

ALL SCRIPTS DUMP - C:\Data-LakeHouse\src
Total .py: 164

INDEX (first pass):

```
\src\audit\audit_ajuste_poscosecha_ml2.py
\src\audit\audit_desp_poscosecha_ml2.py
\src\audit\audit_dh_poscosecha_ml2.py
\src\audit\audit_dist_grado_ml1.py
\src\audit\audit_harvest_horizon_ml2.py
\src\audit\audit_harvest_start_ml2.py
\src\audit\audit_harvest_windows_real.py
\src\audit\audit_hidr_poscosecha_ml2.py
\src\audit\audit_mass_balance_pred_oferta_vs_gold.py
\src\audit\audit_mismatch_tallos_grado_vs_dia.py
\src\audit\audit_ml1_curva_clipping_impact.py
\src\audit\audit_ml1_curva_share_vs_real.py
\src\audit\audit_ml1_curva_shares.py
\src\audit\audit_peso_tallo_ml2.py
\src\audit\audit_poscosecha_ml2_baseline.py
\src\audit\audit_poscosecha_seed.py
\src\audit\audit_share_grado_ml2.py
\src\audit\audit_tallos_curve_ml2.py
\src\audit\audit_universe_harvest_grid_ml1.py
\src\bronze\build_balanza_lc_raw.py
\src\bronze\build_balanza_cosecha_raw.py
\src\bronze\build_balanza_mermas_sources.py
\src\bronze\build_conteo_tallos_alturas_gyp2419_raw.py
\src\bronze\build_fenograma_raw.py
\src\bronze\build_fenograma_sources.py
\src\bronze\build_ghu_maestro_horas.py
\src\bronze\build_personal_sources.py
\src\bronze\build_ventas_sources.py
\src\bronze\build_weather_hour_a4.py
\src\bronze\build_weather_hour_main.py
\src\common\ids.py
\src\common\io.py
\src\common\timegrid.py
\src\eval\build_kpis_ajuste_poscosecha_ml2.py
\src\eval\build_kpis_desp_poscosecha_ml2.py
\src\eval\build_kpis_dh_poscosecha_ml2.py
\src\eval\build_kpis_harvest_horizon_ml2.py
\src\eval\build_kpis_harvest_start_ml2.py
\src\eval\build_kpis_hidr_poscosecha_ml2.py
\src\eval\build_kpis_peso_tallo_ml2.py
\src\eval\build_kpis_poscosecha_ml2_baseline.py
\src\eval\build_kpis_share_grado_ml2.py
\src\eval\build_kpis_tallos_curve_ml2.py
\src\features\build_cap_tallos_real_dia.py
\src\features\build_features_ciclo_fecha.py
\src\features\build_features_cosecha_bloque_fecha.py
\src\features\build_features_curva_cosecha_bloque_dia.py
\src\features\build_features_harvest_window_ml1.py
\src\features\build_features_peso_tallo_grado_bloque_dia.py
\src\features\build_targets_curva_beta_multiplier_dia.py
\src\features\build_targets_curva_beta_params.py
\src\gold\apply_desp_ajuste_poscosecha_ml1_on_ml2_hidr.py
\src\gold\apply_dh_poscosecha_ml1_on_ml2_seed.py
\src\gold\apply_hidr_poscosecha_ml1_on_ml2_dh.py
\src\gold\build_ds_ajuste_poscosecha_ml2_v1.py
\src\gold\build_ds_desp_poscosecha_ml2_v1.py
\src\gold\build_ds_dh_poscosecha_ml2_v1.py
\src\gold\build_ds_harvest_horizon_ml2_v2.py
\src\gold\build_ds_harvest_start_ml2_v2.py
\src\gold\build_ds_hidr_poscosecha_ml2_v1.py
\src\gold\build_ds_peso_tallo_ml2_v1.py
\src\gold\build_ds_share_grado_ml2_v1.py
\src\gold\build_ds_tallos_curve_ml2_v2.py
\src\gold\build_pred_cajas_postcosecha_seed_mix_grado_dia.py
\src\gold\build_pred_kg_cajas_grado_dia_ml1_full.py
\src\gold\build_pred_kg_cajas_ml2.py
\src\gold\build_pred_poscosecha_ml1_views.py
\src\gold\build_pred_poscosecha_ml2_final_views.py
```

```
\src\gold\build_pred_poscosecha_ml2_seed_mix_grado_dia.py
\src\gold\build_pred_poscosecha_ml2_views_from_full.py
\src\gold\build_pred_tallos_grado_dia_ml1.py
\src\gold\build_pred_tallos_grado_dia_ml1_full.py
\src\gold\build_pred_tallos_ml2_full.py
\src\gold\build_universe_harvest_grid_ml1.py
\src\gold\build_universe_harvest_grid_ml2.py
\src\gold\build_view_planificacion_campo_tallos.py
\src\gold\postprocess_curva_share_smooth_ml1.py
\src\models\ml1\apply_curva_beta_params.py
\src\models\ml1\apply_curva_cdf_dia.py
\src\models\ml1\apply_curva_share_dia.py
\src\models\ml1\apply_curva_tallos_dia.py
\src\models\ml1\apply_desp_ajuste_poscosecha_ml1.py
\src\models\ml1\apply_dh_poscosecha_ml1.py
\src\models\ml1\apply_dist_grado.py
\src\models\ml1\apply_harvest_window_ml1.py
\src\models\ml1\apply_hidr_poscosecha_ml1.py
\src\models\ml1\apply_peso_tallo_grado.py
\src\models\ml1\train_ajuste_poscosecha_ml1.py
\src\models\ml1\train_curva_beta_multiplier_dia.py
\src\models\ml1\train_curva_beta_params.py
\src\models\ml1\train_curva_cdf_dia.py
\src\models\ml1\train_curva_share_dia.py
\src\models\ml1\train_curva_tallos_dia.py
\src\models\ml1\train_desp_poscosecha_ml1.py
\src\models\ml1\train_dh_poscosecha_ml1.py
\src\models\ml1\train_dist_grado.py
\src\models\ml1\train_harvest_window_ml1.py
\src\models\ml1\train_hidr_poscosecha_ml1.py
\src\models\ml1\train_peso_tallo_grado.py
\src\models\ml2\apply_ajuste_poscosecha_ml2.py
\src\models\ml2\apply_desp_poscosecha_ml2.py
\src\models\ml2\apply_dh_poscosecha_ml2.py
\src\models\ml2\apply_harvest_horizon_ml2.py
\src\models\ml2\apply_harvest_start_ml2.py
\src\models\ml2\apply_hidr_poscosecha_ml2.py
\src\models\ml2\apply_peso_tallo_ml2.py
\src\models\ml2\apply_share_grado_ml2.py
\src\models\ml2\apply_tallos_curve_ml2.py
\src\models\ml2\train_ajuste_poscosecha_ml2.py
\src\models\ml2\train_desp_poscosecha_ml2.py
\src\models\ml2\train_dh_poscosecha_ml2.py
\src\models\ml2\train_harvest_horizon_ml2.py
\src\models\ml2\train_harvest_start_ml2.py
\src\models\ml2\train_hidr_poscosecha_ml2.py
\src\models\ml2\train_peso_tallo_ml2.py
\src\models\ml2\train_share_grado_ml2.py
\src\models\ml2\train_tallos_curve_ml2.py
\src\ops\io.py
\src\ops\registry.py
\src\ops\runner.py
\src\ops\validators.py
\src\preds\build_capacidad_cosecha_dia.py
\src\preds\build_pred_cajas_from_peso_final.py
\src\preds\build_pred_horas_poscosecha.py
\src\preds\build_pred_milestones_baseline.py
\src\preds\build_pred_oferta_dia.py
\src\preds\build_pred_oferta_dia_from_universe_ml1.py
\src\preds\build_pred_oferta_grado.py
\src\preds\build_pred_peso_final_ajustado_grado.py
\src\preds\build_pred_peso_grado.py
\src\preds\build_pred_peso_hidratado_grado.py
\src\preds\build_pred_plan_horas_dia.py
\src\preds\build_pred_tallos_cosecha_dia.py
\src\run_pipeline.py
\src\silver\build_ciclo_maestro_from_fenograma.py
\src\silver\build_ciclo_maestro_from_sources.py
\src\silver\build_dim_baseline_capacidad_tallos_h_persona.py
\src\silver\build_dim_capacidad_baseline_tallos_proceso.py
\src\silver\build_dim_clima_bloque_dia.py
\src\silver\build_dim_cosecha_progress_bloque_fecha.py
```

```

\src\silver\build_dim_dh_baseline_grado_destino.py
\src\silver\build_dim_dist_grado_baseline.py
\src\silver\build_dim_estado_termico_cultivo_bloque_fecha.py
\src\silver\build_dim_factor_uph_cosecha_clima.py
\src\silver\build_dim_hidratacion_baseline_grado_destino.py
\src\silver\build_dim_hidratacion_fecha_cosecha_grado_destino.py
\src\silver\build_dim_mediana_etapas_tipo_sp_variedad_area.py
\src\silver\build_dim_mermas_ajuste_fecha_post.py
\src\silver\build_dim_mix_proceso_semana_from_ventas.py
\src\silver\build_dim_peso_tallo_baseline.py
\src\silver\build_dim_peso_tallo_promedio_dia.py
\src\silver\build_dim_variedad_canon.py
\src\silver\build_fact_capacidad_proceso_hist.py
\src\silver\build_fact_cosecha_real_grado_dia.py
\src\silver\build_fact_cosecha_uph_hora_clima.py
\src\silver\build_fact_peso_tallo_real_grado_dia.py
\src\silver\build_hidratacion_real_from_balanza2.py
\src\silver\build_milestone_window_ciclo_final_with_inference.py
\src\silver\build_milestones_final.py
\src\silver\build_milestones_windows.py
\src\silver\build_patch_ciclo_maestro_from_pred_tallos.py
\src\silver\build_weather_hour_estado.py
\src\silver\build_weather_hour_wide.py
\src\silver\build_windows_from_milestones_final.py

