```
from __future__ import annotations
import os, json, math, re, sys, time
from collections import defaultdict
from datetime import datetime, date, timedelta
from typing import Dict, Any, List, Tuple, Optional
import requests
from dotenv import load_dotenv
import pandas as pd

try:
    from json_repair import repair_json as _repair_json_lib
except Exception:
    _repair_json_lib = None

load_dotenv()
OLLAMA_URL     = os.getenv("OLLAMA_URL", "http://localhost:11434/api/chat")
MODEL_NAME     = os.getenv("MODEL_NAME", "qwen3:8b")
SEED           = int(os.getenv("SEED", "42"))
TEMPERATURE    = float(os.getenv("TEMPERATURE", "0.2"))
TOP_P          = float(os.getenv("TOP_P", "0.9"))
RUN_MODE       = os.getenv("RUN_MODE", "test").strip().lower()  # "test" | "prod"
XAI_OUTPUT_XLS = os.getenv("XAI_OUTPUT_PATH", "./xai_outputs.xlsx")
XAI_DATA_PATH  = os.getenv("XAI_DATA_PATH", "./Book1.xlsx")
XAI_MAX_DRIVERS= int(os.getenv("XAI_MAX_DRIVERS", "4"))  # rule-based fallback top-N
XAI_CONTEXT_JSON = os.getenv("XAI_CONTEXT_JSON", "").strip()

# Web context toggles (news)
ENABLE_WEB_CONTEXT = os.getenv("ENABLE_WEB_CONTEXT", "0").strip() == "1"
NEWS_BACKEND = os.getenv("NEWS_BACKEND", "gdelt").strip().lower()  # prioritize GDELT
NEWSAPI_KEY = os.getenv("NEWSAPI_KEY", "")
NEWS_MAX_PER_DRIVER = int(os.getenv("NEWS_MAX_PER_DRIVER", "3"))

# Metrics / token estimation
XAI_SAVE_METRICS = os.getenv("XAI_SAVE_METRICS", "1").strip() == "1"
XAI_METRICS_CSV = os.getenv("XAI_METRICS_CSV", "./xai_llm_metrics.csv")
XAI_OVERALL_MODE = os.getenv("XAI_OVERALL_MODE", "hybrid").strip().lower()  # 'hybrid' | 'llm' | 'rb'

# Optional tokenizer (best-effort). If unavailable, fall back to char/4 heuristic.
try:
    import tiktoken as _tiktoken
    _tk_enc = _tiktoken.get_encoding("cl100k_base")
except Exception:
    _tk_enc = None

def _estimate_tokens(text: str) -> int:
    if not text:
        return 0
    try:
        if _tk_enc is not None:
            return len(_tk_enc.encode(text))
    except Exception:
        pass
    # fallback heuristic
    return max(1, int(len(text) / 4))

# Global metrics log
LLM_METRICS_LOG: List[Dict[str, Any]] = []
LAST_CALL_METRICS: Dict[str, Any] = {}

try:
    _df_boot = pd.read_excel(XAI_DATA_PATH, engine="openpyxl")
    _weekly_col_boot = next((c for c in _df_boot.columns if c.strip().lower() in ("weekly_json_drivers","weekly_drivers")), None)
    WEEKLY_JSON_FLAT: Dict[str, Any] = json.loads(_df_boot[_weekly_col_boot][0]) if _weekly_col_boot else {}
except Exception:
    WEEKLY_JSON_FLAT = {}

# ---------------- AGGREGATION ------------------------
def yyyymm(date_str: str) -> str:
    dt = datetime.strptime(date_str, "%Y-%m-%d").date()
    return f"{dt.year:04d}-{dt.month:02d}"

def month_first_day(ym: str) -> str:
    y, m = ym.split("-")
    return f"{y}-{m}-01"

def next_n_months(start_ym: str, n: int) -> List[str]:
    y, m = map(int, start_ym.split("-"))
    out = []
    for i in range(n):
        yy = y + (m + i - 1) // 12
        mm = (m + i - 1) % 12 + 1
        out.append(f"{yy:04d}-{mm:02d}")
    return out

def aggregate_weekly_flat_to_monthly_mean(weekly_flat: Dict[str, Any], out_months: int = 4):
    rows = [weekly_flat[k] for k in sorted(weekly_flat.keys(), key=lambda x:int(x))]
    if not rows:
        raise ValueError("Weekly JSON is empty")

    min_ym = min(yyyymm(r["Date"]) for r in rows)
    months_horizon = next_n_months(min_ym, out_months)

    vals = defaultdict(list)  # (group, ym) -> [values]
    groups = set()
    for r in rows:
        g = r["Group"]; ym = yyyymm(r["Date"])
        groups.add(g)
        if ym in months_horizon:
            vals[(g, ym)].append(float(r["grp_contri"]))

    groups_sorted = sorted(groups)
    monthly_flat = {}
    idx = 0
    for g in groups_sorted:
        for ym in months_horizon:
            vlist = vals.get((g, ym), [])
            avg = sum(vlist)/len(vlist) if vlist else 0.0
            monthly_flat[str(idx)] = {
                "Group": g,
                "Date": month_first_day(ym),
                "Avg_grp_contri": round(avg, 6)
            }
            idx += 1

    return monthly_flat, groups_sorted, months_horizon

# ------------------ QUANTILE POLICY -------------
def percentiles(sorted_vals: List[float], qs: List[float]) -> List[float]:
    if not sorted_vals:
        return [0.0 for _ in qs]
    n = len(sorted_vals)
    out = []
    for q in qs:
        q = min(max(q, 0.0), 1.0)
        idx = int(round(q * (n - 1)))
        out.append(sorted_vals[idx])
```

```
    return out

def build_label_policy_from_monthly(monthly_flat: Dict[str, Any]):
    mags = [abs(float(rec["Avg_grp_contri"])) for rec in monthly_flat.values()]
    mags_sorted = sorted(mags)
    p50, p80 = percentiles(mags_sorted, [0.50, 0.80])
    return {
        "slightly":      {"lt": p50},
        "moderately":    {"ge": p50, "lt": p80},
        "significantly": {"ge": p80}
    }

# ------------- FACTS PREPARATION FOR LLM ------------------
def month_label(ym: str) -> str:
    return datetime.strptime(ym+"-01", "%Y-%m-%d").strftime("%b %Y")

def build_group_trends(payload_months: List[Dict[str, Any]]) -> Dict[str, Any]:
    by_group = defaultdict(list)
    for m in payload_months:
        for g in m["groups"]:
            by_group[g["name"]].append(g["value"])
    out = {}
    for g, vals in by_group.items():
        sign_changes = sum(1 for i in range(1, len(vals)) if (vals[i] > 0) != (vals[i-1] > 0))
        out[g] = {
            "mean": round(sum(vals)/len(vals), 3),
            "share_positive": round(sum(1 for v in vals if v > 0)/len(vals), 2),
            "sign_changes": sign_changes
        }
    return out

def build_month_balance(payload_months: List[Dict[str, Any]]) -> Dict[str, Any]:
    out = {}
    for m in payload_months:
        ym = m["month"]
        vals = [abs(g["value"]) for g in m["groups"]]
        sum_abs = sum(vals)
        net = abs(m["net_effect"])
        ratio = net / sum_abs if sum_abs else 0.0
        if ratio < 0.30: label = "mostly offsetting"
