIntervals(self.exchange)
        for interval, seconds in kline_intervals.intervals.items():
            if seconds == interval_seconds:
                return interval
        raise Exception("Interval not supported")

    def _format_klines(self, data):
        if type(data) is dict:
            # is a dict in form {id (unix time), open, close, low, high, amount, vol,
count}
            #print(candles)
            return [data["id"], data["open"], data["close"], data["high"], data["low"],
data["vol"], data["amount"]]
        # is a list in form [start_time, end_time, open_price, close_price, high_price,
low_price, volume]
        return [data[0], data[6], data[1], data[2], data[3], data[4], data[5]]

    def _store_klines(self, klines):
        try:
            if type(klines) is dict:
                candles = klines["data"]
            else:
                candles = klines

            for candle in candles:
                candle = self._format_klines(candle)
                self.store.write_data_to_csv(candle, id_index=0)
            return
        except Exception as e:
            self._error_handler(e)

    def _error_handler(self, error):
        if type(error) is dict:
            if (error["code"] == self.error_codes.get_error_code("TOO_MANY_REQUESTS")):
                print("Too many requests, sleeping for {DELAY_AFTER_TIMEOUT}  seconds")
                time.sleep(DELAY_AFTER_TIMEOUT)
                self.save_klines()
            elif (error["code"] == self.error_codes.get_error_code("TIMEOUT")):
                print("Timeout, sleeping for {DELAY_AFTER_TIMEOUT}  seconds")
                time.sleep(DELAY_AFTER_TIMEOUT)
                self.save_klines()
            else:
                print("Error getting klines for {symbol} on {exchange} with interval
{interval}".format(symbol=self.symbol, exchange=self.exchange, interval=self.interval))
                print(error)
        else:
```

```
            print("Error getting klines for {symbol} on {exchange} with interval
{interval}".format(symbol=self.symbol, exchange=self.exchange, interval=self.interval))
            print(error)

    def save_klines(self):
        print("Getting klines for {symbol} on {exchange} with interval
{interval}".format(symbol=self.symbol, exchange=self.exchange, interval=self.interval))
        klines = self.interface.get_kline_history(self.symbol, self.interval, 1000)
        self._store_klines(klines)


# Threading classes
class ThreadingBase(threading.Thread):
    def __init__(self, thread_id: str, name: str, exchange: str, symbol: str, metric: str,
kl_interval_seconds: int = None, sleep: int = None, data_store: DataStore = None, duration:
int = None):
        threading.Thread.__init__(self)
        self.thread_id = thread_id
        self.name = name
        self.exchange = exchange
        self.symbol = symbol
        self.metric = metric
        self.sleep = sleep

        self.kl_interval_seconds = kl_interval_seconds
        if self.kl_interval_seconds is None:
            self.kl_interval_seconds = 60
        self.kl_interval = self._get_kline_interval_from_seconds(self.kl_interval_seconds)

        self.data_store = data_store
        if self.data_store is None:
            self.data_store = DataStore(self.exchange, self.symbol, self.metric,
f"data/{self.exchange}/{self.metric}/{self.symbol}.csv")

        self.duration = duration
        if self.duration is None:
            self.duration = 999999

        self.start_time = time.time()
        self._stop_event = threading.Event()

    def _get_api(self):
        api_factory = APIFactory(self.exchange)
        return api_factory.get_api()

    def _timeout_cb(self):
        current_duration = time.time() - self.start_time
        if current_duration > self.duration:
            print(f"{self.name} - Timeout reached")
            return True
        else:
            return False

    def _get_kline_interval_from_seconds(self, seconds: int):
        kline_intervals = KlineIntervals(self.exchange)
        for interval, interval_seconds in kline_intervals.intervals.items():
            if interval_seconds == seconds:
                return interval
        raise Exception("Interval not supported")
```