```

=====

```

-----
[1/164] FILE: \src\audit\audit_ajuste_poscosecha_ml2.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

```

```

from src.common.io import read_parquet, write_parquet

```

```

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

```

```

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"

```

```

IN_G = EVAL / "ml2_ajuste_poscosecha_eval_global.parquet"
IN_D = EVAL / "ml2_ajuste_poscosecha_eval_by_destino.parquet"
IN_DIST = EVAL / "ml2_ajuste_poscosecha_eval_ratio_dist.parquet"

```

```

# Re-lectura de final para ejemplos
GOLD = DATA / "gold"
SILVER = DATA / "silver"
IN_FINAL = GOLD / "pred_poscosecha_ml2_ajuste_grado_dia_bloque_destino_final.parquet"
IN_REAL_MA = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

```

```

def _ts() -> str:
    return datetime.utcnow().strftime("%Y%m%d_%H%M%S")

```

```

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

```

```

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

```

```

def main() -> None:

```

```

ts = _ts()

out_g = EVAL / f"audit_ajuste_poscosecha_ml2_kpi_global_{ts}.parquet"
out_d = EVAL / f"audit_ajuste_poscosecha_ml2_kpi_by_destino_{ts}.parquet"
out_dist = EVAL / f"audit_ajuste_poscosecha_ml2_ratio_dist_{ts}.parquet"
out_ex = EVAL / f"audit_ajuste_poscosecha_ml2_examples_10x2_{ts}.parquet"

g = read_parquet(IN_G).copy()
d = read_parquet(IN_D).copy()
dist = read_parquet(IN_DIST).copy()

# Examples: top improve / worst by |log_ratio_ml1|-|log_ratio_ml2|
df = read_parquet(IN_FINAL).copy()
df.columns = [str(c).strip() for c in df.columns]
df["destino"] = _canon_str(df["destino"])

# real
real = read_parquet(IN_REAL_MA).copy()
real.columns = [str(c).strip() for c in real.columns]
real["fecha_post"] = _to_date(real["fecha_post"])
real["destino"] = _canon_str(real["destino"])
if "factor_ajuste" in real.columns:
    rc = "factor_ajuste"
elif "ajuste" in real.columns:
    rc = "ajuste"
else:
    rc = None
if rc is None:
    raise ValueError("Real MA no trae factor_ajuste/ajuste.")
real[rc] = pd.to_numeric(real[rc], errors="coerce")
real2 = real.groupby(["fecha_post", "destino"], dropna=False, as_index=False).agg(factor_ajuste_real=(rc,
"median"))

# columnas
fecha_post_col = None
for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml1", "fecha_post_pred"]:
    if c in df.columns:
        fecha_post_col = c
        break
if fecha_post_col is None:
    raise KeyError("No encuentro fecha_post_pred_* en final.")
df[fecha_post_col] = _to_date(df[fecha_post_col])

aj_ml1_col = None
for c in ["factor_ajuste_ml1", "ajuste_ml1", "factor_ajuste_seed", "factor_ajuste"]:
    if c in df.columns:
        aj_ml1_col = c
        break
if aj_ml1_col is None:
    raise KeyError("No encuentro ajuste ML1 en final.")
df[aj_ml1_col] = pd.to_numeric(df[aj_ml1_col], errors="coerce")

df["factor_ajuste_final"] = pd.to_numeric(df["factor_ajuste_final"], errors="coerce")

df = df.merge(
    real2.rename(columns={"fecha_post": "fecha_post_key"}),
    left_on=[fecha_post_col, "destino"],
    right_on=["fecha_post_key", "destino"],
    how="left",
)

m = df["factor_ajuste_real"].notna() & df[aj_ml1_col].notna() & df["factor_ajuste_final"].notna()
x = df.loc[m].copy()
if x.empty:
    # igual guardamos los KPI y dist
    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(g, out_g)
    write_parquet(d, out_d)
    write_parquet(dist, out_dist)
    write_parquet(pd.DataFrame([], out_ex))
    print("\n=== ML2 AJUSTE POSCOSECHA AUDIT ===")
    print(f"KPI global parquet : {out_g}")

```

```

print(f"KPI por destino      : {out_d}")
print(f"Ratio dist parquet : {out_dist}")
print(f"Examples 10x2       : {out_ex}")
print("\n[WARN] No hay filas con real para examples.")
return

# pesos (si existen)
w = None
for c in ["tallos_w", "tallos", "tallos_total_ml2", "tallos_total"]:
    if c in x.columns:
        w = pd.to_numeric(x[c], errors="coerce").fillna(0.0)
        break
if w is None:
    w = pd.Series(1.0, index=x.index, dtype="float64")

ratio_ml1 = (x[aj_ml1_col] / x["factor_ajuste_real"].replace(0, np.nan)).astype(float)
ratio_ml2 = (x["factor_ajuste_final"] / x["factor_ajuste_real"].replace(0, np.nan)).astype(float)

lr1 = np.log(ratio_ml1.replace(0, np.nan))
lr2 = np.log(ratio_ml2.replace(0, np.nan))
score = lr1.abs() - lr2.abs() # >0 mejora
wscore = score * w

x["ratio_ml1"] = ratio_ml1
x["ratio_ml2"] = ratio_ml2
x["log_ratio_ml1"] = lr1
x["log_ratio_ml2"] = lr2
x["score"] = score
x["wscore"] = wscore

cols = [
    "fecha",
    "fecha_post_col",
    "destino",
    "grado" if "grado" in x.columns else None,
    aj_ml1_col,
    "factor_ajuste_final",
    "factor_ajuste_real",
    "ratio_ml1",
    "ratio_ml2",
    "log_ratio_ml1",
    "log_ratio_ml2",
    "score",
    "wscore",
]
cols = [c for c in cols if c is not None and c in x.columns]
top = x.sort_values("wscore", ascending=False).head(10).copy()
top["sample_group"] = "TOP_IMPROVE_10"
worst = x.sort_values("wscore", ascending=True).head(10).copy()
worst["sample_group"] = "TOP_WORSE_10"
ex = pd.concat([top[cols + ["sample_group"]], worst[cols + ["sample_group"]]], ignore_index=True)

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(g, out_g)
write_parquet(d, out_d)
write_parquet(dist, out_dist)
write_parquet(ex, out_ex)

print("\n== ML2 AJUSTE POSCOSECHA AUDIT ==")
print(f"KPI global parquet : {out_g}")
print(f"KPI por destino      : {out_d}")
print(f"Ratio dist parquet : {out_dist}")
print(f"Examples 10x2       : {out_ex}")
print("\n--- KPI GLOBAL ---")
print(g.to_string(index=False))
print("\n--- RATIO DIST ---")
print(dist.to_string(index=False))
print("\n--- KPI POR DESTINO ---")
print(d.to_string(index=False))
print("\n--- EXAMPLES 10x2 ---")
print(ex.to_string(index=False))

```

```

if __name__ == "__main__":
    main()

-----
[2/164] FILE: \src\audit\audit_desp_poscosecha_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime

import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

IN_FINAL = GOLD / "pred_poscosecha_ml2_desp_grado_dia_bloque_destino_final.parquet"
IN_REAL = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

TS = datetime.now().strftime("%Y%m%d_%H%M%S")

OUT_KPI_G = EVAL / f"audit_desp_poscosecha_ml2_kpi_global_{TS}.parquet"
OUT_KPI_D = EVAL / f"audit_desp_poscosecha_ml2_kpi_by_destino_{TS}.parquet"
OUT_DIST = EVAL / f"audit_desp_poscosecha_ml2_ratio_dist_{TS}.parquet"
OUT_EX = EVAL / f"audit_desp_poscosecha_ml2_examples_10x2_{TS}.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def wmae_log_ratio(log_ratio: pd.Series, w: pd.Series) -> float:
    x = pd.to_numeric(log_ratio, errors="coerce")
    ww = pd.to_numeric(w, errors="coerce").fillna(0.0)
    m = x.notna() & (ww > 0)
    if not bool(m.any()):
        return float("nan")
    denom = float(ww[m].sum())
    if denom <= 0:
        return float(np.nanmean(np.abs(x[m].values)))
    return float((np.abs(x[m].values) * ww[m].values).sum() / denom)

def main() -> None:
    print("\n=== ML2 DESP POSCOSECHA AUDIT ===")

    df = read_parquet(IN_FINAL).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha_post_pred_used", "destino", "factor_desp_ml1", "factor_desp_final"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"Final sin columnas: {sorted(miss)}")

    df["fecha_post_pred_used"] = _to_date(df["fecha_post_pred_used"])

