**Day1: 14-09-2025 Report:**

**Todays closing output is in the form of: (trade\_suggestions.csv)**

**This doc has:**

* **Output table**
* **What your current output has**
* **What you can do with this**
* **What you actually need to do next to reach your goal**
* **Your goal**
* **How far you are**
* **Are you on the right path?**
* **Codes of stage 1 output**

**Output table:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **symbol** | **timeframe** | **side** | **entry** | **SL** | **TP** | **expected\_duration\_min** | **win\_probability** | **avg\_rr** |
| BNB/USDT | 5m | SHORT | 923.62 | 925.15 | 920.56 | 31 | 66.67 | 2 |
| BNB/USDT | 15m | LONG | 923.62 | 920.71 | 929.45 | 153 | 45.9 | 2 |
| BNB/USDT | 1h | LONG | 923.62 | 918.3 | 934.26 | 598 | 42.31 | 2 |
| BNB/USDT | 12h | LONG | 923.62 | 907.94 | 954.97 | 9785 | 50 | 2 |
| BTC/USDT | 15m | LONG | 115413.5 | 115253.3 | 115733.7 | 131 | 45.45 | 2 |
| ETH/USDT | 15m | LONG | 4591 | 4574.71 | 4623.59 | 180 | 43.9 | 2 |
| ETH/USDT | 1h | LONG | 4591.01 | 4562.86 | 4647.3 | 696 | 33.33 | 2 |
| ETH/USDT | 12h | LONG | 4591.01 | 4473.08 | 4826.87 | 8526 | 42.11 | 2 |

**1️⃣ What your current output has**

**Your table shows, per symbol + timeframe:**

| **Column** | **Meaning** |
| --- | --- |
| **symbol** | **The trading pair (e.g., BTC/USDT)** |
| **timeframe** | **Candle timeframe (5m, 15m, 1h, 12h)** |
| **side** | **Suggested trade direction (LONG/SHORT) based on EMA+RSI** |
| **entry** | **Price to enter the trade (latest candle or signal point)** |
| **SL** | **Stop-loss price for the trade** |
| **TP** | **Take-profit price for the trade** |
| **expected\_duration\_min** | **Average historical duration to hit SL/TP (from backtests)** |
| **win\_probability** | **Historical win rate (TP hits / total trades) for this symbol + timeframe** |
| **avg\_rr** | **Average reward:risk ratio of your past trades** |

**Basically: It tells you, “Based on past data, if you took this trade now, historically it would hit TP with X% probability in Y minutes, and your average RR is Z.”**

**2️⃣ What you can do with this**

**Right now, it’s informational. You can:**

* **See which trades are historically more likely to win (win\_probability).**
* **Compare RR ratios — higher RR trades are more attractive.**
* **Filter trades to avoid low-probability setups.**
* **Start thinking about sizing positions based on RR and historical probability.**

**But, it’s not yet giving live predictions with confidence intervals — it’s based entirely on past performance.**

**3️⃣ What you actually need to do next to reach your goal**

**Your ultimate goal (as per your example output) is:**

**Suggest profitable future trades with: side, entry, SL, TP, expected duration, win probability, RR — essentially a ready-to-act trade recommendation.**

**To get there, you need to:**

1. **Fix your current signal logic:**
   * **Currently, your entry, SL, TP are inconsistent in some rows (SL > entry for LONG, TP < entry, etc.).**
   * **Ensure that for LONG: SL < entry < TP; for SHORT: TP < entry < SL.**
2. **Refine win probability:**
   * **Right now, win\_probability is averaged from past trades only.**
   * **You can improve it by weighting more recent trades, filtering out outliers, or splitting by similar market conditions.**
3. **Include more realistic expected duration:**
   * **Your expected\_duration\_min is purely average. You can also include median, min/max, or standard deviation.**
4. **Add confidence / filtering:**
   * **Only show trades with win\_probability > X% or RR > 1.5 to avoid bad setups.**
   * **Could use multiple signals (EMA+RSI+ATR trend + volume) to filter better.**
5. **Optionally simulate forward performance:**
   * **Run a mini backtest on the last N candles to estimate “next X candles” outcome — gives semi-live prediction.**

**4️⃣ Your goal**

**You want to move from historical backtest stats → actionable trade suggestions, like:**

**Symbol: BTC/USDT**

**Timeframe: 5m**

**Side: LONG**

**Entry: 27250**

**SL: 27100**

**TP: 27450**

**Expected duration: 35 min**

**Win probability: 55%**

**RR ratio: 1.8**

**Currently, your table is almost there, but SL/TP inconsistencies and naive statistics prevent you from making trustable trade suggestions.**

**5️⃣ How far you are**

**✅ You already have:**

* **Multi-symbol, multi-timeframe backtests.**
* **Equity curves and trade histories.**
* **Basic stats (win rate, RR, expected duration).**

**❌ Missing:**

* **Corrected SL/TP assignments for LONG vs SHORT.**
* **Probability confidence adjusted for recent data.**
* **Filters for high-quality trades.**
* **Automatic recommendation generation that can be trusted live.**

**6️⃣ Are you on the right path?**

**Yes — absolutely. Your backtester + trade summary + expected-duration table is the core foundation.**

**Next steps are polishing the signal and metrics, then you can generate real actionable trade suggestions.**

**Next in continuation we need to:**

**Get proper actionable signals with correct SL/TP, win probability, RR, and expected duration, essentially giving your “final-step trade suggestion table”**

**DAY2: 15-09-2025**

Your **end-goal** is → an **automated trading strategy framework** that:

* Scans multiple symbols + timeframes.
* Generates trade setups (entry, SL, TP, RR, probability).
* Filters & ranks trades (only best ones are executed).
* Optionally → auto-executes via exchange API (Binance/Bybit).

**What’s Missing / Next 🔜**

We now need to **transform raw signals → actionable trades**. That means:

1. **Filtering Engine**: Select only trades above certain probability & RR thresholds.
   * Example: Only take trades where win\_probability > 50% and avg\_rr >= 2.
2. **Ranking System**: If multiple trades qualify, rank by best risk-adjusted return.
3. **Trade Management Rules**:
   * Max open trades at once.
   * Risk per trade (% of capital).
   * Max daily loss/win.
4. **Execution Layer (later)**: Connect to Binance API for real orders.
5. **Performance Tracking**: Backtest + log live trades → measure if strategy is profitable.

Codes.:

# strategy\_test.py

# strategy\_test.py

import pandas as pd

import ta

import matplotlib.pyplot as plt

# -------------------------

# Step 1: Load & Clean Data

# -------------------------

import glob

# Merge multiple CSVs in the folder

files = glob.glob("BTCUSDT-5m-2025-09/\*.csv")

df\_list = []

for file in files:

    temp = pd.read\_csv(file, header=None)

    temp.columns = [

        "open\_time", "open", "high", "low", "close", "volume",

        "close\_time", "quote\_asset\_volume", "number\_of\_trades",

        "taker\_buy\_base", "taker\_buy\_quote", "ignore"

    ]

    temp = temp[["open\_time","open","high","low","close","volume"]]

    temp["open\_time"] = pd.to\_datetime(temp["open\_time"].round(0).astype("int64"), unit="ms", errors="coerce")

    temp[["open","high","low","close","volume"]] = temp[["open","high","low","close","volume"]].astype(float)

    df\_list.append(temp)

df = pd.concat(df\_list, ignore\_index=True)