        elif ratio < 0.60: label = "mixed"
        else: label = "concentrated"
        out[ym] = {"sum_abs": round(sum_abs, 3), "balance_ratio": round(ratio, 3), "balance_label": label}
    return out

def build_prev_map(months_list: List[Dict[str, Any]]) -> Dict[str, Dict[str, float]]:
    prev_map = {}
    for i, m in enumerate(months_list):
        if i == 0:
            prev_map[m["month"]] = {}
        else:
            prev = months_list[i-1]
            prev_map[m["month"]] = {g["name"]: float(g["value"]) for g in prev["groups"]}
    return prev_map

# ---------- News helpers (mainly GDELT) ----------
import urllib.parse

def _month_bounds(ym: str) -> tuple[str, str]:
    dt0 = datetime.strptime(ym + "-01", "%Y-%m-%d").date()
    if dt0.month == 12:
        dt1 = date(dt0.year + 1, 1, 1)
    else:
        dt1 = date(dt0.year, dt0.month + 1, 1)
    return dt0.isoformat(), dt1.isoformat()

# Secondary/static synonyms (fallback only)
_DRIVER_SYNONYMS = {
    "Crude_Oil_WTI_Price": ["WTI crude", "oil price", "West Texas Intermediate"],
    "Gasoline_Stock_Level": ["gasoline inventories", "EIA gasoline stocks", "gasoline stockpiles"],
    "Product_Supplied_Demand": ["product supplied", "implied demand", "U.S. gasoline demand"],
    "Refinery_Utilization_Rate": ["refinery utilization", "refinery runs", "refinery throughput"],
}

def _simple_expansion_terms(driver: str) -> list[str]:
    base = driver.replace("_", " ")
    parts = list({base, base.lower(), base.title()})
    parts += _DRIVER_SYNONYMS.get(driver, [])
    # minor variants
    parts += [base.replace("gasoline","gas") if "gasoline" in base.lower() else base]
    return list(dict.fromkeys([p.strip() for p in parts if p and len(p) > 1]))

def _search_newsapi(q: str, start: str, end: str) -> list[dict]:
    if not NEWSAPI_KEY:
        return []
    url = "https://newsapi.org/v2/everything"
    try:
        r = requests.get(url, params={
            "q": q, "from": start, "to": end,
            "language": "en", "sortBy": "relevancy", "pageSize": 5,
            "apiKey": NEWSAPI_KEY
        }, timeout=20)
        r.raise_for_status()
        data = r.json()
        items = data.get("articles", [])[:NEWS_MAX_PER_DRIVER]
        out = []
        for it in items:
            out.append({
                "title": it.get("title","")[:200],
                "outlet": (it.get("source") or {}).get("name",""),
                "url": it.get("url",""),
                "published": it.get("publishedAt","")[:10]
            })
        return out
    except Exception:
        return []

def _search_gdelt(q: str, start: str, end: str) -> list[dict]:
    def ymdhms(d: str, end=False) -> str:
        return d.replace("-", "") + ("235959" if end else "000000")
    base = "https://api.gdeltproject.org/api/v2/doc/doc"
    try:
        r = requests.get(base, params={
            "query": q, "mode": "ArtList", "format": "json",
            "sort": "DateDesc",
            "startdatetime": ymdhms(start), "enddatetime": ymdhms(end, True),
            "maxrecords": 10
        }, timeout=20)
        r.raise_for_status()
        js = r.json()
        arts = js.get("articles", [])[:NEWS_MAX_PER_DRIVER]
        out = []
        for a in arts:
            out.append({
                "title": a.get("title","")[:200],
                "outlet": a.get("sourceCommonName","") or a.get("domain",""),
                "url": a.get("url",""),
```

```
                "published": (a.get("seendate","") or "")[:10]
            })
        return out
    except Exception:
        return []

# ====== Small JSON-call helper (for synonyms) ======
def _ollama_json_call(payload_obj: dict, system_prompt: str) -> dict:
    """Tiny helper to ask Ollama for a small JSON (used for synonyms)."""
    body = {
        "model": MODEL_NAME,
        "messages": [
            {"role":"system","content": system_prompt},
            {"role":"user","content": json.dumps(payload_obj, separators=(',',':'))}
        ],
        "format": "json",
        "think": False,
        "options": {
            "seed": SEED, "temperature": 0.1, "top_p": 0.9,
            "repeat_penalty": 1.1, "num_predict": -1, "num_ctx": 2048, "stop": ["```"]
        },
        "stream": False
    }
    start = time.time()
    try:
        r = requests.post(OLLAMA_URL, json=body, timeout=(15, 120))
        r.raise_for_status()
        data = r.json()
        msg = (data or {}).get("message") or {}
        content = (msg.get("content") or "").strip()
        raw = content
        try:
            js = json.loads(content)
        except Exception:
            js = _attempt_json_repair(content)
        dur = time.time() - start
        # minimal metrics for synonyms calls
        LLM_METRICS_LOG.append({
            "event": "driver_synonyms",
            "model": MODEL_NAME,
            "duration_sec": round(dur,3),
            "prompt_tokens_est": _estimate_tokens(system_prompt + json.dumps(payload_obj)),
            "output_tokens_est": _estimate_tokens(raw),
        })
        return js
    except Exception:
        return {"terms": {}}

def build_driver_synonyms_dynamic_per_month(months_list: List[Dict[str, Any]]) -> Dict[str, Dict[str, List[str]]]:
    """
    Returns {ym: {driver: [terms...]}}
    Primary: ask the LLM per month for synonyms/queries.
    Fallback: simple expansions + hard-coded synonyms.
    """
    if not ENABLE_WEB_CONTEXT:
        return {}
    out: Dict[str, Dict[str, List[str]]] = {}
    for m in months_list:
        ym = m["month"]
        drivers = sorted({g["name"] for g in m["groups"]})
        sys_prompt = (
            "Return JSON only. For each driver name, and given the month (YYYY-MM), "
            "produce up to 6 concise search terms: synonyms/abbreviations AND seasonal/context phrases "
            "relevant to that month (e.g., refinery maintenance, winter driving season). "
            "Schema: {\"terms\": {\"<driver>\": [\"term1\",\"term2\",...]}}. "
            "No quotes inside values, no trailing commas."