```

```

df["destino"] = _canon_str(df["destino"])
df["factor_desp_ml1"] = pd.to_numeric(df["factor_desp_ml1"], errors="coerce")
df["factor_desp_final"] = pd.to_numeric(df["factor_desp_final"], errors="coerce")

if "tallos_w" in df.columns:
    w = pd.to_numeric(df["tallos_w"], errors="coerce")
elif "tallos" in df.columns:
    w = pd.to_numeric(df["tallos"], errors="coerce")
else:
    w = pd.Series(1.0, index=df.index)

df["w"] = w.fillna(0.0)

real = read_parquet(IN_REAL).copy()
real.columns = [str(c).strip() for c in real.columns]

need_r = {"fecha_post", "destino", "factor_desp"}
miss_r = need_r - set(real.columns)
if miss_r:
    raise ValueError(f"Real sin columnas: {sorted(miss_r)}")

real["fecha_post"] = _to_date(real["fecha_post"])
real["destino"] = _canon_str(real["destino"])
real["factor_desp_real"] = pd.to_numeric(real["factor_desp"], errors="coerce")

real2 = (
    real.groupby(["fecha_post", "destino"], dropna=False, as_index=False)
    .agg(factor_desp_real=("factor_desp_real", "median"))
)

x = df.merge(
    real2,
    left_on=["fecha_post_pred_used", "destino"],
    right_on=["fecha_post", "destino"],
    how="left",
)

eps = 1e-12
x["ratio_ml1"] = (x["factor_desp_real"] + eps) / (x["factor_desp_ml1"] + eps)
x["ratio_ml2"] = (x["factor_desp_real"] + eps) / (x["factor_desp_final"] + eps)
x["log_ratio_ml1"] = np.log(x["ratio_ml1"])
x["log_ratio_ml2"] = np.log(x["ratio_ml2"])

m = x["factor_desp_real"].notna() & x["factor_desp_ml1"].notna() & x["factor_desp_final"].notna()
d = x.loc[m].copy()

out_g = pd.DataFrame(
    [
        {
            "n_rows": int(len(d)),
            "n_dates": int(d["fecha_post_pred_used"].nunique()),
            "mae_log_ml1_w": wmae_log_ratio(d["log_ratio_ml1"], d["w"]),
            "mae_log_ml2_w": wmae_log_ratio(d["log_ratio_ml2"], d["w"]),
            "median_ratio_ml1": float(np.nanmedian(d["ratio_ml1"].values)),
            "median_ratio_ml2": float(np.nanmedian(d["ratio_ml2"].values)),
            "improvement_abs_mae_log_w": wmae_log_ratio(d["log_ratio_ml1"], d["w"]) - wmae_log_ratio(d["log_ratio_ml2"], d["w"]),
            "created_at": pd.Timestamp(datetime.now()).normalize(),
        }
    ]
)

rows = []
for dest, g in d.groupby("destino"):
    rows.append(
        {
            "destino": str(dest),
            "n_rows": int(len(g)),
            "n_dates": int(g["fecha_post_pred_used"].nunique()),
            "mae_log_ml1_w": wmae_log_ratio(g["log_ratio_ml1"], g["w"]),
            "mae_log_ml2_w": wmae_log_ratio(g["log_ratio_ml2"], g["w"]),
        }
    )

```

```

        "median_ratio_ml2": float(np.nanmedian(g["ratio_ml2"].values)),
        "improvement_abs_mae_log_w": wmae_log_ratio(g["log_ratio_ml1"], g["w"]) - wmae_log_ratio(g["log_ratio_ml2"], g["w"]),
    }
)
out_d = pd.DataFrame(rows).sort_values("improvement_abs_mae_log_w", ascending=False)

dist = pd.DataFrame(
    [
        {
            "n": int(len(d)),
            "ratio_min": float(np.nanmin(d["ratio_ml2"].values)),
            "ratio_p05": float(np.nanpercentile(d["ratio_ml2"].values, 5)),
            "ratio_p25": float(np.nanpercentile(d["ratio_ml2"].values, 25)),
            "ratio_median": float(np.nanmedian(d["ratio_ml2"].values)),
            "ratio_p75": float(np.nanpercentile(d["ratio_ml2"].values, 75)),
            "ratio_p95": float(np.nanpercentile(d["ratio_ml2"].values, 95)),
            "ratio_max": float(np.nanmax(d["ratio_ml2"].values)),
            "created_at": pd.Timestamp(datetime.now()).normalize(),
        }
    ]
)

# Examples: score = abs(log_ml1) - abs(log_ml2), ponderado por w
d["score"] = d["log_ratio_ml1"].abs() - d["log_ratio_ml2"].abs()
d["wscore"] = d["score"] * d["w"].fillna(0.0)

best = d.sort_values("wscore", ascending=False).head(10).copy()
worst = d.sort_values("wscore", ascending=True).head(10).copy()

best["sample_group"] = "TOP_IMPROVE_10"
worst["sample_group"] = "TOP_WORSE_10"

ex = pd.concat([best, worst], ignore_index=True)
ex_out = ex[
    [
        "fecha",
        "fecha_post_pred_used",
        "destino",
        "w",
        "factor_desp_ml1",
        "factor_desp_final",
        "factor_desp_real",
        "ratio_ml1",
        "ratio_ml2",
        "log_ratio_ml1",
        "log_ratio_ml2",
        "score",
        "wscore",
        "sample_group",
    ]
].copy()

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(out_g, OUT_KPI_G)
write_parquet(out_d, OUT_KPI_D)
write_parquet(dist, OUT_DIST)
write_parquet(ex_out, OUT_EX)

print(f"KPI global parquet : {OUT_KPI_G}")
print(f"KPI por destino      : {OUT_KPI_D}")
print(f"Ratio dist parquet : {OUT_DIST}")
print(f"Examples 10x2        : {OUT_EX}")

print("\n--- KPI GLOBAL ---")
print(out_g.to_string(index=False))
print("\n--- RATIO DIST ---")
print(dist.to_string(index=False))
print("\n--- KPI POR DESTINO ---")
print(out_d.to_string(index=False))
print("\n--- EXAMPLES 10x2 ---")
print(ex_out.to_string(index=False))

```



```

if __name__ == "__main__":
    main()

-----
[3/164] FILE: \src\audit\audit_dh_poscosecha_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime

import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"

IN_GLOBAL = EVAL / "ml2_dh_poscosecha_eval_global.parquet"
IN_BY_DEST = EVAL / "ml2_dh_poscosecha_eval_by_destino.parquet"
IN_BY_GR = EVAL / "ml2_dh_poscosecha_eval_by_grado.parquet"
IN_DIST = EVAL / "ml2_dh_poscosecha_eval_delta_dist.parquet"

# Para ejemplos: usamos el backtest factor + KPIs recomputados localmente
IN_FACTOR = EVAL / "backtest_factor_ml2_dh_poscosecha.parquet"
IN_REAL = ROOT / "data" / "silver" / "fact_hidratacion_real_post_grado_destino.parquet"

OUT_KPI_GLOBAL = EVAL / f"audit_dh_poscosecha_ml2_kpi_global_{datetime.now().strftime('%Y%m%d_%H%M%S')}.parquet"
OUT_KPI_BY_DEST = EVAL / f"audit_dh_poscosecha_ml2_kpi_by_destino_{datetime.now().strftime('%Y%m%d_%H%M%S')}.parquet"
OUT_KPI_BY_GR = EVAL / f"audit_dh_poscosecha_ml2_kpi_by_grado_{datetime.now().strftime('%Y%m%d_%H%M%S')}.parquet"
OUT_DIST = EVAL / f"audit_dh_poscosecha_ml2_delta_dist_{datetime.now().strftime('%Y%m%d_%H%M%S')}.parquet"
OUT_EX = EVAL / f"audit_dh_poscosecha_ml2_examples_10x2_{datetime.now().strftime('%Y%m%d_%H%M%S')}.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def main() -> None:
    print("\n=== ML2 DH POSCOSECHA AUDIT ===")

    g = read_parquet(IN_GLOBAL).copy()
    d = read_parquet(IN_BY_DEST).copy()
    gr = read_parquet(IN_BY_GR).copy()
    dist = read_parquet(IN_DIST).copy()

    # Examples
    fac = read_parquet(IN_FACTOR).copy()
    real = read_parquet(IN_REAL).copy()
    fac.columns = [str(c).strip() for c in fac.columns]
    real.columns = [str(c).strip() for c in real.columns]

