**File:** **autoload.py**

**This will:**

**It automatically builds and updates complete historical candle data for multiple crypto pairs and timeframes, using Binance’s monthly files + API, then saves everything neatly compressed.**

* **Symbols & Timeframes**: Works on 7 pairs (BTC/USDT, ETH/USDT, etc.) across multiple timeframes (1m → 1d).
* **Retention**: Keeps up to 2 years of data (5 years for daily).
* **Historical Data**: Downloads **monthly ZIP files** from Binance Data Vision (older than 30 days).
* **Recent Data**: Uses Binance API (via ccxt) to fetch the last 30 days.
* **Merging**: Combines historical ZIP data + recent API data → removes duplicates → sorts by timestamp.
* **Saving**: Stores result in data/<symbol>/<timeframe>.zip containing full\_history.csv.
* **Raw ZIPs**: Also saves untouched monthly ZIPs in data/<symbol>/<timeframe>/<YYYY-MM>.zip.
* **Logging**: Records progress/errors to console and download\_log.txt.
* **Output**: Final dataset = one ZIP per symbol & timeframe with full OHLCV history (open, high, low, close, volume).

**Code: (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()

**File: agent.py**

**This will:**

* **Data sources**:
  + Uses **only Binance API** (ccxt) to fetch candles.
* **Retention**:
  + Keeps **only recent days** per timeframe (e.g. 30 days for 5m, 60 for 15m, 365 for 12h).
* **Storage format**:
  + Saves each timeframe as a **CSV** (not zipped).
  + Creates an extra merged\_multi\_tf.csv aligning multiple timeframes into one file.
* **Goal**: Maintain a **short rolling window** of recent data, optimized for daily updates & multi-timeframe analysis.

**Differences between 1st (autoload.py) and 2nd (agent.py)**

| **Aspect** | **First Code (autoload.py)** | **Second Code (agent.py)** |
| --- | --- | --- |
| Source | API + Binance archives | API only |
| Retention | Multi-year (2–5 yrs) | Few days–1 year (rolling) |
| Storage | ZIPs with CSV inside | Plain CSVs |
| Extra Features | Historical reconstruction | Multi-timeframe merge |
| Use Case | Research / backtesting (big history) | Daily live updates / trading signals |

**Code (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()} ===")

**File: backtester.py**

**This will:**

**This code backtests a trading strategy by simulating entries/exits with SL & TP, updating balance, and reporting performance stats.**

* It simulates trading a strategy on historical OHLCV data.
* Starts with an **initial balance** (default 10,000).
* Each trade risks a fixed fraction of balance (risk = 0.02 → 2%).
* **Strategy function** decides when to open a trade and gives (side, SL, TP).
* When a trade is open:
* If price hits **SL** → balance decreases by the fixed risk amount.
* If price hits **TP** → balance increases by a “reward amount” (based on TP/SL distance).
* Records each trade in trade\_history.
* Builds an **equity curve** (balance over time).
* Tracks **take-profit hits** (tp\_hits) and **stop-loss hits** (sl\_hits).
* Calculates **win rate** = % of trades hitting TP.
* Tracks **max drawdown** (peak balance – lowest balance after).
* At the end, returns a summary:
* Final balance
* Net profit
* Trade list
* Equity curve
* TP/SL stats
* Win rate
* Max drawdown

**Code (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

**File: ema\_rsi.py**

**This will**

* Basic **EMA + RSI + ADX + ATR** strategy.
* Trades only when **ADX > 20** (trend filter).
* **Long entry**: EMA50 > EMA200 and RSI < 70.
  + SL = price – 1 × ATR.
  + TP = price + 2 × ATR.
* **Short entry**: EMA50 < EMA200 and RSI > 30.
  + SL = price + 1 × ATR.
  + TP = price – 2 × ATR.
* Simple implementation, no extra checks.
* Risk–reward ratio = **1:2**.

Code (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

**File: ema\_rsi\_stage2.py**

**This will**

* Adds **helper function** ensure\_sl\_tp\_orientation() → guarantees SL/TP are always on the correct side of entry.
* Computes indicators with explicit window params and **fills NaNs** for ADX with 0.
* **Extra safety checks**: skips trading if i < 200 or indicators are NaN.
* Makes **ADX threshold configurable** (adx\_threshold=20 by default).
* **Long entry**: EMA50 > EMA200 and RSI < 75 (slightly looser).
  + SL = price – 1 × ATR.
  + TP = price + 1.8 × ATR (tighter than Stage-1’s 2.0).
* **Short entry**: EMA50 < EMA200 and RSI > 25 (slightly looser).
  + SL = price + 1 × ATR.
  + TP = price – 1.8 × ATR.
* Casts all indicator values to float to avoid dtype issues.
* Risk–reward ratio = **~1:1.8**.

**Difference between (ema\_rsi.py) and (ema\_rsi\_stage2.py)**

| **Feature / Rule** | **Stage-1 (ema\_rsi.py)** | **Stage-2 (ema\_rsi\_stage2.py)** |
| --- | --- | --- |
| **Indicators Used** | EMA50, EMA200, RSI(14), ATR(14), ADX(14) | Same indicators, but with explicit window= and ADX NaNs filled with 0 |
| **ADX Threshold** | Hard-coded > 20 | Configurable (adx\_threshold=20 default) |
| **RSI Condition (Long)** | RSI < 70 | RSI < 75 (slightly more lenient) |
| **RSI Condition (Short)** | RSI > 30 | RSI > 25 (slightly more lenient) |
| **Long SL/TP** | SL = price – 1×ATR TP = price + 2×ATR | SL = price – 1×ATR TP = price + 1.8×ATR |
| **Short SL/TP** | SL = price + 1×ATR TP = price – 2×ATR | SL = price + 1×ATR TP = price – 1.8×ATR |
| **Risk-Reward Ratio** | 1 : 2.0 | 1 : 1.8 |
| **Safety Checks** | None (can return NaN signals early) | Skips signals if i < 200 or NaNs present |
| **SL/TP Validation** | None (could place wrong side of entry) | Uses ensure\_sl\_tp\_orientation() to auto-correct |
| **Type Casting** | Implicit (keeps pandas dtype) | Explicit float() casting for indicators |
| **Robustness** | Simpler, more “raw” | Safer, more production-ready |

**Code: (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

**File: merge, run files, startergy test files left**