        )
        resp = _ollama_json_call({"month": ym, "drivers": drivers}, sys_prompt)
        terms_map = (resp or {}).get("terms", {}) if isinstance(resp, dict) else {}
        ym_map: Dict[str, List[str]] = {}
        for d in drivers:
            llm_terms = terms_map.get(d) if isinstance(terms_map, dict) else None
            if isinstance(llm_terms, list) and llm_terms:
                base = _simple_expansion_terms(d)
                ym_map[d] = list(dict.fromkeys(base + [t for t in llm_terms if isinstance(t, str)]))
            else:
                ym_map[d] = _simple_expansion_terms(d)
        out[ym] = ym_map
    return out

def _fetch_driver_news(driver: str, ym: str, terms: List[str]) -> list[dict]:
    start, end = _month_bounds(ym)
    hits = []
    for term in terms:
        q = f"{term} (EIA OR OPEC OR refinery OR gasoline OR crude)"
        if NEWS_BACKEND == "newsapi" and NEWSAPI_KEY:
            hits += _search_newsapi(q, start, end)
        else:
            hits += _search_gdelt(q, start, end)
        if len(hits) >= NEWS_MAX_PER_DRIVER:
            break
    seen, uniq = set(), []
    for h in hits:
        u = h.get("url")
        if u and u not in seen:
            seen.add(u); uniq.append(h)
    return uniq[:NEWS_MAX_PER_DRIVER]

def build_external_context(months_list: List[Dict[str, Any]]) -> Dict[str, Any]:
    """month -> driver -> [articles]; articles = {title,outlet,url,published}"""
    if not ENABLE_WEB_CONTEXT:
        return {}
    # Build dynamic, month-specific synonyms (primary), with fallback baked in
    dyn_terms = build_driver_synonyms_dynamic_per_month(months_list)
    ctx = {}
    for m in months_list:
        ym = m["month"]
        ctx[ym] = {}
        for g in m["groups"]:
            name = g["name"]
            terms = (dyn_terms.get(ym, {}) or {}).get(name) or _simple_expansion_terms(name)
            ctx[ym][name] = _fetch_driver_news(name, ym, terms)
    return ctx

# Small grounding defs (unchanged)
DRIVER_DEFS = {
    "Product_Supplied_Demand": "EIA uses 'product supplied' as a proxy for consumption (implied demand).",
    "Refinery_Utilization_Rate": "Utilization reflects refinery inputs relative to operable capacity; higher runs often coincide with strong demand.",
    "Gasoline_Stock_Level": "Inventories reflect supply/demand balance; builds often signal weaker demand or higher supply.",
    "Crude_Oil_WTI_Price": "WTI is a benchmark crude price influenced by global supply/demand and geopolitics."
}

def build_llm_payload(monthly_flat: Dict[str, Any], groups: List[str], months: List[str]):
    by_month = {m: [] for m in months}
    for rec in monthly_flat.values():
        ym = rec["Date"][:7]
        by_month[ym].append({"name": rec["Group"], "value": float(rec["Avg_grp_contri"])})
```

```
    net_effects = compute_net_effect_for_month(by_month)
    label_policy = build_label_policy_from_monthly(monthly_flat)

    months_list = [{"month": m, "groups": by_month[m], "net_effect": net_effects[m]} for m in months]
    response_contract = [{"month": m, "net_effect": net_effects[m]} for m in months]

    group_trends = build_group_trends(months_list)
    month_balance = build_month_balance(months_list)
    prev_map = build_prev_map(months_list)

    external_context = build_external_context(months_list)  # may be {}

    return {
        "months": months_list,
        "months_count": len(months),
        "month_order": months,
        "label_policy": label_policy,
        "response_contract": response_contract,
        "context": {
            "group_trends": group_trends,
            "month_balance": month_balance,
            "prev_groups": prev_map,
            "external_context": external_context,
            "driver_defs": DRIVER_DEFS,
            "interpretation_guide": (
                "SHAP values are additive explanations relative to a baseline; "
                "they describe contribution signs/magnitudes, not causal effects."
            )
        },
        "notes": {
            "net_effect_definition": "Sum across groups for that month (SHAP additivity).",
            "style": "Be specific and concise; avoid hype; write for business readers."
        }
    }

# ------------ JSON  SCHEMA ---------------
SCHEMA_VERSION = os.getenv("SCHEMA_VERSION", "1").strip()
if SCHEMA_VERSION == "2":
    SCHEMA = {
      "type": "object",
      "properties": {
        "months": {
          "type": "array", "minItems": 4, "maxItems": 4,
          "items": {
            "type": "object",
            "properties": {
              "month": { "type": "string" },
              "summary": { "type": "string" },
              "drivers": {
                "type":"array",
                "items":{
                  "type":"object",
                  "properties":{
                    "group":{"type":"string"},
                    "direction":{"enum":["up","down"]},
                    "magnitude":{"type":"number"},
                    "qualifier":{"enum":["slightly","moderately","significantly"]}
                  },
                  "required":["group","direction","magnitude","qualifier"],
                  "additionalProperties": False
                }
              },
              "net_effect": { "type":"number" },
              "net_explanation": { "type":"string" },
              "evidence": {
                "type":"array",
                "items":{
                  "type":"object",
                  "properties":{
                    "driver":{"type":"string"},
                    "why":{"type":"string"},
                    "sources":{
                      "type":"array",
                      "items":{
                        "type":"object",
                        "properties":{
                          "title":{"type":"string"},
                          "outlet":{"type":"string"},
                          "url":{"type":"string"},
                          "published":{"type":"string"}
                        },
                        "required":["title","url"]
                      }
                    }
                  },
                  "required":["driver","why","sources"]
                }
              },
              "watch_next": { "type":"string" }
            },
            "required": ["month","summary","drivers","net_effect","net_explanation"],
            "additionalProperties": False
          }
        },
        "overall_summary": { "type":"string" }
      },
      "required": ["months","overall_summary"],
      "additionalProperties": False
    }
else:
    SCHEMA = {
      "type": "object",
      "properties": {
        "months": {
          "type": "array",
          "minItems": 4,
          "maxItems": 4,
          "items": {
            "type": "object",
            "properties": {
              "month": { "type": "string" },
              "summary": { "type": "string" },
              "drivers": {
                "type":"array",
                "items":{
                  "type":"object",
                  "properties":{
                    "group":{"type":"string"},
                    "direction":{"enum":["up","down"]},
                    "magnitude":{"type":"number"},