# -------------------------

# Step 2: Indicators

# -------------------------

df["EMA50"] = ta.trend.ema\_indicator(df["close"], window=50)

df["EMA200"] = ta.trend.ema\_indicator(df["close"], window=200)

df["RSI"] = ta.momentum.rsi(df["close"], window=14)

df["ATR"] = ta.volatility.average\_true\_range(df["high"], df["low"], df["close"], window=14)

# -------------------------

# Step 3: Strategy Logic & Backtest

# -------------------------

balance = 10000

risk = 0.02

equity\_curve = []

position = None

trade\_history = []

for i in range(200, len(df)):

    price = df["close"].iloc[i]

    time = df["open\_time"].iloc[i]

    # Close position if TP or SL is hit

    if position:

        if price <= position["SL"]:   # stop-loss

            balance -= position["risk\_amt"]

            position["result"] = "SL"

            position["exit\_price"] = price

            position["exit\_time"] = time

            trade\_history.append(position)

            position = None

        elif price >= position["TP"]: # take-profit

            balance += position["reward\_amt"]

            position["result"] = "TP"

            position["exit\_price"] = price

            position["exit\_time"] = time

            trade\_history.append(position)

            position = None

    # Open new position

    if not position:

        if df["EMA50"].iloc[i] > df["EMA200"].iloc[i] and df["RSI"].iloc[i] < 70:

            atr = df["ATR"].iloc[i]

            entry = price

            sl = entry - 1.5 \* atr

            tp = entry + 2 \* atr

            risk\_amt = balance \* risk

            reward\_amt = risk\_amt \* (2/1.5)

            position = {

                "entry\_time": time,

                "entry": entry,

                "SL": sl,

                "TP": tp,

                "risk\_amt": risk\_amt,

                "reward\_amt": reward\_amt,

                "result": None

            }

    equity\_curve.append(balance)

# -------------------------

# Step 4: Results

# -------------------------

print("Final Balance:", round(balance,2))

print("Net Profit:", round(balance-10000,2))

print("Total Trades Closed:", len(trade\_history))

# Print first 5 trade details

for t in trade\_history[:5]:

    print(t)

# -------------------------

# Step 5: Plot Equity Curve

# -------------------------

plt.figure(figsize=(12,6))

plt.plot(equity\_curve, label="Equity Curve")

plt.title("Strategy Equity Curve")

plt.xlabel("Time Steps (5-min bars)")

plt.ylabel("Balance")

plt.legend()

plt.show()

# backtester.py

import pandas as pd

import numpy as np

class Backtester:

    def \_\_init\_\_(self, df, strategy, initial\_balance=10000, risk=0.02):

        self.df = df.copy()

        self.strategy = strategy

        self.initial\_balance = initial\_balance

        self.balance = initial\_balance

        self.risk = risk

        self.equity\_curve = []

        self.trade\_history = []

        self.position = None

    def run(self):

        self.tp\_hits = 0

        self.sl\_hits = 0

        self.max\_drawdown = 0

        peak = self.balance

        for i in range(200, len(self.df)):

            price = self.df["close"].iloc[i]

            time = self.df["ts"].iloc[i]

            # --- Manage Open Trade ---

            if self.position:

                side = self.position["side"]

                if side == "long":

                    if price <= self.position["SL"]:

                        self.\_close\_trade("SL", price, time)

                        self.sl\_hits += 1

                    elif price >= self.position["TP"]:

                        self.\_close\_trade("TP", price, time)

                        self.tp\_hits += 1

                elif side == "short":

                    if price >= self.position["SL"]:

                        self.\_close\_trade("SL", price, time)

                        self.sl\_hits += 1

                    elif price <= self.position["TP"]:

                        self.\_close\_trade("TP", price, time)

                        self.tp\_hits += 1

            # --- Entry Logic ---

            if not self.position:

                signal = self.strategy(self.df, i)

                if signal:

                    self.\_open\_trade(signal, price, time)

            # Update equity

            self.equity\_curve.append(self.balance)

            peak = max(peak, self.balance)

            drawdown = peak - self.balance

            if drawdown > self.max\_drawdown:

                self.max\_drawdown = drawdown

        # Compute win rate

        win\_rate = (self.tp\_hits / len(self.trade\_history) \* 100) if self.trade\_history else 0

        return {

            "final\_balance": self.balance,

            "net\_profit": self.balance - self.initial\_balance,

            "trades": self.trade\_history,

            "equity\_curve": self.equity\_curve,

            "tp\_hits": self.tp\_hits,

            "sl\_hits": self.sl\_hits,

            "win\_rate": round(win\_rate, 2),

            "max\_drawdown": round(self.max\_drawdown, 2)

        }

    def \_open\_trade(self, signal, price, time):

        side, sl, tp = signal

        risk\_amt = self.balance \* self.risk

        self.position = {

            "side": side,

            "entry\_time": time,

            "entry": price,

            "SL": sl,

            "TP": tp,

            "risk\_amt": risk\_amt,

            "reward\_amt": risk\_amt \* (abs(tp - price) / abs(price - sl)),

        }

    def \_close\_trade(self, result, exit\_price, exit\_time):

        if result == "SL":

            self.balance -= self.position["risk\_amt"]

        else:

            self.balance += self.position["reward\_amt"]

        self.position["result"] = result

        self.position["exit\_price"] = exit\_price

        self.position["exit\_time"] = exit\_time

        self.trade\_history.append(self.position)

        self.position = None

# ema\_rsi.py

# ema\_rsi.py

import ta

def ema\_rsi\_strategy(df, i):

    """

    Returns (side, SL, TP) or None

    Dynamic ATR-based SL/TP + trend confirmation

    """

    # Compute indicators only once

    if "EMA50" not in df:

        df["EMA50"] = ta.trend.ema\_indicator(df["close"], 50)

        df["EMA200"] = ta.trend.ema\_indicator(df["close"], 200)

        df["RSI"] = ta.momentum.rsi(df["close"], 14)

        df["ATR"] = ta.volatility.average\_true\_range(df["high"], df["low"], df["close"], 14)

        df["ADX"] = ta.trend.adx(df["high"], df["low"], df["close"], 14)

    price = df["close"].iloc[i]

    atr = df["ATR"].iloc[i]

    adx = df["ADX"].iloc[i]

    # Only trade strong trends (ADX > 20)

    if adx < 20:

        return None

    # Dynamic SL/TP

    if df["EMA50"].iloc[i] > df["EMA200"].iloc[i] and df["RSI"].iloc[i] < 70:

        sl = price - 1.0 \* atr      # tighter SL

        tp = price + 2.0 \* atr      # wider TP

        return ("long", sl, tp)

    if df["EMA50"].iloc[i] < df["EMA200"].iloc[i] and df["RSI"].iloc[i] > 30:

        sl = price + 1.0 \* atr

        tp = price - 2.0 \* atr

        return ("short", sl, tp)

    return None

# agent.py

# agent.py

import os

import time

import pandas as pd

import ccxt

from datetime import datetime, timedelta

# ======================

# CONFIGURATION

# ======================

SYMBOLS = ["BTC/USDT", "ETH/USDT", "BNB/USDT"]

TIMEFRAMES = ["5m", "15m", "1h", "12h"]

DATA\_DIR = "data"

LIMIT = 100000  # max candles per fetch

KEEP\_DAYS\_MAP = {

    "5m": 30,

    "15m": 60,

    "1h": 180,

    "12h": 365,

}

exchange = ccxt.binance({'enableRateLimit': True})