```

```

fac["fecha"] = _to_date(fac["fecha"])
fac["destino"] = _canon_str(fac["destino"])
fac["grado"] = _canon_int(fac["grado"])
fac["dh_ml1"] = _canon_int(fac.get("dh_ml1"))
fac["dh_dias_final"] = _canon_int(fac.get("dh_dias_final"))

real["fecha_cosecha"] = _to_date(real["fecha_cosecha"])
real["destino"] = _canon_str(real["destino"])
real["grado"] = _canon_int(real["grado"])
real["tallos"] = pd.to_numeric(real.get("tallos"), errors="coerce").fillna(0.0)
real["dh_dias"] = _canon_int(real.get("dh_dias"))

real_g = (
    real.groupby(["fecha_cosecha", "grado", "destino"], dropna=False, as_index=False)
        .agg(dh_real=("dh_dias", "median"), tallos=("tallos", "sum"))
)

ex = fac.merge(real_g, left_on=["fecha", "grado", "destino"], right_on=["fecha_cosecha", "grado", "destino"], how="left")
ex = ex[ex["dh_real"].notna() & ex["dh_ml1"].notna() & ex["dh_dias_final"].notna()].copy()

ex["err_ml1_days"] = (ex["dh_real"].astype(float) - ex["dh_ml1"].astype(float))
ex["err_ml2_days"] = (ex["dh_real"].astype(float) - ex["dh_dias_final"].astype(float))
ex["abs_err_ml1"] = ex["err_ml1_days"].abs()
ex["abs_err_ml2"] = ex["err_ml2_days"].abs()
ex["improvement_abs"] = ex["abs_err_ml1"] - ex["abs_err_ml2"]

# filtrar tallos>0 para evitar ?TOP_BEST? falsos
ex["tallos"] = pd.to_numeric(ex.get("tallos"), errors="coerce").fillna(0.0)
ex = ex[ex["tallos"] > 0].copy()

# Score ponderado por impacto
ex["wscore"] = ex["improvement_abs"] * ex["tallos"]

top = ex.sort_values("wscore", ascending=False).head(10).copy()
top["sample_group"] = "TOP_IMPROVE_10"
worst = ex.sort_values("wscore", ascending=True).head(10).copy()
worst["sample_group"] = "TOP_WORSE_10"
examples = pd.concat([top, worst], ignore_index=True)

# Outputs
EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(g, OUT_KPI_GLOBAL)
write_parquet(d, OUT_KPI_BY_DEST)
write_parquet(gr, OUT_KPI_BY_GR)
write_parquet(dist, OUT_DIST)
write_parquet(examples, OUT_EX)

print(f"KPI global parquet : {OUT_KPI_GLOBAL}")
print(f"KPI por destino : {OUT_KPI_BY_DEST}")
print(f"KPI por grado : {OUT_KPI_BY_GR}")
print(f"Delta dist parquet : {OUT_DIST}")
print(f"Examples 10x2 : {OUT_EX}")

print("\n--- KPI GLOBAL ---")
print(g.to_string(index=False))
print("\n--- DELTA DIST ---")
print(dist.to_string(index=False))
print("\n--- KPI DESTINO ---")
print(d.to_string(index=False))
print("\n--- KPI GRADO (head) ---")
print(gr.head(10).to_string(index=False))
print("\n--- EXAMPLES (10x2) ---")
keep = [c for c in ["fecha", "grado", "destino", "tallos", "dh_ml1", "dh_dias_final", "dh_real", "err_ml1_days", "err_ml2_days", "improvement_abs", "wscore", "sample_group"] if c in examples.columns]
print(examples[keep].to_string(index=False))

if __name__ == "__main__":
    main()

```

```
-----  
[4/164] FILE: \src\audit\audit_dist_grado_m11.py  
-----
```

```
from __future__ import annotations
```

```
from pathlib import Path  
import json  
import numpy as np  
import pandas as pd
```

```
from common.io import read_parquet, write_parquet
```

```
IN_DIST = Path("data/gold/pred_dist_grado_m11.parquet")  
IN_GRID = Path("data/gold/universe_harvest_grid_m11.parquet")
```

```
OUT_AUDIT = Path("data/audit/audit_dist_grado_m11_checks.parquet")  
OUT_SUMMARY = Path("data/audit/audit_dist_grado_m11_summary.json")
```

```
def _to_date(s: pd.Series) -> pd.Series:  
    return pd.to_datetime(s, errors="coerce").dt.normalize()
```

```
def _canon_int(s: pd.Series) -> pd.Series:  
    return pd.to_numeric(s, errors="coerce").astype("Int64")
```

```
def _canon_str(s: pd.Series) -> pd.Series:  
    return s.astype(str).str.upper().str.strip()
```

```
def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:  
    miss = [c for c in cols if c not in df.columns]  
    if miss:  
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")
```

```
def main() -> None:  
    created_at = pd.Timestamp.utcnow()
```

```
    dist = read_parquet(IN_DIST).copy()  
    grid = read_parquet(IN_GRID).copy()
```

```
    dist.columns = [str(c).strip() for c in dist.columns]  
    grid.columns = [str(c).strip() for c in grid.columns]
```

```
    _require(dist, ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_m11"], "pred_d  
ist_grado_m11")
```

```
    _require(grid, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_m11")
```

```
    # Canon  
    dist["ciclo_id"] = dist["ciclo_id"].astype(str)  
    dist["fecha"] = _to_date(dist["fecha"])  
    dist["bloque_base"] = _canon_int(dist["bloque_base"])  
    dist["variedad_canon"] = _canon_str(dist["variedad_canon"])  
    dist["grado"] = _canon_int(dist["grado"])  
    dist["share_grado_m11"] = pd.to_numeric(dist["share_grado_m11"], errors="coerce")
```

```
    grid["ciclo_id"] = grid["ciclo_id"].astype(str)  
    grid["fecha"] = _to_date(grid["fecha"])  
    grid["bloque_base"] = _canon_int(grid["bloque_base"])  
    grid["variedad_canon"] = _canon_str(grid["variedad_canon"])
```

```
    dist_key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"]  
    grp_dist = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
```

```
    dup_count = int(dist.duplicated(subset=dist_key).sum())  
    dup_rate = float(dup_count / max(len(dist), 1))
```

```
    # sum shares  
    shares_sum = dist.groupby(grp_dist, dropna=False)["share_grado_m11"].sum()  
    # tolerancia: 1% (ajustable)  
    bad_sum = (shares_sum - 1.0).abs() > 0.01  
    bad_sum_rate = float(bad_sum.mean()) if len(shares_sum) else float("nan")
```

```

# cobertura dist vs grid
dist_groups = dist[grp_dist].drop_duplicates()
grid_groups = grid[grp_dist].drop_duplicates()
cov = float(
    grid_groups.merge(dist_groups.assign(has=1), on=grp_dist, how="left")["has"].fillna(0).mean()
)

rows = [
    {"metric": "rows_dist", "value": int(len(dist)), "level": "INFO", "hint": "", "created_at": created_at},
],
    {"metric": "dup_count_dist_key", "value": dup_count, "level": "WARN" if dup_count > 0 else "OK", "hint": "Duplicados por llave real (ciclo, fecha, bloque, variedad, grado).", "created_at": created_at},
    {"metric": "dup_rate_dist_key", "value": dup_rate, "level": "WARN" if dup_rate > 0 else "OK", "hint": "", "created_at": created_at},
    {"metric": "coverage_dist_vs_grid_groups", "value": cov, "level": "OK" if cov >= 0.999 else "WARN", "hint": "Grupos día del universe cubiertos por dist.", "created_at": created_at},
    {"metric": "bad_share_sum_rate_abs_gt_0.01", "value": bad_sum_rate, "level": "OK" if (np.isfinite(bad_sum_rate) and bad_sum_rate <= 0.01) else "WARN", "hint": "Proporción de grupos donde sum(shares) se aleja de 1.", "created_at": created_at},
]
audit = pd.DataFrame(rows)

OUT_AUDIT.parent.mkdir(parents=True, exist_ok=True)
write_parquet(audit, OUT_AUDIT)

summary = {
    "created_at_utc": created_at.isoformat(),
    "dist_key": dist_key,
    "grp_dist": grp_dist,
    "rows_dist": int(len(dist)),
    "dup_count": dup_count,
    "dup_rate": dup_rate,
    "coverage_dist_vs_grid_groups": cov,
    "bad_share_sum_rate_abs_gt_0.01": bad_sum_rate,
    "horizon": {
        "min_fecha": str(pd.to_datetime(dist["fecha"].min()).date()) if len(dist) else None,
        "max_fecha": str(pd.to_datetime(dist["fecha"].max()).date()) if len(dist) else None,
    },
}

OUT_SUMMARY.parent.mkdir(parents=True, exist_ok=True)
with open(OUT_SUMMARY, "w", encoding="utf-8") as f:
    json.dump(summary, f, ensure_ascii=False, indent=2)

print(f"OK -> {OUT_AUDIT}")
print(f"OK -> {OUT_SUMMARY}")
print(f"[AUDIT] dup_count={dup_count:}, dup_rate={dup_rate:.6f} coverage={cov:.6f} bad_sum_rate={bad_sum_rate:.6f}")

if __name__ == "__main__":
    main()

```

```

-----
[5/164] FILE: \src\audit\audit_harvest_horizon_ml2.py
-----

```

```

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()

```

```

DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"
IN_FACTOR = EVAL_DIR / "backtest_factor_ml2_harvest_horizon.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else float("nan")

def _bias(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(x)) if len(x) else float("nan")

def _pick_first_existing(df: pd.DataFrame, candidates: list[str]) -> str:
    for c in candidates:
        if c in df.columns:
            return c
    raise KeyError(f"None of these columns exist: {candidates}")

def main() -> None:
    ciclo = read_parquet(IN_CICLO).copy()
    fact = read_parquet(IN_FACTOR).copy()

    # Normalizar fechas en ciclo (autoridad de lo real)
    # Nota: ciclo puede tener nombres distintos según tu versión, cubrimos ambos.
    c_ini = _pick_first_existing(ciclo, ["fecha_inicio_cosecha", "fecha_ini_cosecha"])
    c_fin = _pick_first_existing(ciclo, ["fecha_fin_cosecha", "fecha_fin"])

    ciclo[c_ini] = _to_date(ciclo[c_ini])
    ciclo[c_fin] = _to_date(ciclo[c_fin])

    # Para evitar sufijos _x/_y: tomamos de ciclo solo lo que necesitamos
    cols = ["ciclo_id", c_ini, c_fin]
    if "estado" in ciclo.columns:
        cols.insert(1, "estado")
    ciclo_small = ciclo[cols].copy()

    ciclo_small = ciclo_small.rename(columns={c_ini: "fecha_inicio_cosecha_real", c_fin: "fecha_fin_cosecha_real"})

    df = ciclo_small.merge(fact, on="ciclo_id", how="inner")
    # Estado: puede venir en ciclo o en factor. Normalizamos y garantizamos que exista.
    if "estado" not in df.columns:
        if "estado_x" in df.columns:
            df["estado"] = df["estado_x"]
        elif "estado_y" in df.columns:
            df["estado"] = df["estado_y"]
        elif "estado_ml1" in df.columns:
            df["estado"] = df["estado_ml1"]
        else:
            df["estado"] = "UNKNOWN"

    df["estado"] = df["estado"].astype(str).str.upper().str.strip()

    # Real duration
    df["n_harvest_days_real"] = (df["fecha_fin_cosecha_real"] - df["fecha_inicio_cosecha_real"]).dt.days + 1

    # Errors (duración)
    df["err_ml1_days"] = pd.to_numeric(df["n_harvest_days_real"], errors="coerce") - pd.to_numeric(df["n_harvest_days_pred"], errors="coerce")
    df["err_ml2_days"] = pd.to_numeric(df["n_harvest_days_real"], errors="coerce") - pd.to_numeric(df["n_harvest_days_final"], errors="coerce")