```
    def _check_kl_interval(self):
        # check if kline interval is valid
        kline_intervals = KlineIntervals(self.exchange)
        if self.kl_interval not in kline_intervals.get_intervals():
            raise Exception(f"Invalid kline interval {self.kl_interval}")

    def _get_kl_interval_seconds(self):
        kline_intervals = KlineIntervals(self.exchange)
        return kline_intervals.get_interval_seconds(self.kl_interval)

    def stop(self):
        self._stop_event.set()

    def stopped(self):
        return self._stop_event.is_set()

    def run(self):
        print(f"Starting {self.name}")
        self.start_time = time.time()
        self.collection_loop()
        print(f"Exiting {self.name}")
        self.stop()

    def collection_loop(self):
        raise Exception("Not implemented")

# -------------------- HB data collection threads --------------------
class HBTradingDataCollectionThread(ThreadingBase):
    def __init__(self, thread_id: str, name: str, exchange: str, symbol: str, metric: str,
kl_interval: int = None, sleep: int = None, data_store: DataStore = None, duration: int =
None):
        ThreadingBase.__init__(self, thread_id, name, exchange, symbol, metric,
kl_interval, sleep, data_store, duration)

    def trading_data_callback(self, trade_data: TradeDetailReq):
        #self.data_store.store_data(trade_data)
        trade_list = trade_data.data
        data = []
        for trade in trade_list:
            print(f"{self.name} - {trade.tradeId} - {trade.price} - {trade.amount} -
{trade.direction} - {trade.ts}")
            try:
                self.data_store.write_data_to_csv([trade.tradeId, trade.price,
trade.amount, trade.direction, trade.ts], id_index=0)
            except Exception as e:
                print(f"{self.name} - {e}")

        if (self._timeout_cb()):
            self.stop()

    def collection_loop(self):
        api = self._get_api()
        api.request_trades(self.symbol, self.trading_data_callback)
        while (not self._timeout_cb()):
            time.sleep(self.sleep)
            api.request_trades(self.symbol, self.trading_data_callback)


class HBKlineDataCollectionThread(ThreadingBase):
```

```
    def __init__(self, thread_id: str, name: str, exchange: str, symbol: str, metric: str,
kl_interval: int = None, sleep: int = None, data_store: DataStore = None, duration: int =
None):
        ThreadingBase.__init__(self, thread_id, name, exchange, symbol, metric,
kl_interval, sleep, data_store, duration)

    def kline_data_callback(self, kline_data: CandlestickEvent):
        #self.data_store.store_data(kline_data)
        kline_tick = kline_data.tick # Candlestick object
        try:
            self.data_store.write_data_to_csv([kline_tick.id, kline_tick.amount,
kline_tick.close, kline_tick.count, kline_tick.high, kline_tick.low, kline_tick.open,
kline_tick.vol], id_index=0)
        except Exception as e:
            print(f"{self.name} - {e}")

        if (self._timeout_cb()):
            self.stop()

    def collection_loop(self):
        api = self._get_api()
        api.subscribe_to_candlestick(self.symbol, interval=self.kl_interval,
callback_func=self.kline_data_callback)
        #while (not self._timeout_cb()):
        #    time.sleep(self.interval)


# -------------------- KUCOIN data collection threads --------------------
class KCKlineDataCollectionThread(ThreadingBase):
    def __init__(self, thread_id: str, name: str, exchange: str, symbol: str, metric: str,
kl_interval: int = None, sleep: int = None, data_store: DataStore = None, duration: int =
None):
        ThreadingBase.__init__(self, thread_id, name, exchange, symbol, metric,
kl_interval, sleep, data_store, duration)

    def _process_data(self, candles: list):
        start_time = candles[0]
        end_time = candles[6]
        open_price = candles[1]
        close_price = candles[2]
        high_price = candles[3]
        low_price = candles[4]
        volume = candles[5]
        return [start_time, end_time, open_price, close_price, high_price, low_price,
volume]

    def kline_data_callback(self, kline_data: dict):
        kline_tick = kline_data["data"]
        candles = kline_tick["candles"]
        try:
            self.data_store.write_data_to_csv(self._process_data(candles), id_index=0)
        except Exception as e:
            print(f"{self.name} - {e}")

        if (self._timeout_cb()):
            self.stop()

    def collection_loop(self):
        api = self._get_api()
        api.subscribe_to_candlestick(self.symbol, interval=self.kl_interval,
callback_func=self.kline_data_callback, duration=self.duration) # needs additional duration
parameter
```