# ======================

# HELPERS

# ======================

def ensure\_dir(path: str):

    if not os.path.exists(path):

        os.makedirs(path)

def fetch\_ohlcv(symbol: str, timeframe: str, since=None, limit=LIMIT):

    """Fetch OHLCV data from Binance with pagination"""

    all\_data = []

    while True:

        candles = exchange.fetch\_ohlcv(symbol, timeframe, since=since, limit=limit)

        if not candles:

            break

        all\_data.extend(candles)

        since = candles[-1][0] + 1

        time.sleep(0.5)

        if len(candles) < limit:

            break

    return all\_data

def load\_existing\_csv(symbol: str, timeframe: str):

    out\_dir = os.path.join(DATA\_DIR, symbol.replace("/", ""))

    out\_path = os.path.join(out\_dir, f"{timeframe}.csv")

    if os.path.exists(out\_path):

        df = pd.read\_csv(out\_path, parse\_dates=["ts"])

        return df, out\_path

    return pd.DataFrame(), out\_path

def update\_csv(symbol: str, timeframe: str):

    """Update CSV with latest candles + auto-clean"""

    df\_old, out\_path = load\_existing\_csv(symbol, timeframe)

    if not df\_old.empty:

        last\_ts = int(df\_old["ts"].iloc[-1].timestamp() \* 1000)

    else:

        last\_ts = None

    new\_data = fetch\_ohlcv(symbol, timeframe, since=last\_ts)

    if not new\_data and not df\_old.empty:

        print(f"[SKIP] {symbol} {timeframe} already up-to-date.")

        return df\_old, out\_path

    df\_new = pd.DataFrame(new\_data, columns=["ts", "open", "high", "low", "close", "volume"])

    df\_new["ts"] = pd.to\_datetime(df\_new["ts"], unit="ms")

    df\_new[["open", "high", "low", "close", "volume"]] = df\_new[

        ["open", "high", "low", "close", "volume"]

    ].astype(float)

    df\_final = pd.concat([df\_old, df\_new]).drop\_duplicates(subset="ts").reset\_index(drop=True)

    # auto-clean

    keep\_days = KEEP\_DAYS\_MAP.get(timeframe, 90)

    cutoff = datetime.now() - timedelta(days=keep\_days)

    df\_final = df\_final[df\_final["ts"] >= cutoff].reset\_index(drop=True)

    ensure\_dir(os.path.dirname(out\_path))

    df\_final.to\_csv(out\_path, index=False)

    print(f"[OK] {symbol} {timeframe} updated -> {len(df\_final)} rows (kept {keep\_days} days)")

    return df\_final, out\_path

def merge\_timeframes(symbol: str, timeframes=TIMEFRAMES):

    """Merge multiple timeframe CSVs into one aligned dataset"""

    symbol\_dir = os.path.join(DATA\_DIR, symbol.replace("/", ""))

    merged = None

    for tf in timeframes:

        tf\_path = os.path.join(symbol\_dir, f"{tf}.csv")

        if not os.path.exists(tf\_path):

            continue

        df = pd.read\_csv(tf\_path, parse\_dates=["ts"])

        df = df[["ts", "open", "high", "low", "close", "volume"]]

        df = df.rename(columns={c: f"{c}\_{tf}" for c in ["open","high","low","close","volume"]})

        if merged is None:

            merged = df

        else:

            merged = pd.merge\_asof(

                merged.sort\_values("ts"),

                df.sort\_values("ts"),

                on="ts",

                direction="backward"

            )

    if merged is not None:

        out\_path = os.path.join(symbol\_dir, "merged\_multi\_tf.csv")

        merged.to\_csv(out\_path, index=False)

        print(f"[MERGED] {symbol} -> {out\_path} ({len(merged)} rows)")

    else:

        print(f"[WARN] No data to merge for {symbol}")

# ======================

# MAIN

# ======================

if \_\_name\_\_ == "\_\_main\_\_":

    print(f"=== Daily Update Started @ {datetime.now()} ===")

    for sym in SYMBOLS:

        for tf in TIMEFRAMES:

            try:

                update\_csv(sym, tf)

            except Exception as e:

                print(f"[ERR] {sym} {tf}: {e}")

        # After all timeframes updated → merge them

        merge\_timeframes(sym)

    print(f"=== Update Finished @ {datetime.now()} ===")

# merge.py

import os

import pandas as pd

DATA\_DIR = "data"   # where your CSVs are stored

TIMEFRAMES = ["5m", "15m", "1h", "12h"]

def load\_timeframe(symbol: str, timeframe: str) -> pd.DataFrame:

    """Load one timeframe CSV for given symbol"""

    path = os.path.join(DATA\_DIR, symbol, f"{timeframe}.csv")

    if not os.path.exists(path):

        print(f"[WARN] Missing file: {path}")

        return pd.DataFrame()

    df = pd.read\_csv(path, parse\_dates=["ts"])

    df = df[["ts", "open", "high", "low", "close", "volume"]]

    # Add suffix for columns to avoid clashes

    suffix = f"\_{timeframe}"

    df = df.rename(columns={c: c + suffix for c in ["open","high","low","close","volume"]})

    return df

def merge\_timeframes(symbol: str, timeframes=TIMEFRAMES) -> pd.DataFrame:

    """Merge multiple timeframe CSVs into one aligned dataset"""

    merged = None

    for tf in timeframes:

        df = load\_timeframe(symbol, tf)

        if df.empty:

            continue

        if merged is None:

            merged = df

        else:

            merged = pd.merge\_asof(

                merged.sort\_values("ts"),

                df.sort\_values("ts"),

                on="ts",

                direction="backward"  # align smaller candles to latest higher timeframe

            )

    return merged

def merge\_all\_symbols():

    """Loop over all symbol folders and create merged dataset"""

    for symbol\_folder in os.listdir(DATA\_DIR):

        folder\_path = os.path.join(DATA\_DIR, symbol\_folder)

        if not os.path.isdir(folder\_path):

            continue

        print(f"\n[INFO] Processing {symbol\_folder}...")

        merged\_df = merge\_timeframes(symbol\_folder)

        if merged\_df is None or merged\_df.empty:

            print(f"[SKIP] No data for {symbol\_folder}")

            continue

        out\_path = os.path.join(folder\_path, "merged\_multi\_tf.csv")

        merged\_df.to\_csv(out\_path, index=False)

        print(f"[OK] Saved {out\_path} ({len(merged\_df)} rows)")

if \_\_name\_\_ == "\_\_main\_\_":

    merge\_all\_symbol

# run.py

# run.py

import os

import pandas as pd

from backtester import Backtester

from ema\_rsi import ema\_rsi\_strategy  # your strategy

from datetime import timedelta

DATA\_DIR = "data"

TIMEFRAMES = ["5m", "15m", "1h", "12h"]

SUGGESTIONS\_CSV = "trade\_suggestions.csv"

def load\_data(symbol: str, timeframe: str):

    path = os.path.join(DATA\_DIR, symbol.replace("/", ""), f"{timeframe}.csv")

    if not os.path.exists(path):

        return None

    df = pd.read\_csv(path)

    df["ts"] = pd.to\_datetime(df["ts"])

    df[["open","high","low","close","volume"]] = df[["open","high","low","close","volume"]].astype(float)

    return df

def compute\_signal\_stats(trade\_history, side):