```

```

df = df.loc[df["err_ml1_days"].notna() & df["err_ml2_days"].notna(), :].copy()

# Paths outputs con timestamp
ts = datetime.now().strftime("%Y%m%d_%H%M%S")
out_kpi_global = EVAL_DIR / f"audit_harvest_horizon_ml2_kpi_global_{ts}.parquet"
out_dist = EVAL_DIR / f"audit_harvest_horizon_ml2_adjust_dist_{ts}.parquet"
out_by_estado = EVAL_DIR / f"audit_harvest_horizon_ml2_kpi_by_estado_{ts}.parquet"
out_examples = EVAL_DIR / f"audit_harvest_horizon_ml2_examples_10x2_{ts}.parquet"

# KPI GLOBAL
kpi_global = pd.DataFrame([
    "n": int(len(df)),
    "mae_ml1_days": _mae(df["err_ml1_days"]),
    "mae_ml2_days": _mae(df["err_ml2_days"]),
    "bias_ml1_days": _bias(df["err_ml1_days"]),
    "bias_ml2_days": _bias(df["err_ml2_days"]),
    "improvement_abs_days": (_mae(df["err_ml1_days"]) - _mae(df["err_ml2_days"])),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
])
write_parquet(kpi_global, out_kpi_global)

# DIST AJUSTE
adj = pd.to_numeric(df["pred_error_horizon_days"], errors="coerce")
dist = pd.DataFrame([
    "n_factor_rows": int(adj.notna().sum()),
    "adj_min": float(np.nanmin(adj)) if len(adj) else np.nan,
    "adj_p25": float(np.nanpercentile(adj, 25)) if len(adj) else np.nan,
    "adj_median": float(np.nanmedian(adj)) if len(adj) else np.nan,
    "adj_p75": float(np.nanpercentile(adj, 75)) if len(adj) else np.nan,
    "adj_max": float(np.nanmax(adj)) if len(adj) else np.nan,
    "pct_clip": float(np.nanmean((adj <= -14) | (adj >= 21))) if len(adj) else np.nan,
])
write_parquet(dist, out_dist)

# KPI POR ESTADO
rows = []
for estado, g in df.groupby("estado"):
    rows.append({
        "estado": estado,
        "n": int(len(g)),
        "mae_ml1_days": _mae(g["err_ml1_days"]),
        "mae_ml2_days": _mae(g["err_ml2_days"]),
        "bias_ml1_days": _bias(g["err_ml1_days"]),
        "bias_ml2_days": _bias(g["err_ml2_days"]),
        "improvement_abs_days": (_mae(g["err_ml1_days"]) - _mae(g["err_ml2_days"])),
    })
by_estado = pd.DataFrame(rows).sort_values(["estado"])
write_parquet(by_estado, out_by_estado)

# EJEMPLOS 10x2: 10 CERRADO + 10 ACTIVO (si existen), priorizando mayor |err_ml1|
df["abs_err_ml1"] = pd.to_numeric(df["err_ml1_days"], errors="coerce").abs()
examples = (
    df.sort_values("abs_err_ml1", ascending=False)
    .groupby("estado", group_keys=False)
    .head(10)
    .copy()
)
write_parquet(examples, out_examples)

print("\n=== ML2 HARVEST HORIZON AUDIT ===")
print(f"KPI global parquet : {out_kpi_global}")
print(f"Dist ajustes parquet: {out_dist}")
print(f"KPI por estado      : {out_by_estado}")
print(f"Examples 10x2       : {out_examples}")

print("\n--- KPI GLOBAL ---")
print(kpi_global.to_string(index=False))
print("\n--- AJUSTE DIST ---")
print(dist.to_string(index=False))
print("\n--- KPI POR ESTADO ---")
print(by_estado.to_string(index=False))

```

```

print("\n--- EXAMPLES (head) ---")
cols_show = [c for c in [
    "ciclo_id", "estado",
    "fecha_inicio_cosecha_real", "fecha_fin_cosecha_real",
    "n_harvest_days_pred", "pred_error_horizon_days", "n_harvest_days_final",
    "err_ml1_days", "err_ml2_days",
    "ml1_version", "ml2_run_id",
] if c in examples.columns]
print(examples[cols_show].head(10).to_string(index=False))

if __name__ == "__main__":
    main()

-----
[6/164] FILE: \src\audit\audit_harvest_start_ml2.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    # .../src/eval/file.py -> repo_root = parents[2]
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"
IN_FACTOR = GOLD_DIR / "factors" / "factor_ml2_harvest_start.parquet"
IN_DS = GOLD_DIR / "ml2_datasets" / "ds_harvest_start_ml2_v2.parquet" # para features clima agregadas

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else float("nan")

def _bias(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(x)) if len(x) else float("nan")

def main() -> None:
    # ----- Load -----
    ciclo = read_parquet(IN_CICLO).copy()
    fact = read_parquet(IN_FACTOR).copy()

    # Dataset v2 (para features clima as_of); puede no existir en prod, así que lo protegemos
    ds = None
    if IN_DS.exists():
        ds = read_parquet(IN_DS).copy()

```

```

# ----- Canon / dates -----
ciclo["estado"] = _canon_str(ciclo["estado"])
ciclo["bloque_base"] = _canon_str(ciclo["bloque_base"])
ciclo["fecha_sp"] = _to_date(ciclo["fecha_sp"])
ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])

fact["bloque_base"] = _canon_str(fact["bloque_base"])
if "estado" in fact.columns:
    fact["estado"] = _canon_str(fact["estado"])

fact["fecha_sp"] = _to_date(fact["fecha_sp"])
fact["harvest_start_pred"] = _to_date(fact["harvest_start_pred"])
fact["harvest_start_final"] = _to_date(fact["harvest_start_final"])

# ----- Join core audit frame -----
df = ciclo.merge(
    fact,
    on="ciclo_id",
    how="inner",
    suffixes=("_real", "_ml2"),
)

# usar estado de ciclo maestro como autoridad
df["estado"] = df["estado_real"]

# ----- Errors -----
df["err_ml1_days"] = (df["fecha_inicio_cosecha"] - df["harvest_start_pred"]).dt.days
df["err_ml2_days"] = (df["fecha_inicio_cosecha"] - df["harvest_start_final"]).dt.days

# Algunas filas pueden no tener real (activos sin inicio real aún). Filtramos para KPI real.
df_eval = df.loc[df["fecha_inicio_cosecha"].notna() & df["harvest_start_pred"].notna() & df["harvest_start_final"].notna(), :].copy()

# ----- KPI global -----
mae_ml1 = _mae(df_eval["err_ml1_days"])
mae_ml2 = _mae(df_eval["err_ml2_days"])
bias_ml1 = _bias(df_eval["err_ml1_days"])
bias_ml2 = _bias(df_eval["err_ml2_days"])

kpi_global = pd.DataFrame([
    {
        "n": int(len(df_eval)),
        "mae_ml1_days": mae_ml1,
        "mae_ml2_days": mae_ml2,
        "bias_ml1_days": bias_ml1,
        "bias_ml2_days": bias_ml2,
        "improvement_abs_days": (mae_ml1 - mae_ml2) if (pd.notna(mae_ml1) and pd.notna(mae_ml2)) else np.nan,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }
])

# ----- Adjustment distribution -----
adj = pd.to_numeric(df.get("pred_error_start_days"), errors="coerce")
dist = pd.DataFrame([
    {
        "n_factor_rows": int(len(df)),
        "adj_min": float(np.nanmin(adj)) if len(adj) else np.nan,
        "adj_p25": float(np.nanpercentile(adj, 25)) if len(adj) else np.nan,
        "adj_median": float(np.nanmedian(adj)) if len(adj) else np.nan,
        "adj_p75": float(np.nanpercentile(adj, 75)) if len(adj) else np.nan,
        "adj_max": float(np.nanmax(adj)) if len(adj) else np.nan,
    }
])

# % clipping (asumimos guardrail ±21)
CLIP_LO, CLIP_HI = -21, 21
clip_mask = (adj <= CLIP_LO) | (adj >= CLIP_HI)
dist["pct_clip"] = float(np.nanmean(clip_mask)) if len(adj) else np.nan

# ----- KPI por estado (ABIERTO/CERRADO) -----
by_estado_rows = []
for est, g in df_eval.groupby("estado"):
    mae1 = _mae(g["err_ml1_days"])
    mae2 = _mae(g["err_ml2_days"])
    by_estado_rows.append({
        "estado": est,