```
        #while (not self._timeout_cb()):
        #    time.sleep(self.interval)

# -------------------- BINANCE data collection threads --------------------
class BNCandlestickDataCollectionThread(ThreadingBase):
    def __init__(self, thread_id: str, name: str, exchange: str, symbol: str, metric: str,
kl_interval: int = None, sleep: int = None, data_store: DataStore = None, duration: int =
None):
        ThreadingBase.__init__(self, thread_id, name, exchange, symbol, metric,
kl_interval, sleep, data_store, duration)

    def _process_data(self, candles: list):
        start_time = candles[0]
        end_time = candles[6]
        open_price = candles[1]
        close_price = candles[2]
        high_price = candles[3]
        low_price = candles[4]
        volume = candles[5]
        return [start_time, end_time, open_price, close_price, high_price, low_price,
volume]

    def kline_data_callback(self, _, kline_data: dict): # has extra parameter
        # enforce dict type - bn api returns string
        if (type(kline_data) == str):
            kline_data = json.loads(kline_data)
        kline_tick = kline_data["result"][0]
        try:
            self.data_store.write_data_to_csv(self._process_data(kline_tick), id_index=0)
        except Exception as e:
            print(f"{self.name} - {e}")

        if (self._timeout_cb()):
            self.stop()

    def collection_loop(self):
        api = self._get_api()
        while (not self._timeout_cb()):
            api.subscribe_to_candlestick(self.symbol, interval=self.kl_interval,
callback_func=self.kline_data_callback)
            time.sleep(self.kl_interval_seconds)

# ---------------------------- FUNCTIONS ----------------------------

# ---------------------------- BINANCE ----------------------------
def binance_setup():
    """
    Returns API instance and list of supported symbols
    """
    binance_api = APIFactory("binance").get_api()
    binance_symbols_manager = SymbolManagerFactory("binance",
binance_api).get_symbol_manager()

    # get coins that are online and not excluded
    binance_symbols = binance_api.get_symbols()
    bn_symbols = binance_set_coins_to_track(binance_symbols, binance_symbols_manager)
    return binance_api, bn_symbols

def binance_set_coins_to_track(bn_symbols: list, bn_symbols_manager: SymbolsManagerBase):
    """
```

```
    Returns list of coins to track
    """
    bn_symbols_df = bn_symbols_manager.convert_to_dataframe(bn_symbols)
    #bn_symbols_included =
bn_symbols_manager.convert_to_list(bn_symbols_manager.filter_excluded(bn_symbols_df))
    bn_symbols_online =
bn_symbols_manager.convert_to_list(bn_symbols_manager.filter_offline(bn_symbols_df))
    #bn_symbols = list(set(bn_symbols_included) & set(bn_symbols_online))
    bn_symbols = bn_symbols_online
    # temp set to BTC AND ETH
    #bn_symbols = ["BTCUSDT", "ETHUSDT"]
    return bn_symbols

def binance_get_klines(bn_api, bn_symbols: str):
    """
    Create threads to get kline data for each symbol
    """
    threads = []
    for symbol in bn_symbols:
        bn_thread = BNCandlestickDataCollectionThread(thread_id=f"BN_{symbol}",
name=f"BN_{symbol}", exchange="binance", symbol=symbol, metric="klines",
kl_interval=KLINE_INTERVAL_SECONDS, sleep=INTERVAL, duration=60)
        threads.append(bn_thread)

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()


# ---------------------------- KUCOIN ----------------------------
def kucoin_setup():
    """
    Returns API instance and list of supported symbols
    """
    kucoin_api = APIFactory("kucoin").get_api()
    kucoin_symbols_manager = SymbolManagerFactory("kucoin",
kucoin_api).get_symbol_manager()