    """Compute win probability, avg duration, RR ratio"""

    if not trade\_history:

        return 0, 0, 0

    # Filter trades by side

    trades = [t for t in trade\_history if t["side"] == side]

    if not trades:

        return 0, 0, 0

    wins = [t for t in trades if t["result"]=="TP"]

    win\_prob = len(wins)/len(trades)\*100

    durations = [(t["exit\_time"] - t["entry\_time"]).total\_seconds()/60 for t in trades]

    avg\_duration = round(sum(durations)/len(durations)) if durations else 0

    rr\_ratios = [abs(t["TP"] - t["entry"])/abs(t["entry"] - t["SL"]) for t in trades]

    avg\_rr = round(sum(rr\_ratios)/len(rr\_ratios),2) if rr\_ratios else 0

    return win\_prob, avg\_duration, avg\_rr

def generate\_trade\_suggestions(symbols):

    suggestions = []

    for sym in symbols:

        for tf in TIMEFRAMES:

            df = load\_data(sym, tf)

            if df is None or len(df)<200:

                continue

            bt = Backtester(df, ema\_rsi\_strategy)

            results = bt.run()

            # Take last candle signal

            signal = ema\_rsi\_strategy(df, len(df)-1)

            if signal is None:

                continue

            side, sl, tp = signal

            last\_price = df["close"].iloc[-1]

            # Compute historical stats

            win\_prob, avg\_duration, avg\_rr = compute\_signal\_stats(results["trades"], side)

            suggestions.append({

                "symbol": sym,

                "timeframe": tf,

                "side": side.upper(),

                "entry": round(last\_price,2),

                "SL": round(sl,2),

                "TP": round(tp,2),

                "expected\_duration\_min": avg\_duration,

                "win\_probability": round(win\_prob,2),

                "avg\_rr": avg\_rr

            })

    return pd.DataFrame(suggestions)

if \_\_name\_\_ == "\_\_main\_\_":

    # detect symbols from folders

    symbols = [d for d in os.listdir(DATA\_DIR) if os.path.isdir(os.path.join(DATA\_DIR, d))]

    symbols = [s.replace("USDT","/USDT") for s in symbols]

    df\_suggestions = generate\_trade\_suggestions(symbols)

    print("\n=== TRADE SUGGESTIONS ===")

    print(df\_suggestions)

    df\_suggestions.to\_csv(SUGGESTIONS\_CSV, index=False)

    print(f"\nSaved trade suggestions -> {SUGGESTIONS\_CSV}")

DATE: 21-09-2025

Stage 2:

# ema\_rsi\_stage2.py

# ema\_rsi\_stage2.py

import ta

import numpy as np

def ensure\_sl\_tp\_orientation(side: str, entry: float, sl: float, tp: float):

    """

    Ensure SL/TP follow convention:

      - LONG:  SL < entry < TP

      - SHORT: TP < entry < SL

    If not, fix automatically (swap or nudge slightly).

    """

    if side.lower() == "long":

        # if SL >= entry or TP <= entry, correct them

        if sl >= entry:

            sl = entry - abs(entry - sl) - 1e-6

        if tp <= entry:

            tp = entry + abs(tp - entry) + 1e-6

    else:  # short

        if sl <= entry:

            sl = entry + abs(entry - sl) + 1e-6

        if tp >= entry:

            tp = entry - abs(tp - entry) - 1e-6

    return sl, tp

def ema\_rsi\_strategy(df, i, adx\_threshold=20):

    """

    Stage-2 EMA+RSI strategy with ADX filter and ATR-based dynamic SL/TP.

    returns (side, SL, TP) or None.

    """

    # compute indicators once

    if "EMA50" not in df:

        df["EMA50"] = ta.trend.ema\_indicator(df["close"], window=50)

        df["EMA200"] = ta.trend.ema\_indicator(df["close"], window=200)

        df["RSI"] = ta.momentum.rsi(df["close"], window=14)

        df["ATR"] = ta.volatility.average\_true\_range(df["high"], df["low"], df["close"], window=14)

        df["ADX"] = ta.trend.adx(df["high"], df["low"], df["close"], window=14).fillna(0)

    # protect against NaNs early in series

    if i < 200 or np.isnan(df["ATR"].iloc[i]) or np.isnan(df["EMA50"].iloc[i]) :

        return None

    price = float(df["close"].iloc[i])

    atr = float(df["ATR"].iloc[i])

    adx = float(df["ADX"].iloc[i])

    rsi = float(df["RSI"].iloc[i])

    ema50 = float(df["EMA50"].iloc[i])

    ema200 = float(df["EMA200"].iloc[i])

    # Trend strength filter

    if adx < adx\_threshold:

        return None

    # LONG rules

    if ema50 > ema200 and rsi < 75:

        sl = price - 1.0 \* atr      # tighter SL for stage-2

        tp = price + 1.8 \* atr

        sl, tp = ensure\_sl\_tp\_orientation("long", price, sl, tp)

        return ("long", sl, tp)

    # SHORT rules

    if ema50 < ema200 and rsi > 25:

        sl = price + 1.0 \* atr

        tp = price - 1.8 \* atr

        sl, tp = ensure\_sl\_tp\_orientation("short", price, sl, tp)

        return ("short", sl, tp)

    return None

# run\_stage2.py

# run\_stage2.py

import os

import math

import pandas as pd

from datetime import datetime

from backtester import Backtester

from ema\_rsi\_stage2 import ema\_rsi\_strategy

DATA\_DIR = "data"

TIMEFRAMES = ["5m", "15m", "1h", "12h"]

SUGGESTIONS\_CSV = "trade\_suggestions\_stage2.csv"

SUGGESTIONS\_JSON = "trade\_suggestions\_stage2.json"

# Register strategies here (name -> function)

STRATEGIES = {

    "ema\_rsi": ema\_rsi\_strategy,

    # add more strategies here in future

}

RECENT\_TRADE\_WINDOW = 50   # use last N trades for statistics (Stage-2 uses recent history)

def load\_df(symbol: str, timeframe: str):

    path = os.path.join(DATA\_DIR, symbol.replace("/", ""), f"{timeframe}.csv")

    if not os.path.exists(path):

        return None

    df = pd.read\_csv(path)

    df["ts"] = pd.to\_datetime(df["ts"])

    df[["open","high","low","close","volume"]] = df[["open","high","low","close","volume"]].astype(float)

    return df

def sample\_recent\_trades(trades, side, n=RECENT\_TRADE\_WINDOW):

    # filter by side and take last n trades by entry\_time

    trades\_side = [t for t in trades if t.get("side","").lower() == side.lower()]

    trades\_side\_sorted = sorted(trades\_side, key=lambda x: x["entry\_time"])  # oldest->newest

    return trades\_side\_sorted[-n:]

def compute\_side\_stats(trades, side):

    """Compute win\_prob (%), avg\_duration (minutes), avg\_rr from recent trades of same side"""

    recent = sample\_recent\_trades(trades, side, RECENT\_TRADE\_WINDOW)

    if not recent:

        return 0.0, 0.0, 0.0, 0  # win\_prob, avg\_dur\_min, avg\_rr, sample\_size

    sample\_size = len(recent)

    wins = sum(1 for t in recent if t.get("result") == "TP")

    win\_prob = wins / sample\_size \* 100.0

    # durations and rr

    durations = []

    rrs = []

    for t in recent:

        # ensure exit\_time/entry\_time are datetimes

        et = t.get("exit\_time")

        st = t.get("entry\_time")

        if isinstance(et, str):

            et = pd.to\_datetime(et)

        if isinstance(st, str):

            st = pd.to\_datetime(st)

        if et is not None and st is not None:

            dur\_min = (et - st).total\_seconds() / 60.0

            durations.append(dur\_min)