                    "qualifier":{"enum":["slightly","moderately","significantly"]}
                  },
                  "required":["group","direction","magnitude","qualifier"],
                  "additionalProperties": False
                }
              },
```

```
              "net_effect": { "type":"number" },
              "net_explanation": { "type":"string" },
              "contextual_reasons": {
                "type":"array",
                "items":{
                  "type":"object",
                  "properties":{
                    "driver":{"type":"string"},
                    "reason":{"type":"string"},
                    "source_type":{"enum":["news","hypothesis","internal"]},
                    "source_url":{"type":"string"},
                    "confidence":{"enum":["low","medium","high"]},
                    "disclaimer":{"type":"string"}
                  },
                  "required":["driver","reason","source_type","confidence","disclaimer"],
                  "additionalProperties": False
                }
              }
            },
            "required": ["month","summary","drivers","net_effect","net_explanation"],
            "additionalProperties": False
          }
        },
        "overall_summary": { "type":"string" }
      },
      "required": ["months","overall_summary"],
      "additionalProperties": False
    }

FORMAT_SCHEMA = {
  "type": "json_schema",
  "json_schema": {
    "name": "xai_monthly_report",
    "schema": SCHEMA
  }
}

# ------------------ SYSTEM PROMPT -----------------
SYSTEM_PROMPT = (
    "You are an XAI Narrator for time-series model drivers.\n"
    "You write succinct, analyst-grade monthly narratives that are dense with specifics, not fluff.\n"
    "INPUT:\n"
    " â€¢ months[4]: each has {month:'YYYY-MM', groups:[{name,value}], net_effect}.\n"
    " â€¢ label_policy: absolute-magnitude bins â†’ {slightly, moderately, significantly}.\n"
    " â€¢ response_contract: {month, net_effect} â†’ you MUST echo these net_effect values.\n"
    " â€¢ context: {group_trends, month_balance, prev_groups, driver_defs, external_context}.\n"
    "STYLE & SCOPE:\n"
    " â€¢ For each month, write 2-4 sentences.\n"
    " â€¢ Mention up to the top-3 positive AND top-3 negative drivers (if present), each with adjective + signed magnitude.\n"
    " â€¢ Compare to prior month when meaningful (sign flips, large deltas).\n"
    " â€¢ Explain the net using the balance_label (offsetting/mixed/concentrated) and |net|/Î£|effects| ratio.\n"
    " â€¢ If external_context has headlines for a driver that month, you MAY add a short plausible reason:\n"
    "    - Preface with 'reported at the time' or 'according to contemporaneous reports'.\n"
    "    - Avoid causal claims; correlation â‰  causation.\n"
    "    - Keep to one short clause per cited driver.\n"
    " â€¢ If all magnitudes are below 'slightly', say 'No dominant drivers' and return empty drivers.\n"
    "OUTPUT JSON:\n"
    " â€¢ months: EXACTLY 4, in the SAME order as month_order; copy net_effects from response_contract.\n"
    " â€¢ drivers: include those you mention (â‰¥2 when available; more allowed).\n"
    " â€¢ net_explanation: concise sentence naming why the net landed where it did.\n"
    " â€¢ If schema v2: include 'evidence' (driver-tagged 1-2 bullets) and 'watch_next' (1-2 risks).\n"
    "JSON RULES: no embedded double quotes; no trailing commas.\n"
)

# ----------------- OLLAMA CALL -------------------------
def _largest_balanced_json_object(text: str) -> Optional[str]:
    start = None
    depth = 0
    best = None
    for i, ch in enumerate(text):
        if ch == '{':
            if depth == 0:
                start = i
            depth += 1
        elif ch == '}':
            if depth > 0:
                depth -= 1
                if depth == 0 and start is not None:
                    cand = text[start:i+1]
                    if best is None or len(cand) > len(best):
                        best = cand
    return best

def _quote_unquoted_keys(s: str) -> str:
    return re.sub(r'([{\s,])([A-Za-z_][A-Za-z0-9_\-]*)\s*:', r'\1"\2":', s)

def _fix_constants(s: str) -> str:
    s = re.sub(r'\bNaN\b', 'null', s, flags=re.IGNORECASE)
    s = re.sub(r'\bInfinity\b', '1e9999', s, flags=re.IGNORECASE)
    s = re.sub(r'\b-Inf(inity)?\b', '-1e9999', s, flags=re.IGNORECASE)
    return s

def _normalize_quotes(s: str) -> str:
    s = s.replace("\u201c", '"').replace("\u201d", '"').replace("\u201e", '"').replace("\u201f", '"')
    s = s.replace("\u2018", "'").replace("\u2019", "'")
    return s

def _strip_fences_and_garbage(s: str) -> str:
    s = s.strip().lstrip("\ufeff")
    if s.startswith("```"):
        blocks = [b.strip() for b in s.split("```") if b.strip()]
        for b in blocks:
            if b.startswith("{") or b.lower().startswith("json"):
                s = b
                break
    if s.lower().startswith("json"):
        s = s[4:].strip()
    if not s.startswith("{") or not s.endswith("}"):
        maybe = _largest_balanced_json_object(s)
        if maybe:
            s = maybe
    return s

def _attempt_json_repair(text: str) -> dict:
    if not text:
        raise ValueError("Empty content; nothing to repair")
    t = _strip_fences_and_garbage(text)
    t = _normalize_quotes(t)
    t = re.sub(r",(\s*[}\]])", r"\1", t)
    t = _quote_unquoted_keys(t)
    t = _fix_constants(t)
    try:
        return json.loads(t)
    except Exception:
        pass
    if _repair_json_lib is not None:
        try:
```

```
            repaired = _repair_json_lib(t)
            return json.loads(repaired)
        except Exception:
            pass
    maybe = _largest_balanced_json_object(t)
    if maybe:
        return json.loads(maybe)
    raise

_SCHEMAS_SUPPORTED = None
def _schemas_supported() -> bool:
    global _SCHEMAS_SUPPORTED
    if _SCHEMAS_SUPPORTED is not None:
        return _SCHEMAS_SUPPORTED
    try:
        r = requests.post(
            OLLAMA_URL,
            json={
                "model": MODEL_NAME,
                "messages": [{"role": "user", "content": "ping"}],
                "format": {"type": "json_schema", "json_schema": {"name": "ping", "schema": {"type": "object"}}},
                "stream": False
            },
            timeout=(5, 10)
        )
        _SCHEMAS_SUPPORTED = (r.status_code == 200)
    except Exception:
        _SCHEMAS_SUPPORTED = False
    return _SCHEMAS_SUPPORTED

def _extract_possible_json_from_http_error(e: Exception) -> Optional[str]:
    try:
        resp = getattr(e, "response", None)
        if resp is None:
            return None
        try:
            j = resp.json()
            for k in ("response", "message", "data", "error"):
                v = j.get(k)
                if isinstance(v, str) and ("{" in v):
                    return v
        except Exception:
            pass
        txt = resp.text or ""
        return txt if "{" in txt else None
    except Exception:
        return None

def call_ollama_llm(payload: Dict[str, Any]) -> tuple[dict, str]:
    """Calls Ollama; returns (json, raw_text). Also records LAST_CALL_METRICS & logs."""