    # get coins that are online and not excluded
    kucoin_symbols = kucoin_api.get_symbols()
    kc_symbols = kucoin_set_coins_to_track(kucoin_symbols, kucoin_symbols_manager)
    return kucoin_api, kc_symbols

def kucoin_set_coins_to_track(kc_symbols: list, kc_symbols_manager: SymbolsManagerBase):
    """
    Returns list of coins to track
    """
    kc_symbols_df = kc_symbols_manager.convert_to_dataframe(kc_symbols)
    kc_symbols_included = kc_symbols_manager.filter_excluded(kc_symbols_df)
    kc_symbols_online = kc_symbols_manager.filter_offline(kc_symbols_df)
    kc_symbols = list(set(kc_symbols_included) & set(kc_symbols_online))
    kc_symbols = ["BTC-USDT", "ETH-USDT"]
    return kc_symbols

def kucoin_get_klines(kc_api, kc_symbols: list):
    """
    Create threads to get klines for each symbol
    """
```

```
    threads = []
    # create new threads
    print("Creating threads")
    for symbol in kc_symbols:
        thread = KCKlineDataCollectionThread(thread_id="temp", name=f"{symbol}_klines",
exchange="kucoin", symbol=symbol, metric="klines", kl_interval=KLINE_INTERVAL_SECONDS,
sleep=INTERVAL, duration=DURATION)
        threads.append(thread)
        print(f"Created thread for {symbol}")
    # start new threads
    for t in threads:
        t.start()
    # wait for all threads to complete
    for t in threads:
        t.join()


# ---------------------------- HUOBI ----------------------------
def huobi_setup():
    """
    Returns API instance and list of supported symbols
    """
    huobi_api = APIFactory("huobi").get_api()
    huobi_symbols_manager = SymbolManagerFactory("huobi", huobi_api).get_symbol_manager()

    # get coins that are online and not excluded
    huobi_symbols = huobi_api.get_symbols()
    hb_symbols = huobi_set_coins_to_track(huobi_symbols, huobi_symbols_manager)
    return huobi_api, hb_symbols

def huobi_set_coins_to_track(hb_symbols: list, hb_symbols_manager: SymbolsManagerBase):
    """
    Returns list of coins to track
    """
    hb_symbols_df = hb_symbols_manager.convert_to_dataframe(hb_symbols)
    hb_symbols_excluded = hb_symbols_manager.filter_excluded(hb_symbols_df, EXCLUDED_COINS)
    hb_symbols_offline = hb_symbols_manager.filter_offline(hb_symbols_df)
    hb_symbols = list(set(hb_symbols_excluded) & set(hb_symbols_offline))
    hb_symbols = ["btcusdt", "ethusdt"]
    return hb_symbols

def huobi_staggered_get_trades(hb_api, hb_symbols):
    # Create threads for x symbols at a time to avoid rate limit
    threads = []
    for i in range(0, len(hb_symbols), SIMULTANEOUS_REQUESTS):
        threads.append(threading.Thread(target=huobi_get_trades, args=(hb_api,
hb_symbols[i:i+SIMULTANEOUS_REQUESTS])))
        for t in threads:
            t.start()
        for t in threads:
            t.join()
        threads = []
        time.sleep(SLEEP_BETWEEN_THREAD_GEN)

def huobi_get_trades(hb_api, hb_symbols):
    """
    Create threads for each symbol and start collecting trades
    """
    threads = []
```

```
    # create new threads
    print("Creating threads")
    for symbol in hb_symbols:
        thread = HBTradingDataCollectionThread(thread_id="temp", name=f"{symbol}_trades",
exchange="huobi", symbol=symbol, metric="trades", kl_interval=KLINE_INTERVAL,
sleep=INTERVAL, duration=DURATION)
        threads.append(thread)
        print(f"Created thread for {symbol}")
    # start new threads
    for t in threads:
        t.start()
    # wait for all threads to complete
    for t in threads:
        t.join()

def huobi_staggered_get_klines(hb_api, hb_symbols):
    # Create threads for x symbols at a time to avoid rate limit
    threads = []
    for i in range(0, len(hb_symbols), SIMULTANEOUS_REQUESTS):
        threads.append(threading.Thread(target=huobi_get_klines, args=(hb_api,
hb_symbols[i:i+SIMULTANEOUS_REQUESTS])))
        for t in threads:
            t.start()
        for t in threads:
            t.join()
        threads = []
        time.sleep(SLEEP_BETWEEN_THREAD_GEN)

def huobi_get_klines(hb_api, hb_symbols):
    """
    Create threads for each symbol and start collecting klines
    """
    threads = []
    # create new threads
    print("Creating threads")
    for symbol in hb_symbols:
        thread = HBKlineDataCollectionThread(thread_id="temp", name=f"{symbol}_klines",
exchange="huobi", symbol=symbol, metric="klines", kl_interval=KLINE_INTERVAL_SECONDS,
sleep=INTERVAL, duration=DURATION)
        threads.append(thread)
        print(f"Created thread for {symbol}")
    # start new threads
    for t in threads:
        t.start()
    # wait for all threads to complete
    for t in threads:
        t.join()