        # rr: (TP-entry)/(entry-SL) absolute

        try:

            rr = abs(t.get("TP",0) - t.get("entry",0)) / max(1e-9, abs(t.get("entry",0) - t.get("SL",0)))

            rrs.append(rr)

        except Exception:

            pass

    avg\_dur = round(sum(durations)/len(durations)) if durations else 0

    avg\_rr = round(sum(rrs)/len(rrs), 2) if rrs else 0.0

    return round(win\_prob,2), int(avg\_dur), avg\_rr, sample\_size

def confidence\_label(win\_prob, avg\_rr, sample\_size):

    """Simple label rules for Stage-2"""

    if sample\_size >= 20 and win\_prob >= 60 and avg\_rr >= 1.5:

        return "HIGH"

    if sample\_size >= 10 and (win\_prob >= 55 and avg\_rr >= 1.2):

        return "MEDIUM"

    return "LOW"

def build\_suggestions(symbols):

    suggestions = []

    now = datetime.utcnow().isoformat()

    for sym in symbols:

        for tf in TIMEFRAMES:

            df = load\_df(sym, tf)

            if df is None or len(df) < 300:

                # skip insufficient data

                continue

            for sname, sfunc in STRATEGIES.items():

                # run backtest to get historical trades (you may cache this in prod)

                bt = Backtester(df, sfunc)

                results = bt.run()

                trades = results.get("trades", [])

                # compute latest signal on last candle

                sig = sfunc(df, len(df)-1)

                if sig is None:

                    continue

                side, sl, tp = sig

                entry = float(df["close"].iloc[-1])

                # safety: ensure SL/TP orientation using simple checks (extra guard)

                if side.lower() == "long":

                    if not (sl < entry < tp):

                        # recompute/correct

                        if sl >= entry:

                            sl = entry - abs(sl-entry) - 1e-6

                        if tp <= entry:

                            tp = entry + abs(tp-entry) + 1e-6

                else:

                    if not (tp < entry < sl):

                        if sl <= entry:

                            sl = entry + abs(sl-entry) + 1e-6

                        if tp >= entry:

                            tp = entry - abs(tp-entry) - 1e-6

                # compute stats for side

                win\_prob, avg\_dur, avg\_rr, sample\_size = compute\_side\_stats(trades, side)

                conf = confidence\_label(win\_prob, avg\_rr, sample\_size)

                rr\_ratio\_live = round(abs(tp - entry) / max(1e-9, abs(entry - sl)), 2)

                suggestion = {

                    "generated\_at": now,

                    "symbol": sym,

                    "timeframe": tf,

                    "strategy": sname,

                    "side": side.upper(),

                    "entry": round(entry, 6),

                    "SL": round(sl, 6),

                    "TP": round(tp, 6),

                    "expected\_duration\_min": avg\_dur,

                    "win\_probability": win\_prob,

                    "avg\_rr": avg\_rr,

                    "rr\_ratio\_live": rr\_ratio\_live,

                    "confidence": conf,

                    "sample\_size": sample\_size

                }

                suggestions.append(suggestion)

    return suggestions

if \_\_name\_\_ == "\_\_main\_\_":

    # detect symbols (folders in data/)

    symbols = [d for d in os.listdir(DATA\_DIR) if os.path.isdir(os.path.join(DATA\_DIR, d))]

    # symbols folder names might be like 'BTCUSDT' so convert to 'BTC/USDT'

    symbols = [s.replace("USDT","/USDT") for s in symbols]

    suggestions = build\_suggestions(symbols)

    if not suggestions:

        print("No suggestions generated.")

    else:

        df = pd.DataFrame(suggestions)

        print(df)

        df.to\_csv(SUGGESTIONS\_CSV, index=False)

        # json for API/frontend

        with open(SUGGESTIONS\_JSON, "w") as f:

            f.write(pd.DataFrame(suggestions).to\_json(orient="records", date\_format="iso"))

        print(f"\nSaved -> {SUGGESTIONS\_CSV}, {SUGGESTIONS\_JSON}")

**output:** **trade\_suggestions\_stage2.csv**



Day 21-09-2025

Stage 3  
  
here the data from stage 2 is taken to processing   
so stage 2 excel sheet is input here  
the symbols should 1st run in stage 2 then to stage 3

Stage 3 is the pure form of filtered data

Stage 3 gives reliable output, but needs huge data  
that huge data we need various symbols at different interventions,

So here you can go with the task

Try doing this asap so to move ahed ‘

**📝 Data Collection Task — Trading Bot Project**

**🎯 Objective**

We need **historical OHLCV data (Open, High, Low, Close, Volume)** for multiple crypto symbols and timeframes. This data will feed into our backtester and strategy engine to generate trading signals.

**📌 Scope of Work**

1. **Symbols**  
   Collect for:
   * BTC/USDT
   * ETH/USDT
   * BNB/USDT
   * (later, we’ll expand to 10–20 symbols)
2. **Timeframes**
   * 1m, 5m, 15m, 1h, 4h, 12h, 1d  
     (at least last **1–2 years** for intraday, 3+ years for daily).
3. **Data Source**
   * Binance (preferred) → provides reliable OHLCV.
   * Use either:
     + **Binance API + Python script** (ccxt or binance library), OR
     + Manual export from Binance web charts (if API is blocked).
4. **Storage Format**
   * Save into data/ folder.
   * Folder structure:
   * data/
   * BTCUSDT/
   * 5m.csv
   * 15m.csv
   * 1h.csv
   * ...
   * ETHUSDT/
   * 5m.csv
   * 15m.csv
   * ...
   * Each CSV must have:
   * ts, open, high, low, close, volume
   * 2021-01-01 00:00:00, 29000, 29500, 28900, 29300, 120.5
   * Ensure **timestamps are UTC**.
5. **Compression**
   * Save files as .csv.gz (compressed). Pandas can read them directly → reduces storage by ~70%.
6. **Update Process**
   * Write/download script that can **append new candles daily** (so we don’t re-download everything).
   * Keep data up-to-date.

**✅ Deliverables**

1. Organized data/ folder with **symbols × timeframes** CSV files.
2. Each CSV has **consistent format** (ts, open, high, low, close, volume).
3. Compressed files (.csv.gz).
4. (Optional, bonus) A **Python script** to auto-download & update from Binance API.

**🔮 Why This Matters**

* Without reliable historical data, our backtests and signal generation are meaningless.
* Stage 2 and Stage 3 of our pipeline require enough **trade samples** (50–100+ trades) to calculate win probability, RR ratio, and confidence levels.
* More symbols + timeframes = richer strategy evaluation.

DATE: 30-09-2025

STAGE-3 with zip files and output

This code needs data file,

The data file contains the data of various timezones for various signals

We assumed 6 signals with diff time zones (in code)  
the output is: (trade\_suggestions\_stage2.csv)



This output is of:

**🎯 Strategy Name: EMA + RSI Confluence Strategy**

This strategy combines **trend direction** (from EMAs) and **momentum strength** (from RSI) to find **high-probability entries**.

**🧠 Core Idea**

We only trade **when trend and momentum agree**.

* EMA → tells us the **direction** of the market (up or down).
* RSI → tells us the **strength** of the move (momentum).

When both say the same thing, we take the trade.

**⚙️ Step-by-Step Logic**

**1. Identify Trend using EMAs**

We use **two EMAs**:

* **Fast EMA (short period)** — reacts quickly
* **Slow EMA (long period)** — reacts slowly

👉 If **Fast EMA > Slow EMA**, trend is **up (bullish)**  
👉 If **Fast EMA < Slow EMA**, trend is **down (bearish)**

This tells us whether to look for **LONG** or **SHORT** opportunities.