    # Warm-up (non-fatal)
    try:
        requests.post(
            OLLAMA_URL,
            json={
                "model": MODEL_NAME,
                "messages": [{"role":"user","content":"ok"}],
                "think": False,
                "options": {
                    "seed": SEED, "temperature": TEMPERATURE, "top_p": TOP_P,
                    "repeat_penalty": 1.1, "num_predict": -1, "num_ctx": 8192, "stop": ["````"]
                },
                "stream": False
            },
            timeout=(10, 60)
        ).raise_for_status()
    except Exception:
        pass

    def _do_call(body: dict, mode: str) -> tuple[dict, str, float, int, int]:
        sys_text = ""
        for m in body.get("messages", []):
            if m.get("role") == "system":
                sys_text += m.get("content","")
        user_text = ""
        for m in body.get("messages", []):
            if m.get("role") == "user":
                user_text += m.get("content","")
        prompt_tokens = _estimate_tokens(sys_text) + _estimate_tokens(user_text)
        start = time.time()
        resp = requests.post(OLLAMA_URL, json=body, timeout=(15, 300))
        resp.raise_for_status()
        data = resp.json()
        msg = (data or {}).get("message") or {}
        content = (msg.get("content") or "").strip()
        raw = content
        try:
            js = json.loads(content)
        except Exception:
            try:
                js = _attempt_json_repair(content)
            except Exception:
                best = _largest_balanced_json_object(content)
                if best:
                    js = json.loads(best)
                else:
                    raise
        dur = time.time() - start
        out_tokens = _estimate_tokens(raw)
        return js, raw, dur, prompt_tokens, out_tokens

    # Try schema mode first
    try_schema = _schemas_supported()
    if try_schema:
        body_schema = {
            "model": MODEL_NAME,
            "messages": [
                {"role":"system","content": SYSTEM_PROMPT},
                {"role":"user","content": json.dumps({"facts": payload, "contract": payload["response_contract"]}, separators=(',',':'))}
            ],
            "think": False,
            "format": FORMAT_SCHEMA,
            "options": {
                "seed": SEED, "temperature": TEMPERATURE, "top_p": TOP_P,
                "repeat_penalty": 1.1, "num_predict": -1, "num_ctx": 8192, "stop": ["```"]
            },
            "keep_alive": "30m",
            "stream": False
        }
        try:
            js, raw, dur, ptok, otok = _do_call(body_schema, "schema")
            LAST_CALL_METRICS.update({
                "model": MODEL_NAME, "schema": True, "duration_sec": round(dur,3),
                "prompt_tokens_est": ptok, "output_tokens_est": otok,
                "tps_est": round((otok / dur) if dur>0 else 0.0, 2)
            })
            LLM_METRICS_LOG.append({**LAST_CALL_METRICS})
```

```
            return js, raw
        except Exception as e1:
            possible = _extract_possible_json_from_http_error(e1)
            if possible:
                try:
                    repaired = _attempt_json_repair(possible)
                    raw = possible
                    dur = 0.0
                    ptok = _estimate_tokens(SYSTEM_PROMPT) + _estimate_tokens(json.dumps(payload))
                    otok = _estimate_tokens(raw)
                    LAST_CALL_METRICS.update({
                        "model": MODEL_NAME, "schema": True, "duration_sec": round(dur,3),
                        "prompt_tokens_est": ptok, "output_tokens_est": otok,
                        "tps_est": 0.0
                    })
                    LLM_METRICS_LOG.append({**LAST_CALL_METRICS})
                    return repaired, raw
                except Exception:
                    pass
            print("[WARN] Schema call failed; switching to plain JSON mode...")

    # Plain JSON fallback
    body_plain = {
        "model": MODEL_NAME,
        "messages": [
            {"role":"system","content": SYSTEM_PROMPT + "\nReturn ONE JSON object only."},
            {"role":"user","content": json.dumps(payload, separators=(',',':'))}
        ],
        "think": False,
        "format": "json",
        "options": {
            "seed": SEED, "temperature": TEMPERATURE, "top_p": TOP_P,
            "repeat_penalty": 1.1, "num_predict": -1, "num_ctx": 8192, "stop": ["```"]
        },
        "keep_alive": "30m",
        "stream": False
    }
    js, raw, dur, ptok, otok = _do_call(body_plain, "plain")
    LAST_CALL_METRICS.update({
        "model": MODEL_NAME, "schema": False, "duration_sec": round(dur,3),
        "prompt_tokens_est": ptok, "output_tokens_est": otok,
        "tps_est": round((otok / dur) if dur>0 else 0.0, 2)
    })
    LLM_METRICS_LOG.append({**LAST_CALL_METRICS})
    return js, raw

def compute_net_effect_for_month(by_month: Dict[str, List[Dict[str, float]]]) -> Dict[str, float]:
    out = {}
    for ym, items in by_month.items():
        s = sum(float(it["value"]) for it in items)
        out[ym] = round(s, 2)
    return out

def _qualifier_for(value: float, policy: Dict[str, Dict[str, float]]) -> str:
    a = abs(value)
    lo = policy["slightly"]["lt"]
    mid_ge = policy["moderately"]["ge"]; mid_lt = policy["moderately"]["lt"]
    if a < lo: return "slightly"
    if a >= mid_ge and a < mid_lt: return "moderately"
    return "significantly"

def _extrema(groups: List[Dict[str, float]]):
    pos = [g for g in groups if g["value"] > 0]
    neg = [g for g in groups if g["value"] < 0]
    top_pos = max(pos, key=lambda x: x["value"]) if pos else None
    top_neg = min(neg, key=lambda x: x["value"]) if neg else None
    return top_pos, top_neg

def _rb_summarize(month_rec: Dict[str, Any], policy: Dict[str, Any], prev_values: Optional[Dict[str, float]] = None) -> Tuple[str, List[Dict[str, Any]], str]:
    """Rule-based fallback: now includes top-N drivers by |magnitude| (N=XAI_MAX_DRIVERS) + extrema mention."""