# ---------------------------- MAIN ----------------------------
def run_hb_threads():
    huobi_api, huobi_symbols = huobi_setup()
    #huobi_get_trades(huobi_api, huobi_symbols)
    #huobi_staggered_get_trades(huobi_api, huobi_symbols)
    huobi_staggered_get_klines(huobi_api, huobi_symbols)

def run_kc_threads():
    kc_api, kc_symbols = kucoin_setup()
    #kucoin_get_trades(kc_api, kc_symbols)
    print(kc_symbols)
```

```
    kucoin_get_klines(kc_api, kc_symbols)

def run_bn_threads():
    bn_api, bn_symbols = binance_setup()
    #binance_get_trades(bn_api, bn_symbols)
    binance_get_klines(bn_api, bn_symbols)

def get_historical_all():
    # Setup
    print("Setting up")
    huobi_api, huobi_symbols = huobi_setup()
    kc_api, kc_symbols = kucoin_setup()
    bn_api, bn_symbols = binance_setup()

    print("Getting historical klines")
    # Get max historical klines for each exchange
    #for symbol in huobi_symbols:
    #    HistoricalKlines("huobi", symbol, KLINE_INTERVAL_SECONDS, huobi_api).save_klines()
    #    time.sleep(SLEEP_BETWEEN_HISTORY_REQUESTS)
    for symbol in kc_symbols:
        HistoricalKlines("kucoin", symbol, KLINE_INTERVAL_SECONDS, kc_api).save_klines()
        time.sleep(SLEEP_BETWEEN_HISTORY_REQUESTS)
    #for symbol in bn_symbols:
    #    HistoricalKlines("binance", symbol, KLINE_INTERVAL_SECONDS, bn_api).save_klines()
    #    time.sleep(SLEEP_BETWEEN_HISTORY_REQUESTS)

def all_threads():
    # Setup
    huobi_api, huobi_symbols = huobi_setup()
    kc_api, kc_symbols = kucoin_setup()
    bn_api, bn_symbols = binance_setup()

    print("Starting threads")
    # Create threads
    threads = []
    # Huobi
    for symbol in huobi_symbols:
        thread = HBKlineDataCollectionThread(thread_id="temp", name=f"{symbol}_klines",
exchange="huobi", symbol=symbol, metric="klines", kl_interval=KLINE_INTERVAL_SECONDS,
sleep=INTERVAL, duration=DURATION)
        threads.append(thread)
    # Kucoin
    for symbol in kc_symbols:
        thread = KCKlineDataCollectionThread(thread_id="temp", name=f"{symbol}_klines",
exchange="kucoin", symbol=symbol, metric="klines", kl_interval=KLINE_INTERVAL_SECONDS,
sleep=INTERVAL, duration=DURATION)
        threads.append(thread)
    # Binance
    for symbol in bn_symbols:
        thread = BNCandlestickDataCollectionThread(thread_id="temp", name=f"
{symbol}_klines", exchange="binance", symbol=symbol, metric="klines",
kl_interval=KLINE_INTERVAL_SECONDS, sleep=INTERVAL, duration=DURATION)
        threads.append(thread)

    # Start threads - staggered
    for i in range(0, len(threads), SIMULTANEOUS_REQUESTS):
        for t in threads[i:i+SIMULTANEOUS_REQUESTS]:
            t.start()
        for t in threads[i:i+SIMULTANEOUS_REQUESTS]:
            t.join()
```

```
        time.sleep(SLEEP_BETWEEN_THREAD_GEN)


def main():
    #run_kc_threads()
    #run_hb_threads()
    #run_bn_threads()
    get_historical_all()
    #all_threads()


if __name__ == "__main__":
    main()
```