**2. Confirm Momentum using RSI**

We use RSI (Relative Strength Index):

* RSI > 50 → buyers gaining control
* RSI < 50 → sellers gaining control

So we only take:

* **LONG trade** if RSI crosses above 50 (momentum rising)
* **SHORT trade** if RSI crosses below 50 (momentum falling)

**3. Entry Signal**

We wait for a candle that **closes in the direction of the trend**:

* LONG: Candle closes **above both EMAs**
* SHORT: Candle closes **below both EMAs**

✅ This ensures we don’t enter too early.

**4. Stop Loss (SL) and Take Profit (TP)**

We calculate:

* **SL** = Just below recent swing low (for LONG) or above swing high (for SHORT)
* **TP** = Based on a fixed **Risk:Reward = 1.8x**  
  → If you risk $1, you aim to gain $1.8

**5. Probability & Confidence**

From past data (backtesting):

* We calculate **win probability** (e.g. 55%)
* Show **confidence** (LOW/MED/HIGH) depending on how often the setup worked before

**📘 Example**

Let’s say BTC/USDT 5m chart:

* Fast EMA (20) is above Slow EMA (50) ✅ → Uptrend
* RSI just crossed **above 50** ✅ → Buyers gaining control
* Candle closes above both EMAs ✅ → Confirmation

👉 We enter a **LONG trade at 27,250**

* **Stop Loss** = 27,100
* **Take Profit** = 27,450
* **RR** = 1.8
* **Win Probability** = 55%

That means historically, 55% of similar trades have hit TP.

**🧩 Summary**

✅ **Trade only when trend + momentum match**  
✅ **Filter weak signals**  
✅ **Fixed risk/reward & confidence scoring**  
✅ Works across **multiple timeframes (5m to 12h)**

Code for zip (complete run\_stage2.py)

# run\_stage2.py

import os

import math

import pandas as pd

from datetime import datetime

from backtester import Backtester

from ema\_rsi\_stage2 import ema\_rsi\_strategy

DATA\_DIR = "data"

TIMEFRAMES = ["5m", "15m", "1h", "12h"]

SUGGESTIONS\_CSV = "trade\_suggestions\_stage2.csv"

SUGGESTIONS\_JSON = "trade\_suggestions\_stage2.json"

# Register strategies here (name -> function)

STRATEGIES = {

    "ema\_rsi": ema\_rsi\_strategy,

    # add more strategies here in future

}

RECENT\_TRADE\_WINDOW = 50   # use last N trades for statistics (Stage-2 uses recent history)

def load\_df(symbol: str, timeframe: str):

    """Load CSV or ZIP file for given symbol/timeframe"""

    base\_path = os.path.join(DATA\_DIR, symbol.replace("/", ""))

    csv\_path = os.path.join(base\_path, f"{timeframe}.csv")

    zip\_path = os.path.join(base\_path, f"{timeframe}.zip")

    if os.path.exists(csv\_path):

        df = pd.read\_csv(csv\_path)

    elif os.path.exists(zip\_path):

        df = pd.read\_csv(zip\_path, compression='zip')

    else:

        print(f"[WARN] No data found for {symbol} {timeframe} in {base\_path}")

        return None

    # Clean + format

    df["ts"] = pd.to\_datetime(df["ts"])

    df[["open", "high", "low", "close", "volume"]] = df[["open", "high", "low", "close", "volume"]].astype(float)

    return df

def sample\_recent\_trades(trades, side, n=RECENT\_TRADE\_WINDOW):

    # filter by side and take last n trades by entry\_time

    trades\_side = [t for t in trades if t.get("side","").lower() == side.lower()]

    trades\_side\_sorted = sorted(trades\_side, key=lambda x: x["entry\_time"])  # oldest->newest

    return trades\_side\_sorted[-n:]

def compute\_side\_stats(trades, side):

    """Compute win\_prob (%), avg\_duration (minutes), avg\_rr from recent trades of same side"""

    recent = sample\_recent\_trades(trades, side, RECENT\_TRADE\_WINDOW)

    if not recent:

        return 0.0, 0.0, 0.0, 0  # win\_prob, avg\_dur\_min, avg\_rr, sample\_size

    sample\_size = len(recent)

    wins = sum(1 for t in recent if t.get("result") == "TP")

    win\_prob = wins / sample\_size \* 100.0

    # durations and rr

    durations = []

    rrs = []

    for t in recent:

        # ensure exit\_time/entry\_time are datetimes

        et = t.get("exit\_time")

        st = t.get("entry\_time")

        if isinstance(et, str):

            et = pd.to\_datetime(et)

        if isinstance(st, str):

            st = pd.to\_datetime(st)

        if et is not None and st is not None:

            dur\_min = (et - st).total\_seconds() / 60.0

            durations.append(dur\_min)

        # rr: (TP-entry)/(entry-SL) absolute

        try:

            rr = abs(t.get("TP",0) - t.get("entry",0)) / max(1e-9, abs(t.get("entry",0) - t.get("SL",0)))

            rrs.append(rr)

        except Exception:

            pass

    avg\_dur = round(sum(durations)/len(durations)) if durations else 0

    avg\_rr = round(sum(rrs)/len(rrs), 2) if rrs else 0.0

    return round(win\_prob,2), int(avg\_dur), avg\_rr, sample\_size

def confidence\_label(win\_prob, avg\_rr, sample\_size):

    """Simple label rules for Stage-2"""

    if sample\_size >= 20 and win\_prob >= 60 and avg\_rr >= 1.5:

        return "HIGH"

    if sample\_size >= 10 and (win\_prob >= 55 and avg\_rr >= 1.2):

        return "MEDIUM"

    return "LOW"

def build\_suggestions(symbols):

    suggestions = []

    now = datetime.utcnow().isoformat()

    for sym in symbols:

        for tf in TIMEFRAMES:

            df = load\_df(sym, tf)

            if df is None or len(df) < 300:

                # skip insufficient data

                continue

            for sname, sfunc in STRATEGIES.items():

                # run backtest to get historical trades (you may cache this in prod)

                bt = Backtester(df, sfunc)

                results = bt.run()

                trades = results.get("trades", [])

                # compute latest signal on last candle

                sig = sfunc(df, len(df)-1)

                if sig is None:

                    continue

                side, sl, tp = sig

                entry = float(df["close"].iloc[-1])

                # safety: ensure SL/TP orientation using simple checks (extra guard)

                if side.lower() == "long":

                    if not (sl < entry < tp):

                        # recompute/correct

                        if sl >= entry:

                            sl = entry - abs(sl-entry) - 1e-6

                        if tp <= entry:

                            tp = entry + abs(tp-entry) + 1e-6

                else:

                    if not (tp < entry < sl):

                        if sl <= entry:

                            sl = entry + abs(sl-entry) + 1e-6

                        if tp >= entry:

                            tp = entry - abs(tp-entry) - 1e-6

                # compute stats for side

                win\_prob, avg\_dur, avg\_rr, sample\_size = compute\_side\_stats(trades, side)

                conf = confidence\_label(win\_prob, avg\_rr, sample\_size)

                rr\_ratio\_live = round(abs(tp - entry) / max(1e-9, abs(entry - sl)), 2)

                suggestion = {

                    "generated\_at": now,

                    "symbol": sym,

                    "timeframe": tf,

                    "strategy": sname,

                    "side": side.upper(),

                    "entry": round(entry, 6),

                    "SL": round(sl, 6),

                    "TP": round(tp, 6),

                    "expected\_duration\_min": avg\_dur,

                    "win\_probability": win\_prob,

                    "avg\_rr": avg\_rr,

                    "rr\_ratio\_live": rr\_ratio\_live,

                    "confidence": conf,

                    "sample\_size": sample\_size

                }

                suggestions.append(suggestion)

    return suggestions

if \_\_name\_\_ == "\_\_main\_\_":

    # detect symbols (folders in data/)

    symbols = [d for d in os.listdir(DATA\_DIR) if os.path.isdir(os.path.join(DATA\_DIR, d))]

    # symbols folder names might be like 'BTCUSDT' so convert to 'BTC/USDT'

    symbols = [s.replace("USDT","/USDT") for s in symbols]

    suggestions = build\_suggestions(symbols)

    if not suggestions:

        print("No suggestions generated.")

    else:

        df = pd.DataFrame(suggestions)

        print(df)

        df.to\_csv(SUGGESTIONS\_CSV, index=False)

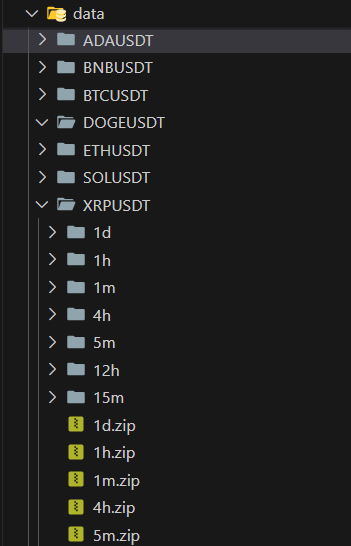
        # json for API/frontend

        with open(SUGGESTIONS\_JSON, "w") as f:

            f.write(pd.DataFrame(suggestions).to\_json(orient="records", date\_format="iso"))

        print(f"\nSaved -> {SUGGESTIONS\_CSV}, {SUGGESTIONS\_JSON}")

the data folder is as follows :



The data is automatically called or downloaded by (autoload.py)

import ccxt

import pandas as pd

import os

import logging

from datetime import datetime, timedelta

import time

import requests

import zipfile

from io import BytesIO

import tempfile

# Set up logging

logging.basicConfig(

    level=logging.INFO,

    format='%(asctime)s - %(levelname)s - %(message)s',

    handlers=[

        logging.FileHandler('download\_log.txt'),

        logging.StreamHandler()

    ]

)

logger = logging.getLogger(\_\_name\_\_)

# Configuration

SYMBOLS = ['BTC/USDT', 'ETH/USDT', 'BNB/USDT', 'ADA/USDT', 'SOL/USDT', 'XRP/USDT', 'DOGE/USDT']  # 7 symbols

TIMEFRAMES = ['1m', '5m', '15m', '1h', '4h', '12h', '1d']

DATA\_DIR = 'data'

LIMIT = 1000  # Binance API limit per request (fixed max; optimal for all timeframes)

BASE\_URL = 'https://data.binance.vision/data/spot/monthly/klines'

# Timeframe to retention period mapping (in days)

RETENTION\_DAYS = {

    tf: 365 \* 2 if tf != '1d' else 365 \* 5 for tf in TIMEFRAMES

}

# For updates, use API for last 30 days to bridge to ZIP historical

HISTORICAL\_CUTOFF\_DAYS = 30

def create\_symbol\_dir(symbol):

    """Create directory for symbol if it doesn't exist."""

    symbol\_name = symbol.replace('/', '')

    dir\_path = os.path.join(DATA\_DIR, symbol\_name)

    os.makedirs(dir\_path, exist\_ok=True)

    return dir\_path

def get\_start\_date(timeframe):

    """Get start date based on retention period."""

    retention\_days = RETENTION\_DAYS[timeframe]

    start\_date = datetime.now() - timedelta(days=retention\_days)

    return start\_date

def load\_existing\_data(file\_path):

    """Load CSV from ZIP and return the last timestamp."""

    if os.path.exists(file\_path):

        try:

            with zipfile.ZipFile(file\_path, 'r') as zf:

                csv\_name = [name for name in zf.namelist() if name.endswith('.csv')][0]

                with zf.open(csv\_name) as csv\_file:

                    df = pd.read\_csv(csv\_file)

                    if not df.empty and 'ts' in df.columns:

                        last\_ts = pd.to\_datetime(df['ts'].iloc[-1])

                        return last\_ts

        except Exception as e:

            logger.warning(f"Error loading existing data from {file\_path}: {e}")

    return None

def download\_monthly\_zip(symbol, timeframe, year, month):

    """Download monthly ZIP from Binance data vision and save as-is."""

    symbol\_name = symbol.replace('/', '')

    zip\_filename = f"{symbol\_name}-{timeframe}-{year}-{month:02d}.zip"

    url = f"{BASE\_URL}/{symbol\_name}/{timeframe}/{zip\_filename}"

    try:

        response = requests.get(url, stream=True)

        response.raise\_for\_status()

        # Save raw ZIP to disk (in timeframe subdir)

        tf\_dir = os.path.join(DATA\_DIR, symbol\_name, timeframe)

        os.makedirs(tf\_dir, exist\_ok=True)

        local\_zip\_path = os.path.join(tf\_dir, f"{year}-{month:02d}.zip")

        with open(local\_zip\_path, 'wb') as f:

            for chunk in response.iter\_content(chunk\_size=8192):

                f.write(chunk)

        logger.info(f"Downloaded raw ZIP: {local\_zip\_path}")

        return local\_zip\_path  # Return path for potential use

    except requests.exceptions.HTTPError as e:

        if e.response.status\_code == 404:

            logger.info(f"ZIP not found (normal for future months): {zip\_filename}")

        else:

            logger.warning(f"Error downloading {zip\_filename}: {e}")

    except Exception as e:

        logger.warning(f"Error saving {zip\_filename}: {e}")

    return None

def fetch\_historical\_zip\_data(symbol, timeframe, start\_date):

    """Download historical monthly ZIPs (saved as-is)."""

    cutoff\_date = datetime.now() - timedelta(days=HISTORICAL\_CUTOFF\_DAYS)

    year = start\_date.year

    month = start\_date.month

    current\_date = start\_date.replace(day=1)

    downloaded\_zips = []

    while current\_date < cutoff\_date:

        zip\_path = download\_monthly\_zip(symbol, timeframe, year, month)

        if zip\_path:

            downloaded\_zips.append(zip\_path)

        # Next month

        if month == 12:

            month = 1

            year += 1

        else:

            month += 1

        current\_date = current\_date.replace(year=year, month=month)

    logger.info(f"Downloaded {len(downloaded\_zips)} historical ZIPs for {symbol} {timeframe}")

    return downloaded\_zips  # List of ZIP paths

def fetch\_ohlcv\_api(exchange, symbol, timeframe, since=None):

    """Fetch recent OHLCV data from Binance API, handling chunking."""

    all\_data = []

    while True:

        try:

            ohlcv = exchange.fetch\_ohlcv(symbol, timeframe, since=since, limit=LIMIT)

            if not ohlcv:

                break

            all\_data.extend(ohlcv)

            logger.info(f"Fetched {len(ohlcv)} recent candles for {symbol} {timeframe}, up to {datetime.fromtimestamp(ohlcv[-1][0] / 1000)}")

            since = ohlcv[-1][0] + 1

            if len(ohlcv) < LIMIT:

                break

            time.sleep(0.1)  # Rate limit respect

        except Exception as e:

            logger.error(f"Error fetching recent data for {symbol} {timeframe}: {e}")

            break

    return all\_data

def process\_recent\_data(all\_data):