    groups = month_rec.get("groups", [])
    net = float(month_rec["net_effect"])
    prev_values = prev_values or {}

    sorted_groups = sorted(groups, key=lambda g: abs(g["value"]), reverse=True)
    topk = sorted_groups[:max(2, XAI_MAX_DRIVERS)]

    sum_abs = sum(abs(g["value"]) for g in groups) or 0.0
    balance_ratio = abs(net) / sum_abs if sum_abs else 0.0
    if balance_ratio < 0.30: balance_label = "mostly offsetting"
    elif balance_ratio < 0.60: balance_label = "mixed"
    else: balance_label = "concentrated"

    drivers, parts = [], []

    def delta_str(name: str, cur_val: float) -> str:
        if name in prev_values:
            d = cur_val - prev_values[name]
            if abs(d) >= 0.05:
                return f" vs prior month {d:+.2f}"
        return ""

    for item in topk:
        name = item["name"]; val = float(item["value"])
        q = _qualifier_for(val, policy)
        if val >= 0:
            drivers.append({"group": name, "direction": "up", "magnitude": round(val, 2), "qualifier": q})
            parts.append(f'{name} contributed {q} positive impact (+{val:.2f}{delta_str(name, val)})')
        else:
            drivers.append({"group": name, "direction": "down", "magnitude": round(abs(val), 2), "qualifier": q})
            parts.append(f'{name} contributed {q} negative impact (-{abs(val):.2f}{delta_str(name, val)})')

    # Mention extrema explicitly (uses the previously unused helper)
    top_pos, top_neg = _extrema(groups)
    if top_pos or top_neg:
        tail = []
        if top_pos: tail.append(f"strongest up: {top_pos['name']} (+{top_pos['value']:.2f})")
        if top_neg: tail.append(f"strongest down: {top_neg['name']} (-{top_neg['value']:.2f})")
        if tail:
            parts.append("; ".join(tail))

    if not parts:
        summary = "No dominant drivers; movements were small and largely offsetting."
        drivers = []
    else:
        summary = "; ".join(parts)
        if len(summary) > 240:
            mid = len(summary)//2
            cut = summary.rfind("; ", 0, mid)
            if cut != -1:
                summary = summary[:cut+1] + " " + summary[cut+2:] + "."
            else:
                summary += "."
        else:
            summary += "."

    if sum_abs > 0:
```

```
        net_expl = f'Net effect {net:+.2f} = sum of monthly SHAP contributions. Profile: {balance_label} (|net|/Î£|effects| = {balance_ratio:.2f}).'
    else:
        net_expl = "Net effect +0.00 = no material contributions this month."

    summary += f' Overall monthly impact: {net:+.2f} ({balance_label}).'
    return summary, drivers, net_expl


def _compose_overall_summary(payload: Dict[str, Any], months_out: List[Dict[str, Any]]) -> str:
    """
    Build a detailed, business-friendly overall summary from payload facts + normalized months_out.
    Returns a multi-line string.
    """
    months_order: List[str] = payload["month_order"]
    month_balance: Dict[str, Any] = payload["context"]["month_balance"]
    ext_ctx: Dict[str, Any] = payload["context"].get("external_context", {}) or {}
    # raw facts per month for full group coverage
    facts_by_month = {m["month"]: m for m in payload["months"]}
    # nets by month
    nets = {m: round(float(facts_by_month[m]["net_effect"]), 2) for m in months_order}
    # label buckets
    def _lab(m): return (month_balance[m]["balance_label"], month_balance[m]["balance_ratio"])
    offsetting = [(m, nets[m], _lab(m)[1]) for m in months_order if _lab(m)[0] == "mostly offsetting"]
    mixed      = [(m, nets[m], _lab(m)[1]) for m in months_order if _lab(m)[0] == "mixed"]
    conc_pos   = [(m, nets[m], _lab(m)[1]) for m in months_order if _lab(m)[0] == "concentrated" and nets[m] > 0]
    conc_neg   = [(m, nets[m], _lab(m)[1]) for m in months_order if _lab(m)[0] == "concentrated" and nets[m] < 0]

    # group-level stats across the 4 months
    groups = {}
    for m in months_order:
        for g in facts_by_month[m]["groups"]:
            name = g["name"]; val = float(g["value"])
            groups.setdefault(name, {"vals": [], "months": []})
            groups[name]["vals"].append(val)
            groups[name]["months"].append(m)

    group_stats = {}
    for name, info in groups.items():
        vals = info["vals"]; ms = info["months"]
        total = round(sum(vals), 2)
        pos_ct = sum(1 for v in vals if v > 0)
        neg_ct = sum(1 for v in vals if v < 0)
        sign_flips = sum(1 for i in range(1, len(vals)) if (vals[i] > 0) != (vals[i-1] > 0))
        max_up = max(vals) if vals else 0.0
        max_up_idx = vals.index(max_up) if vals else 0
        max_down = min(vals) if vals else 0.0
        max_down_idx = vals.index(max_down) if vals else 0
        rng = round((max_up - max_down), 2)
        mom = round(vals[-1] - vals[0], 2) if len(vals) >= 2 else 0.0
        group_stats[name] = {
            "total": total,
            "pos_ct": pos_ct,
            "neg_ct": neg_ct,
            "sign_flips": sign_flips,
            "max_up": round(max_up, 2),
            "max_up_m": ms[max_up_idx],
            "max_down": round(max_down, 2),
            "max_down_m": ms[max_down_idx],
            "range": abs(rng),
            "momentum": mom,
            "last": round(vals[-1], 2),
            "first": round(vals[0], 2),
        }

    # top tailwind/headwind by cumulative total
    sorted_total = sorted(group_stats.items(), key=lambda kv: kv[1]["total"], reverse=True)
    top_tail = next((kv for kv in sorted_total if kv[1]["total"] > 0), None)
    top_head = next((kv for kv in sorted(group_stats.items(), key=lambda kv: kv[1]["total"]) if kv[1]["total"] < 0), None)
    # most volatile by (sign_flips, range)
    most_vol = max(group_stats.items(), key=lambda kv: (kv[1]["sign_flips"], kv[1]["range"])) if group_stats else None
    # last-month leaders (top 2 ups/downs)
    last_m = months_order[-1]
    last_vals = []
    for name, st in group_stats.items():
        # find last month value
        try:
            idx = groups[name]["months"].index(last_m)
            last_vals.append((name, round(groups[name]["vals"][idx], 2)))
        except Exception:
            pass
    last_ups  = sorted([(n,v) for (n,v) in last_vals if v > 0], key=lambda x: x[1], reverse=True)[:2]
    last_down = sorted([(n,abs(v)) for (n,v) in last_vals if v < 0], key=lambda x: x[1], reverse=True)[:2]

    # news/evidence hit count
    news_hits = 0
    for ym, dmap in ext_ctx.items():
        for _drv, arts in (dmap or {}).items():
            news_hits += len(arts or [])

    # month label helper
    def ml(m): return datetime.strptime(m + "-01", "%Y-%m-%d").strftime("%b %Y")
    def mfmt(triplets):
        return ", ".join(f"{ml(m)} ({nets[m]:+,.2f}; r={r:.2f})" for m,_,r in triplets)

    lines = []
    # Header / net picture
    window = f"{ml(months_order[0])}â€“{ml(months_order[-1])}"
    head_bits = []
    if offsetting: head_bits.append(f"offsetting in {mfmt(offsetting)}")
    if mixed:      head_bits.append(f"mixed in {mfmt(mixed)}")
    if conc_pos:   head_bits.append(f"concentrated up in {mfmt(conc_pos)}")
    if conc_neg:   head_bits.append(f"concentrated down in {mfmt(conc_neg)}")
    if head_bits:
        lines.append(f"Net picture ({window}): " + "; ".join(head_bits) + ".")
    else:
        lines.append(f"Net picture ({window}): monthly nets were modest without clear concentration.")