```



```

        "n": int(len(g)),
        "mae_ml1_days": mae1,
        "mae_ml2_days": mae2,
        "bias_ml1_days": _bias(g["err_ml1_days"]),
        "bias_ml2_days": _bias(g["err_ml2_days"]),
        "improvement_abs_days": (mae1 - mae2) if (pd.notna(mae1) and pd.notna(mae2)) else np.nan,
    })
kpi_by_estado = pd.DataFrame(by_estado_rows).sort_values(["estado"])

# ----- ?Sentido agronómico? simple (si hay dataset con features) -----
# Join con ds (tiene gdc_cum_sp, rain_cum_sp, etc. a nivel ciclo_id+as_of_date).
# Para audit, tomamos el último as_of por ciclo (más cercano al inicio real).
agr = pd.DataFrame()
if ds is not None and len(ds):
    ds = ds.copy()
    ds["as_of_date"] = _to_date(ds["as_of_date"])
    # último as_of por ciclo
    ds_last = ds.sort_values(["ciclo_id", "as_of_date"]).groupby("ciclo_id", as_index=False).tail(1)
    # join
    wanted = ["ciclo_id", "gdc_cum_sp", "gdc_l4d", "rain_7d", "rain_cum_sp", "solar_7d", "temp_avg_7d"]
    cols = [c for c in wanted if c in ds_last.columns]
    agr = df.merge(ds_last[cols].copy(), on="ciclo_id", how="left")

# Correlaciones simples (no causal proof, solo sanity)
rows = []
feat_list = [c for c in ["gdc_cum_sp", "gdc_l4d", "rain_7d", "rain_cum_sp", "solar_7d", "temp_avg_7d"]]
if c in agr.columns:
    for col in feat_list:

        x = pd.to_numeric(agr[col], errors="coerce")
        y = pd.to_numeric(agr["pred_error_start_days"], errors="coerce")
        m = x.notna() & y.notna()
        if m.sum() >= 30:
            corr = float(np.corrcoef(x[m], y[m])[0, 1])
        else:
            corr = np.nan
        rows.append({"feature": col, "corr_with_pred_error_days": corr, "n": int(m.sum())})
    agr_corr = pd.DataFrame(rows)
else:
    agr_corr = pd.DataFrame([{"feature": None, "corr_with_pred_error_days": np.nan, "n": 0}])

# ----- 10 ejemplos ABIERTO y 10 CERRADO -----
# Muestra reproducible, y prioriza casos con mayor ajuste absoluto para que sea informativo.
def _pick_examples(est: str, n: int = 10) -> pd.DataFrame:
    g = df.copy()
    g = g.loc[g["estado"] == est, :].copy()
    g["abs_adj"] = pd.to_numeric(g["pred_error_start_days"], errors="coerce").abs()
    # si no hay suficientes, devuelve lo que haya
    g = g.sort_values(["abs_adj"], ascending=False).head(max(n * 3, n)) # pool
    # sample dentro del pool para variedad
    if len(g) > n:
        g = g.sample(n=n, random_state=42)
    cols = [
        "ciclo_id", "bloque_base_real", "variedad", "area", "tipo_sp",
        "estado",
        "fecha_sp_real", "fecha_inicio_cosecha",
        "harvest_start_pred", "harvest_start_final",
        "pred_error_start_days",
        "err_ml1_days", "err_ml2_days",
        "ml1_version", "ml2_run_id",
    ]
    # Algunas columnas tienen sufijos por el merge; normalizamos nombres:
    rename = {
        "bloque_base_real": "bloque_base",
        "fecha_sp_real": "fecha_sp",
    }
    out = g[[c for c in cols if c in g.columns]].rename(columns=rename)
    return out

ex_abierto = _pick_examples("ABIERTO", 10)
ex_cerrado = _pick_examples("CERRADO", 10)

```

```

examples = pd.concat(
    [
        ex_abierto.assign(sample_group="ABIERTO_10"),
        ex_cerrado.assign(sample_group="CERRADO_10"),
    ],
    ignore_index=True,
)

# ----- Save outputs -----
EVAL_DIR.mkdir(parents=True, exist_ok=True)
ts = datetime.now().strftime("%Y%m%d_%H%M%S")

out_kpi_global = EVAL_DIR / f"audit_harvest_start_ml2_kpi_global_{ts}.parquet"
out_dist = EVAL_DIR / f"audit_harvest_start_ml2_adjust_dist_{ts}.parquet"
out_by_estado = EVAL_DIR / f"audit_harvest_start_ml2_kpi_by_estado_{ts}.parquet"
out_corr = EVAL_DIR / f"audit_harvest_start_ml2_agro_corr_{ts}.parquet"
out_examples = EVAL_DIR / f"audit_harvest_start_ml2_examples_10x2_{ts}.parquet"

write_parquet(kpi_global, out_kpi_global)
write_parquet(dist, out_dist)
write_parquet(kpi_by_estado, out_by_estado)
write_parquet(agr_corr, out_corr)
write_parquet(examples, out_examples)

# CSV opcional (muy útil para revisar rápido)
examples.to_csv(EVAL_DIR / f"audit_harvest_start_ml2_examples_10x2_{ts}.csv", index=False)

# ----- Print summary -----
print("\n=== ML2 HARVEST START AUDIT ===")
print(f"KPI global parquet : {out_kpi_global}")
print(f"Dist ajustes parquet: {out_dist}")
print(f"KPI por estado      : {out_by_estado}")
print(f"Corr agronómica      : {out_corr}")
print(f"Examples 10x2        : {out_examples}")
print("\n--- KPI GLOBAL ---")
print(kpi_global.to_string(index=False))
print("\n--- AJUSTE DIST ---")
print(dist.to_string(index=False))
print("\n--- KPI POR ESTADO ---")
print(kpi_by_estado.to_string(index=False))
print("\n--- CORR (sanity) ---")
print(agr_corr.to_string(index=False))
print("\n--- EXAMPLES (top) ---")
print(examples.head(10).to_string(index=False))

if __name__ == "__main__":
    main()

-----
[7/164] FILE: \src\audit\audit_harvest_windows_real.py
-----

from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet

MAESTRO_PATH = Path("data/silver/fact_ciclo_maestro.parquet")

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

```

```

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _describe_series(x: pd.Series) -> str:
    x = pd.to_numeric(x, errors="coerce")
    if x.dropna().empty:
        return "EMPTY"
    q = x.quantile([0.01, 0.05, 0.10, 0.50, 0.90, 0.95, 0.99])
    desc = x.describe()
    lines = []
    lines.append(desc.to_string())
    lines.append("")
    lines.append("quantiles:")
    lines.append(q.to_string())
    return "\n".join(lines)

def main() -> None:
    df = read_parquet(MAESTRO_PATH).copy()
    print(f"OK read -> {MAESTRO_PATH} | rows={len(df):,}")

    # --- Requeridos según tu schema ---
    req = [
        "ciclo_id",
        "bloque_base",
        "variedad",
        "tipo_sp",
        "area",
        "fecha_sp",
        "fecha_inicio_cosecha",
        "fecha_fin_cosecha",
    ]
    _require(df, req, "fact_ciclo_maestro")

    # --- Canon ---
    df["ciclo_id"] = df["ciclo_id"].astype(str)
    df["bloque_base"] = _canon_int(df["bloque_base"])
    df["variedad"] = _canon_str(df["variedad"])
    df["tipo_sp"] = _canon_str(df["tipo_sp"])
    df["area"] = _canon_str(df["area"])

    df["fecha_sp"] = _to_date(df["fecha_sp"])
    df["harvest_start_real"] = _to_date(df["fecha_inicio_cosecha"])
    df["harvest_end_real"] = _to_date(df["fecha_fin_cosecha"])

    # --- Duplicados clave ---
    key = ["ciclo_id"]
    dup = int(df.duplicated(subset=key).sum())
    if dup > 0:
        print(f"[WARN] duplicated ciclo_id: {dup:,} (esto puede existir si hay varias líneas por ciclo; revisa)")
    else:
        print("[OK] ciclo_id único (sin duplicados)")

    # --- Coberturas ---
    cov = {
        "fecha_sp": float(df["fecha_sp"].notna().mean()),
        "harvest_start_real": float(df["harvest_start_real"].notna().mean()),
        "harvest_end_real": float(df["harvest_end_real"].notna().mean()),
        "bloque_base": float(df["bloque_base"].notna().mean()),
        "variedad": float(df["variedad"].notna().mean()),
        "tipo_sp": float(df["tipo_sp"].notna().mean()),
        "area": float(df["area"].notna().mean()),
    }