    """Process raw API OHLCV into DataFrame."""

    if not all\_data:

        return pd.DataFrame()

    df\_new = pd.DataFrame(all\_data, columns=['timestamp', 'open', 'high', 'low', 'close', 'volume'])

    df\_new['ts'] = pd.to\_datetime(df\_new['timestamp'], unit='ms').dt.strftime('%Y-%m-%d %H:%M:%S')

    df\_new = df\_new[['ts', 'open', 'high', 'low', 'close', 'volume']]

    df\_new[['open', 'high', 'low', 'close', 'volume']] = df\_new[['open', 'high', 'low', 'close', 'volume']].astype(float)

    df\_new.drop\_duplicates(subset=['ts'], keep='last', inplace=True)

    df\_new.sort\_values('ts', inplace=True)

    df\_new.reset\_index(drop=True, inplace=True)

    return df\_new

def extract\_and\_merge\_zips(zip\_paths):

    """Extract and merge DataFrames from list of ZIP paths."""

    all\_data = []

    for zip\_path in zip\_paths:

        try:

            with zipfile.ZipFile(zip\_path, 'r') as zf:

                csv\_name = [name for name in zf.namelist() if name.endswith('.csv')][0]

                with zf.open(csv\_name) as csv\_file:

                    df\_month = pd.read\_csv(csv\_file, header=None)

                    df\_month.columns = ['open\_time', 'open', 'high', 'low', 'close', 'volume',

                                        'close\_time', 'quote\_asset\_volume', 'number\_of\_trades',

                                        'taker\_buy\_base\_asset\_volume', 'taker\_buy\_quote\_asset\_volume', 'ignore']

                    df\_month = df\_month[['open\_time', 'open', 'high', 'low', 'close', 'volume']]

                    df\_month['ts'] = pd.to\_datetime(df\_month['open\_time'], unit='ms').dt.strftime('%Y-%m-%d %H:%M:%S')

                    df\_month = df\_month[['ts', 'open', 'high', 'low', 'close', 'volume']]

                    df\_month[['open', 'high', 'low', 'close', 'volume']] = df\_month[['open', 'high', 'low', 'close', 'volume']].astype(float)

                    df\_month.drop\_duplicates(subset=['ts'], keep='last', inplace=True)

                    df\_month.sort\_values('ts', inplace=True)

                    all\_data.append(df\_month)

        except Exception as e:

            logger.warning(f"Error extracting {zip\_path}: {e}")

    if all\_data:

        df\_historical = pd.concat(all\_data, ignore\_index=True)

        df\_historical.drop\_duplicates(subset=['ts'], keep='last', inplace=True)

        df\_historical.sort\_values('ts', inplace=True)

        df\_historical.reset\_index(drop=True, inplace=True)

        return df\_historical

    return pd.DataFrame()

def merge\_and\_save(df\_historical, df\_recent, file\_path):

    """Merge historical + recent data, save uncompressed CSV inside a ZIP (compressed storage, text content)."""

    if df\_historical.empty and df\_recent.empty:

        return

    df\_combined = pd.concat([df\_historical, df\_recent], ignore\_index=True) if not df\_historical.empty else df\_recent

    if not df\_historical.empty and not df\_recent.empty:

        # Filter recent to avoid overlap

        hist\_last = pd.to\_datetime(df\_historical['ts'].iloc[-1])

        df\_recent\_filtered = df\_recent[df\_recent['ts'] > hist\_last.strftime('%Y-%m-%d %H:%M:%S')]

        df\_combined = pd.concat([df\_historical, df\_recent\_filtered], ignore\_index=True)

    df\_combined.drop\_duplicates(subset=['ts'], keep='last', inplace=True)

    df\_combined.sort\_values('ts', inplace=True)

    df\_combined.reset\_index(drop=True, inplace=True)

    parent\_dir = os.path.dirname(file\_path)

    os.makedirs(parent\_dir, exist\_ok=True)

    # Save uncompressed CSV to temp

    with tempfile.NamedTemporaryFile(mode='w', suffix='.csv', delete=False, newline='') as temp\_csv:

        df\_combined.to\_csv(temp\_csv.name, index=False)

        temp\_csv\_path = temp\_csv.name

    # Create ZIP and add the CSV

    with zipfile.ZipFile(file\_path, 'w', zipfile.ZIP\_DEFLATED) as zf:

        zf.write(temp\_csv\_path, 'full\_history.csv')  # Add as 'full\_history.csv' inside ZIP

    # Clean up temp

    os.unlink(temp\_csv\_path)

    logger.info(f"Saved {len(df\_combined)} total rows in ZIP: {file\_path} (uncompressed CSV inside, ~70% size reduction via ZIP)")

def main():

    """Main function: Download raw ZIPs for historical + API for recent, merge into ZIP with text CSV."""

    os.makedirs(DATA\_DIR, exist\_ok=True)

    exchange = ccxt.binance({'enableRateLimit': True})

    for symbol in SYMBOLS:

        symbol\_dir = create\_symbol\_dir(symbol)

        logger.info(f"Processing symbol: {symbol}")

        for timeframe in TIMEFRAMES:

            file\_path = os.path.join(symbol\_dir, f"{timeframe}.zip")  # Final merged ZIP

            existing\_last\_ts = load\_existing\_data(file\_path)

            start\_date = get\_start\_date(timeframe)

            cutoff\_date = datetime.now() - timedelta(days=HISTORICAL\_CUTOFF\_DAYS)

            since\_for\_api = int(cutoff\_date.timestamp() \* 1000)

            # Download historical raw ZIPs

            logger.info(f"Downloading historical ZIPs for {symbol} {timeframe} from {start\_date} to {cutoff\_date}")

            historical\_zips = fetch\_historical\_zip\_data(symbol, timeframe, start\_date)

            # Extract/merge historical

            df\_historical = extract\_and\_merge\_zips(historical\_zips)

            # Fetch recent via API from cutoff to now

            logger.info(f"Fetching recent API data for {symbol} {timeframe} from {cutoff\_date}")

            all\_recent = fetch\_ohlcv\_api(exchange, symbol, timeframe, since=since\_for\_api)

            df\_recent = process\_recent\_data(all\_recent)

            # If existing data, merge with it only if newer

            if existing\_last\_ts:

                logger.info(f"Existing data up to {existing\_last\_ts}; merging if needed")

                try:

                    with zipfile.ZipFile(file\_path, 'r') as zf:

                        csv\_name = [name for name in zf.namelist() if name.endswith('.csv')][0]

                        with zf.open(csv\_name) as csv\_file:

                            existing\_df = pd.read\_csv(csv\_file)

                            df\_combined = pd.concat([existing\_df, df\_historical, df\_recent], ignore\_index=True)

                            df\_combined.drop\_duplicates(subset=['ts'], keep='last', inplace=True)

                            df\_combined.sort\_values('ts', inplace=True)

                            df\_combined.reset\_index(drop=True, inplace=True)

                            df\_historical = df\_combined  # Reuse for save

                except Exception as e:

                    logger.warning(f"Error merging existing: {e}")

            merge\_and\_save(df\_historical, df\_recent, file\_path)

    logger.info("Data download completed.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

the output wil reflect in the **data folder**