    # Driver storyline
    story = []
    if top_tail:
        n, st = top_tail
        story.append(f"Top tailwind: {n} (total {st['total']:+.2f}; peak {st['max_up']:+.2f} in {ml(st['max_up_m'])}).")
    if top_head:
        n, st = top_head
        story.append(f"Largest headwind: {n} (total {st['total']:+.2f}; trough {st['max_down']:+.2f} in {ml(st['max_down_m'])}).")
    if most_vol:
        n, st = most_vol
        story.append(f"Most volatile: {n} (sign flips {st['sign_flips']}, range Â±{st['range']:.2f}).")
    if last_ups:
        story.append("Momentum into last month: " + ", ".join(f"{n} (+{v:.2f})" for n,v in last_ups) + ".")
    if last_down:
        story.append("Last-month headwinds: " + ", ".join(f"{n} (-{v:.2f})" for n,v in last_down) + ".")
    if story:
        lines.append("Driver storyline: " + " ".join(story))

    # Mix & concentration quick view
    ratios = ", ".join(f"{ml(m)} {month_balance[m]['balance_ratio']:.2f}" for m in months_order)
```

```
    lines.append(f"Mix & concentration (|net|/Î£|effects|): {ratios}.")

    # Evidence/news note
    if news_hits > 0:
        lines.append(f"Context used: {news_hits} contemporaneous headline{'s' if news_hits!=1 else ''} (correlation â‰  causation).")
    else:
        lines.append("Context used: no contemporaneous headlines matched the month windows.")

    # Watch next (risk cues)
    watch_bits = []
    if most_vol:
        n, _ = most_vol
        watch_bits.append(f"{n} directionality (another sign flip could dominate the net)")
    if top_head:
        n, _ = top_head
        watch_bits.append(f"persistence of {n} headwinds")
    if not watch_bits:
        watch_bits.append("shifts in drivers with largest absolute monthly magnitudes")
    lines.append("What to watch next: " + "; ".join(watch_bits) + ".")

    # SHAP reminder
    lines.append("Note: SHAP is additive relative to a baseline; it explains contributions, not causality.")

    return "\n".join(lines)


def normalize_llm_out(llm_out: Dict[str, Any], payload: Dict[str, Any]) -> Dict[str, Any]:
    policy = payload["label_policy"]
    want_order = payload["month_order"]
    by_month = {m["month"]: m for m in payload["months"]}
    prev_map = payload.get("context", {}).get("prev_groups", {})
    external_ctx = payload.get("context", {}).get("external_context", {}) or {}

    months_out = []
    got = {m["month"]: m for m in llm_out.get("months", []) if isinstance(m, dict)}

    for ym in want_order:
        base = {"month": ym, "net_effect": by_month[ym]["net_effect"]}
        existing = got.get(ym, {})
        summary = existing.get("summary")
        drivers = existing.get("drivers")
        net_effect = existing.get("net_effect", base["net_effect"])
        net_expl = existing.get("net_explanation")

        contextual_reasons = existing.get("contextual_reasons", None)
        evidence = existing.get("evidence", None)
        wn = existing.get("watch_next")

        if not (summary and isinstance(drivers, list) and net_expl):
            facts = {"month": ym, "groups": by_month[ym]["groups"], "net_effect": net_effect}
            summary, drivers, net_expl = _rb_summarize(facts, policy, prev_map.get(ym, {}))

        month_obj = {
            "month": ym,
            "summary": summary,
            "drivers": drivers,
            "net_effect": net_effect,
            "net_explanation": net_expl
        }

        ext_month = external_ctx.get(ym, {})
        if SCHEMA_VERSION == "2":
            if evidence is None:
                ev = []
                for d in (drivers or [])[:2]:
                    arts = ext_month.get(d["group"], [])
                    if arts:
                        ev.append({"driver": d["group"], "why": "reported at the time; correlation, not causation", "sources": arts})
                    else:
                        ev.append({"driver": d["group"], "why": "No relevant news available in this month window", "sources": []})
                month_obj["evidence"] = ev
            else:
                month_obj["evidence"] = evidence
            if wn:
                month_obj["watch_next"] = wn
        else:
            if contextual_reasons is None:
                cr = []
                for d in (drivers or [])[:2]:
                    arts = ext_month.get(d["group"], [])
                    if arts:
                        a0 = arts[0]
                        cr.append({
                            "driver": d["group"],
                            "reason": f"reported at the time: {a0.get('title','')}",
                            "source_type": "news",
                            "source_url": a0.get("url",""),
                            "confidence": "low",
                            "disclaimer": "Correlation â‰  causation; used as plausible context."
                        })
                    else:
                        cr.append({
                            "driver": d["group"],
                            "reason": "No relevant news available in this month window",
                            "source_type": "news",
                            "source_url": "",
                            "confidence": "low",
                            "disclaimer": "No contemporaneous sources found for the window."
                        })
                month_obj["contextual_reasons"] = cr
            else:
                month_obj["contextual_reasons"] = contextual_reasons

        months_out.append(month_obj)


    llm_overall = llm_out.get("overall_summary") if isinstance(llm_out, dict) else None
    auto_overall = _compose_overall_summary(payload, months_out)
    if XAI_OVERALL_MODE == "llm" and isinstance(llm_overall, str) and llm_overall.strip():
        overall = llm_overall
    elif XAI_OVERALL_MODE == "rb":
        overall = auto_overall
    else:  # 'hybrid' (default): if LLM overall is too short (<200 chars), replace with detailed version
        if isinstance(llm_overall, str) and len(llm_overall.strip()) >= 200:
            overall = llm_overall
        else:
            overall = auto_overall

    return {"months": months_out, "overall_summary": overall}

# ---------------- DATA PREPROCESSING ------------------
def _safe_load_weekly_json(x: Any) -> Dict[str, Any]:
    if isinstance(x, dict):
        return x
    if isinstance(x, str):
        try:
            return json.loads(x)
```

```
        except Exception:
            if _repair_json_lib is not None:
                try:
                    repaired = _repair_json_lib(x)
                    return json.loads(repaired)
                except Exception:
                    pass
            maybe = _largest_balanced_json_object(x)
            if maybe:
                return json.loads(maybe)
    raise ValueError("Unable to parse Weekly_JSON_Drivers")

def _make_monthly_text(llm_out: Dict[str, Any], payload: Dict[str, Any]) -> str:
    lines = []
    for m in payload["months"]:
        ym = m["month"]
        mo = next((x for x in llm_out["months"] if x["month"] == ym), None)
        if mo:
            label = datetime.strptime(ym+"-01", "%Y-%m-%d").strftime("%b %Y")
            balance = payload["context"]["month_balance"][ym]["balance_label"]
            lines.append(f"- {label}: {mo['summary']} Net (sum of SHAP): {mo['net_effect']:+.2f} [{balance}]")
            # v2 evidence preferred
            evid = mo.get("evidence", [])
            if evid:
                for ev in evid[:2]:
                    srcs = ev.get("sources", [])[:2]
                    if srcs:
                        src_bits = "; ".join(f"{s.get('outlet','')}: {s.get('title','')}" for s in srcs if s.get("title"))
                        lines.append(f"  â€¢ Evidence ({ev.get('driver','')}): {ev.get('why','')}. {src_bits}")
                    else:
                        lines.append(f"  â€¢ Evidence ({ev.get('driver','')}): {ev.get('why','')}.")
            else:
                # v1 contextual reasons or explicit none
                cr = mo.get("contextual_reasons", [])
                if cr:
                    for r in cr[:2]:
                        if r.get("source_url"):
                            lines.append(f"  â€¢ Context ({r.get('driver','')}): {r.get('reason','')} [{r.get('source_url')}]")
                        else:
                            lines.append(f"  â€¢ Context ({r.get('driver','')}): {r.get('reason','')}")
    lines.append("\n=== OVERALL SUMMARY ===")
    lines.append(llm_out.get("overall_summary", ""))
    return "\n".join(lines)

def run_xai_once(weekly_flat: Dict[str, Any]) -> Tuple[Dict[str, Any], Dict[str, Any], Dict[str, Any], str]:
    monthly_flat, groups, months = aggregate_weekly_flat_to_monthly_mean(weekly_flat, out_months=4)
    payload = build_llm_payload(monthly_flat, groups, months)
    try:
        llm_raw_out, _raw = call_ollama_llm(payload)
        llm_out = normalize_llm_out(llm_raw_out, payload)
    except Exception:
        llm_out = normalize_llm_out({}, payload)
    xai_text = _make_monthly_text(llm_out, payload)
    return monthly_flat, payload, llm_out, xai_text

def run_pipeline_from_df(df_in: pd.DataFrame, mode: str = "test") -> Optional[pd.DataFrame]:
    weekly_col = next((c for c in df_in.columns if c.strip().lower() in ("weekly_json_drivers","weekly_drivers")), None)
    if weekly_col is None:
        raise ValueError("Input DataFrame must contain a 'Weekly_JSON_Drivers' (or 'Weekly_drivers') column.")
    model_col = next((c for c in df_in.columns if c.strip().lower() == "model"), None)
    ref_col   = next((c for c in df_in.columns if c.strip().lower() in ("reference_date","reference_date_","ref_date")), None)
    if model_col is None or ref_col is None:
        raise ValueError("Input DataFrame must contain 'Model' and 'Reference_Date' columns.")

    if mode == "test":
        row = df_in.iloc[0]
        weekly_flat = _safe_load_weekly_json(row[weekly_col])
        monthly_flat, payload, llm_out, xai_text = run_xai_once(weekly_flat)

        print("\n=== MONTHLY MEAN JSON (flat) ===")
        print(json.dumps(monthly_flat, indent=2))
        print("\n=== LLM STRUCTURED JSON (normalized) ===")
        print(json.dumps(llm_out, indent=2, ensure_ascii=False))
        print("\n=== MONTHLY TEXT (LLM) ===")
        print(xai_text)
        # NEW: metrics print
        if LAST_CALL_METRICS:
            print("\n=== LLM METRICS ===")
            print(json.dumps(LAST_CALL_METRICS, indent=2))
        return None

    # prod
    records: List[Dict[str, Any]] = []
    for _, row in df_in.iterrows():
        try:
            weekly_flat = _safe_load_weekly_json(row[weekly_col])
            monthly_flat, payload, llm_out, xai_text = run_xai_once(weekly_flat)
            metrics = {**LAST_CALL_METRICS} if LAST_CALL_METRICS else {}
            news_hits = 0
            if ENABLE_WEB_CONTEXT:
                ext = payload.get("context", {}).get("external_context", {}) or {}
                for ym, dmap in ext.items():
                    for _drv, arts in dmap.items():
                        news_hits += len(arts or [])
            metrics["news_hits"] = news_hits
            records.append({
                "Model": row[model_col],
                "Reference_Date": row[ref_col],
                "Weekly_JSON_Drivers": json.dumps(weekly_flat, separators=(',',':'), ensure_ascii=False),
                "Monthly_JSON_Drivers": json.dumps(monthly_flat, separators=(',',':'), ensure_ascii=False),
                "XAI_JSON_Params": json.dumps(llm_out, separators=(',',':'), ensure_ascii=False),
                "XAI_Explaination": xai_text,
                # minimal metrics columns for side-by-side model comparison
                "LLM_Model": metrics.get("model", MODEL_NAME),
                "LLM_Prompt_Tokens": metrics.get("prompt_tokens_est"),
                "LLM_Output_Tokens": metrics.get("output_tokens_est"),
                "LLM_TPS_Est": metrics.get("tps_est"),
                "LLM_Duration_Sec": metrics.get("duration_sec"),
                "LLM_News_Hits": metrics.get("news_hits"),
                "LLM_Schema_Mode": metrics.get("schema"),
            })
        except Exception as e:
            records.append({
                "Model": row.get(model_col),
                "Reference_Date": row.get(ref_col),
                "Weekly_JSON_Drivers": str(row.get(weekly_col)),
                "Monthly_JSON_Drivers": "",
                "XAI_JSON_Params": "",
                "XAI_Explaination": f"[ERROR] {type(e).__name__}: {e}",
                "LLM_Model": MODEL_NAME
            })

    out_df = pd.DataFrame.from_records(records)
    out_df.to_excel(XAI_OUTPUT_XLS, index=False, engine="openpyxl")
    if XAI_SAVE_METRICS and LLM_METRICS_LOG:
        pd.DataFrame(LLM_METRICS_LOG).to_csv(XAI_METRICS_CSV, index=False)
    print(f"[OK] Wrote results to {XAI_OUTPUT_XLS} ({len(out_df)} rows).")
```

```
    if XAI_SAVE_METRICS:
        print(f"[OK] LLM metrics logged to {XAI_METRICS_CSV} ({len(LLM_METRICS_LOG)} events).")
    return out_df

# ========================
# 9) MAIN
# ========================
def main():
    try:
        df = pd.read_excel(XAI_DATA_PATH, engine="openpyxl")
    except Exception as e:
        raise RuntimeError(f"Failed to read input Excel at {XAI_DATA_PATH}: {e}")

    if RUN_MODE not in ("test","prod"):
        print(f"[WARN] Unknown RUN_MODE={RUN_MODE}; defaulting to 'test'.")
        mode = "test"
    else:
        mode = RUN_MODE

    run_pipeline_from_df(df, mode=mode)

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