```

```

print("\n[COVERAGE]")
for k, v in cov.items():
    print(f"- {k}: {v:.4f}")

# --- Consistencia temporal ---
# (a) start >= sp
m1 = df["fecha_sp"].notna() & df["harvest_start_real"].notna()
bad_start_before_sp = float((df.loc[m1, "harvest_start_real"] < df.loc[m1, "fecha_sp"]).mean()) if m1.any()
else np.nan

# (b) end >= start
m2 = df["harvest_start_real"].notna() & df["harvest_end_real"].notna()
bad_end_before_start = float((df.loc[m2, "harvest_end_real"] < df.loc[m2, "harvest_start_real"]).mean()) if m2.any()
else np.nan

print("\n[CONSISTENCY]")
print(f"- % harvest_start_real < fecha_sp: {bad_start_before_sp:.4f}")
print(f"- % harvest_end_real < harvest_start_real: {bad_end_before_start:.4f}")

# --- Targets ML1 ---
# d_start_real = start - sp
df["d_start_real"] = (df["harvest_start_real"] - df["fecha_sp"]).dt.days
# n_harvest_days_real = end - start + 1
df["n_harvest_days_real"] = (df["harvest_end_real"] - df["harvest_start_real"]).dt.days + 1

print("\n[TARGET DISTRIBUTIONS: d_start_real (days)]")
print(_describe_series(df["d_start_real"]))

print("\n[TARGET DISTRIBUTIONS: n_harvest_days_real (days)]")
print(_describe_series(df["n_harvest_days_real"]))

# --- Outliers / casos imposibles ---
# define tolerancias duras (ajústalas si lo deseas)
# start negativo o demasiado alto
m3 = df["d_start_real"].notna()
bad_d_start_neg = float((df.loc[m3, "d_start_real"] < 0).mean()) if m3.any() else np.nan
bad_d_start_huge = float((df.loc[m3, "d_start_real"] > 365).mean()) if m3.any() else np.nan

m4 = df["n_harvest_days_real"].notna()
bad_n_days_le0 = float((df.loc[m4, "n_harvest_days_real"] <= 0).mean()) if m4.any() else np.nan
bad_n_days_huge = float((df.loc[m4, "n_harvest_days_real"] > 120).mean()) if m4.any() else np.nan

print("\n[OUTLIERS]")
print(f"- % d_start_real < 0: {bad_d_start_neg:.4f}")
print(f"- % d_start_real > 365: {bad_d_start_huge:.4f}")
print(f"- % n_harvest_days_real <= 0: {bad_n_days_le0:.4f}")
print(f"- % n_harvest_days_real > 120: {bad_n_days_huge:.4f}")

# --- Segmentación (clave para ver si ?medianas? de 30 rows era normal o no) ---
# Para cada (area,variedad,tipo_sp): conteo, mediana, IQR
seg_cols = ["area", "variedad", "tipo_sp"]
seg = (
    df.groupby(seg_cols, dropna=False, as_index=False)
    .agg(
        n=("ciclo_id", "nunique"),
        d_start_med=("d_start_real", "median"),
        d_start_p25=("d_start_real", lambda x: pd.to_numeric(x, errors="coerce").quantile(0.25)),
        d_start_p75=("d_start_real", lambda x: pd.to_numeric(x, errors="coerce").quantile(0.75)),
        n_days_med=("n_harvest_days_real", "median"),
        n_days_p25=("n_harvest_days_real", lambda x: pd.to_numeric(x, errors="coerce").quantile(0.25)),
        n_days_p75=("n_harvest_days_real", lambda x: pd.to_numeric(x, errors="coerce").quantile(0.75)),
    )
    .sort_values(["n"], ascending=False)
)

print("\n[SEGMENT SUMMARY TOP 30 by n]")
print(seg.head(30).to_string(index=False))

# --- Leakage sanity: ¿cuántos ciclos tienen real completo? ---
full = df["fecha_sp"].notna() & df["harvest_start_real"].notna() & df["harvest_end_real"].notna()
print("\n[TRAINABLE ROWS]")
print(f"- rows with full (sp,start,end): {int(full.sum()):,} / {len(df):,} = {float(full.mean()):.4f}")

```

```

# --- Recomendación automática de baseline vs ML ---
# Si trainable es muy bajo por segmento, el ML puede colapsar; te lo marca.
low = seg[seg["n"] < 20]
print("\n[WARN] segmentos con n < 20 (ML1 puede generalizar peor; usar regularización / pooling):")
print(low.head(30).to_string(index=False))

if __name__ == "__main__":
    main()

-----
[8/164] FILE: \src\audit\audit_hidr_poscosecha_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"

IN_GLOBAL = EVAL / "ml2_hidr_poscosecha_eval_global.parquet"
IN_BY_DESTINO = EVAL / "ml2_hidr_poscosecha_eval_by_destino.parquet"
IN_BY_GRADO = EVAL / "ml2_hidr_poscosecha_eval_by_grado.parquet"
IN_RATIO_DIST = EVAL / "ml2_hidr_poscosecha_eval_ratio_dist.parquet"

# Para ejemplos necesitamos final + real:
GOLD = DATA / "gold"
SILVER = DATA / "silver"
IN_FINAL = GOLD / "pred_poscosecha_ml2_hidr_grado_dia_bloque_destino_final.parquet"
IN_REAL = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _factor_from_hidr_pct(hidr_pct: pd.Series) -> pd.Series:
    x = pd.to_numeric(hidr_pct, errors="coerce")
    return np.where(x.isna(), np.nan, np.where(x > 3.5, 1.0 + x / 100.0, x)).astype(float)

def main() -> None:
    ts = datetime.now().strftime("%Y%m%d_%H%M%S")

    out_global = EVAL / f"audit_hidr_poscosecha_ml2_kpi_global_{ts}.parquet"
    out_dest = EVAL / f"audit_hidr_poscosecha_ml2_kpi_by_destino_{ts}.parquet"
    out_gr = EVAL / f"audit_hidr_poscosecha_ml2_kpi_by_grado_{ts}.parquet"
    out_dist = EVAL / f"audit_hidr_poscosecha_ml2_ratio_dist_{ts}.parquet"
    out_examples = EVAL / f"audit_hidr_poscosecha_ml2_examples_10x2_{ts}.parquet"

    g = read_parquet(IN_GLOBAL).copy()

```

```

bd = read_parquet(IN_BY_DESTINO).copy()
bg = read_parquet(IN_BY_GRADO).copy()
dist = read_parquet(IN_RATIO_DIST).copy()

# ejemplos
fin = read_parquet(IN_FINAL).copy()
fin.columns = [str(c).strip() for c in fin.columns]

# fecha_post_pred usada
fpp = None
for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml2", "fecha_post_pred_ml1", "
fecha_post_pred"]:
    if c in fin.columns:
        fpp = c
        break
if fpp is None:
    raise KeyError("No encuentro fecha_post_pred en final para ejemplos.")

fin[fpp] = _to_date(fin[fpp])
fin["destino"] = _canon_str(fin["destino"])
fin["grado"] = _canon_int(fin["grado"])
fin["factor_hidr_ml1"] = pd.to_numeric(fin.get("factor_hidr_ml1"), errors="coerce")
fin["factor_hidr_final"] = pd.to_numeric(fin.get("factor_hidr_final"), errors="coerce")

real = read_parquet(IN_REAL).copy()
real.columns = [str(c).strip() for c in real.columns]
if "hidr_pct" in real.columns:
    real["factor_hidr_real"] = _factor_from_hidr_pct(real["hidr_pct"])
else:
    pb = pd.to_numeric(real.get("peso_base_g"), errors="coerce")
    pp = pd.to_numeric(real.get("peso_post_g"), errors="coerce")
    real["factor_hidr_real"] = np.where(pb > 0, pp / pb, np.nan)

real["fecha_post"] = _to_date(real["fecha_post"])
real["destino"] = _canon_str(real["destino"])
real["grado"] = _canon_int(real["grado"])

if "tallos" in real.columns:
    real["tallos"] = pd.to_numeric(real["tallos"], errors="coerce").fillna(0.0)
    g2 = real.groupby(["fecha_post", "grado", "destino"], dropna=False)
    real2 = g2.apply(
        lambda x: pd.Series({
            "factor_hidr_real": float(np.nansum(x["factor_hidr_real"] * x["tallos"]) / np.nansum(x["tallos"]
        ))) if np.nansum(x["tallos"]) > 0 else float(np.nanmedian(x["factor_hidr_real"])),
        "tallos": float(np.nansum(x["tallos"])),
    })
    ).reset_index()
else:
    real2 = (
        real.groupby(["fecha_post", "grado", "destino"], dropna=False, as_index=False)
        .agg(factor_hidr_real=("factor_hidr_real", "median"))
    )
    real2["tallos"] = 1.0

df = fin.merge(
    real2,
    left_on=[fpp, "grado", "destino"],
    right_on=["fecha_post", "grado", "destino"],
    how="left",
).drop(columns=["fecha_post"], errors="ignore")

m = df["factor_hidr_real"].notna() & df["factor_hidr_ml1"].notna() & df["factor_hidr_final"].notna()
d = df.loc[m].copy()
eps = 1e-9
d["ratio_ml1"] = d["factor_hidr_real"] / d["factor_hidr_ml1"].clip(lower=eps)
d["ratio_ml2"] = d["factor_hidr_real"] / d["factor_hidr_final"].clip(lower=eps)
d["abs_log_ml1"] = np.abs(np.log(d["ratio_ml1"].clip(lower=eps)))
d["abs_log_ml2"] = np.abs(np.log(d["ratio_ml2"].clip(lower=eps)))

# score ponderado por tallos (impacto)
w = pd.to_numeric(d.get("tallos"), errors="coerce").fillna(1.0).clip(lower=0.0)
d["wscore"] = d["abs_log_ml2"] * w

```

```

# top improve/worst según mejora en abs_log (ponderado)
d["impr"] = (d["abs_log_ml1"] - d["abs_log_ml2"]) * w

best = d.sort_values("impr", ascending=False).head(10).assign(sample_group="TOP_IMPROVE_10")
worst = d.sort_values("impr", ascending=True).head(10).assign(sample_group="TOP_WORSE_10")
ex = pd.concat([best, worst], ignore_index=True)

ex_out = ex[[
    "fecha", fpp, "grado", "destino", "factor_hidr_ml1", "factor_hidr_final",
    "factor_hidr_real", "ratio_ml1", "ratio_ml2", "impr", "wscore", "sample_group"
]].rename(columns={fpp: "fecha_post_pred_used"})

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(g, out_global)
write_parquet(bd, out_dest)
write_parquet(bg, out_gr)
write_parquet(dist, out_dist)
write_parquet(ex_out, out_examples)

print("\n=== ML2 HIDR POSCOSECHA AUDIT ===")
print(f"KPI global parquet : {out_global}")
print(f"KPI por destino : {out_dest}")
print(f"KPI por grado : {out_gr}")
print(f"Ratio dist parquet : {out_dist}")
print(f"Examples 10x2 : {out_examples}")

print("\n--- KPI GLOBAL ---")
print(g.to_string(index=False))
print("\n--- RATIO DIST ---")
print(dist.to_string(index=False))
print("\n--- KPI DESTINO ---")
print(bd.to_string(index=False))
print("\n--- KPI GRADO (head) ---")
print(bg.head(13).to_string(index=False))
print("\n--- EXAMPLES (10x2) ---")
print(ex_out.to_string(index=False))

if __name__ == "__main__":
    main()

-----
[9/164] FILE: \src\audit\audit_mass_balance_pred_oferta_vs_gold.py
-----

from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet

PRED_OFER_DIA = Path("data/preds/pred_oferta_dia.parquet")
GOLD_TALLOS_GR = Path("data/gold/pred_tallos_grado_dia_ml1_full.parquet")

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def main() -> None:

```

```

if not PRED_OFER_DIA.exists():
    raise FileNotFoundError(PRED_OFER_DIA)
if not GOLD_TALLOS_GR.exists():
    raise FileNotFoundError(GOLD_TALLOS_GR)

ofer = read_parquet(PRED_OFER_DIA).copy()
gold = read_parquet(GOLD_TALLOS_GR).copy()

# -----
# Canon
# -----
ofer["fecha"] = _to_date(ofer["fecha"])
ofer["ciclo_id"] = ofer["ciclo_id"].astype(str)

if "bloque_base" in ofer.columns:
    ofer["bloque_base"] = _canon_int(ofer["bloque_base"])
elif "bloque_padre" in ofer.columns:
    ofer["bloque_base"] = _canon_int(ofer["bloque_padre"])
else:
    raise ValueError("pred_oferta_dia: falta bloque_base/bloque_padre")

if "variedad_canon" in ofer.columns:
    ofer["variedad_canon"] = _canon_str(ofer["variedad_canon"])
elif "variedad" in ofer.columns:
    ofer["variedad_canon"] = _canon_str(ofer["variedad"])
else:
    raise ValueError("pred_oferta_dia: falta variedad/variedad_canon")

ofer["tallos_pred"] = pd.to_numeric(ofer["tallos_pred"], errors="coerce")
ofer["tallos_proy"] = pd.to_numeric(ofer["tallos_proy"], errors="coerce")

gold["fecha"] = _to_date(gold["fecha"])
gold["ciclo_id"] = gold["ciclo_id"].astype(str)
gold["bloque_base"] = _canon_int(gold["bloque_base"])
gold["variedad_canon"] = _canon_str(gold["variedad_canon"])
gold["grado"] = _canon_int(gold["grado"])
gold["tallos_pred_baseline_grado_dia"] = pd.to_numeric(gold["tallos_pred_baseline_grado_dia"], errors="coerce")
gold["tallos_pred_m11_grado_dia"] = pd.to_numeric(gold["tallos_pred_m11_grado_dia"], errors="coerce")

# -----
# A) Duplicados en pred_oferta_dia
# -----
key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
dup_n = int(ofer.duplicated(subset=key).sum())
print("=====")
print("[A] DUPLICADOS pred_oferta_dia")
print("=====")
print(f"rows={len(ofer):,} | dup_rows_by_key={dup_n:,} | dup_rate={dup_n/max(len(ofer),1):.4f}")

if dup_n > 0:
    top = (
        ofer[ofer.duplicated(subset=key, keep=False)]
        .groupby(key, dropna=False)
        .size()
        .sort_values(ascending=False)
        .head(20)
    )
    print("\nTop duplicated keys (count):")
    print(top.to_string())

# -----
# B) Consistencia de tallos_proy dentro de ciclo_id
# -----
print("\n=====")
print("[B] CONSISTENCIA tallos_proy")
print("=====")
cyc_proy = ofer.groupby("ciclo_id", dropna=False).agg(
    n_rows=("tallos_proy", "size"),
    n_unique_proy=("tallos_proy", lambda x: x.dropna().nunique()),
    proy_min=("tallos_proy", "min"),
    proy_max=("tallos_proy", "max"),

```



```

).reset_index()

bad = cyc_proy[cyc_proy["n_unique_proy"] > 1].copy()
print(f" Ciclos total={len(cyc_proy):,} | Ciclos con tallos_proy variable={len(bad):,} ({len(bad)/max(len(cyc_proy),1):.2%})")
if len(bad):
    print("\nEjemplos (top 20 por rango proy_max-proy_min):")
    bad["range"] = bad["proy_max"] - bad["proy_min"]
    print(bad.sort_values("range", ascending=False).head(20).to_string(index=False))

# -----
# C) Mass balance baseline vs tallos_proy (en pred_oferta_dia)
# -----
print("\nn=====")
print("[C] MASS BALANCE en pred_oferta_dia (baseline vs tallos_proy)")
print("=====")
cyc = ofer.groupby("ciclo_id", dropna=False).agg(
    proy_any=("tallos_proy", "max"),          # max es más robusto si viene repetido
    base_sum=("tallos_pred", "sum"),
).reset_index()
cyc["abs_diff"] = (cyc["base_sum"] - cyc["proy_any"]).abs()
print(cyc["abs_diff"].describe(percentiles=[0.5, 0.9, 0.95, 0.99]).to_string())
print(f"max_abs_diff={float(cyc['abs_diff'].max()):.3f}")
worst = cyc.sort_values("abs_diff", ascending=False).head(20)
print("\nTop 20 ciclos peor diff (pred_oferta_dia):")
print(worst.to_string(index=False))

# -----
# D) Mass balance en GOLD (ML1 sum grado/dia)
# -----
print("\nn=====")
print("[D] MASS BALANCE en GOLD (sum ML1 grado/día vs target)")
print("=====")
gold_cyc = gold.groupby("ciclo_id", dropna=False).agg(
    mll_sum=("tallos_pred_mll_grado_dia", "sum"),
    base_sum=("tallos_pred_baseline_grado_dia", "sum"),
).reset_index()

gold_join = gold_cyc.merge(cyc[["ciclo_id", "proy_any"]], on="ciclo_id", how="left")
gold_join["abs_diff_mll_vs_proy"] = (gold_join["mll_sum"] - gold_join["proy_any"]).abs()
print(gold_join["abs_diff_mll_vs_proy"].describe(percentiles=[0.5, 0.9, 0.95, 0.99]).to_string())
print(f"max_abs_diff_mll_vs_proy={float(gold_join['abs_diff_mll_vs_proy'].max()):.3f}")

print("\nTop 20 ciclos peor diff (GOLD vs proy_any):")
print(gold_join.sort_values("abs_diff_mll_vs_proy", ascending=False).head(20).to_string(index=False))

if __name__ == "__main__":
    main()

-----
[10/164] FILE: \src\audit\audit_mismatch_tallos_grado_vs_dia.py
-----
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd
from common.io import read_parquet

GOLD = Path("data/gold/pred_tallos_grado_dia_mll_full.parquet")

def main() -> None:
    df = read_parquet(GOLD).copy()
    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()

    # Detectar columnas tallos por grado
    cand_base = [c for c in df.columns if c in ("tallos_pred_baseline_grado_dia", "tallos_baseline_grado_dia")]
]
    cand_mll = [c for c in df.columns if c in ("tallos_pred_mll_grado_dia", "tallos_mll_grado_dia")]

```

```

if not cand_base or not cand_mll:
    raise ValueError(f"No encuentro tallos por grado. Cols={list(df.columns)}")

col_base = cand_base[0]
col_mll = cand_mll[0]

# Detectar columnas tallos día (si existen)
col_base_dia = "tallos_pred_baseline_dia" if "tallos_pred_baseline_dia" in df.columns else None
col_mll_dia = "tallos_pred_mll_dia" if "tallos_pred_mll_dia" in df.columns else None

grp = ["fecha", "bloque_base", "variedad_canon"]

g = df.groupby(grp, dropna=False).agg(
    n_rows=("grado", "size"),
    n_grados=("grado", pd.Series.nunique),
    base_sum=(col_base, "sum"),
    mll_sum=(col_mll, "sum"),
    base_nan_rate=(col_base, lambda s: float(pd.isna(s).mean())),
    mll_nan_rate=(col_mll, lambda s: float(pd.isna(s).mean())),
    share_base_nan=("share_grado_baseline", lambda s: float(pd.isna(s).mean())) if "share_grado_baseline"
in df.columns else ("grado", lambda s: np.nan),
    share_mll_nan=("share_grado_mll", lambda s: float(pd.isna(s).mean())) if "share_grado_mll" in df.columns
else ("grado", lambda s: np.nan),
    share_base_sum=("share_grado_baseline", "sum") if "share_grado_baseline" in df.columns else ("grado",
lambda s: np.nan),
    share_mll_sum=("share_grado_mll", "sum") if "share_grado_mll" in df.columns else ("grado", lambda s: n
p.nan),
).reset_index()

if col_base_dia:
    base_day = df.groupby(grp, dropna=False)[col_base_dia].first().reset_index().rename(columns={col_base_
dia: "base_dia"})
    g = g.merge(base_day, on=grp, how="left")
    g["diff_base"] = (g["base_sum"] - g["base_dia"]).abs()
else:
    g["diff_base"] = np.nan

if col_mll_dia:
    mll_day = df.groupby(grp, dropna=False)[col_mll_dia].first().reset_index().rename(columns={col_mll_dia
: "mll_dia"})
    g = g.merge(mll_day, on=grp, how="left")
    g["diff_mll"] = (g["mll_sum"] - g["mll_dia"]).abs()
else:
    g["diff_mll"] = np.nan

eps = 1e-6
bad = g[(g["diff_base"] > eps) | (g["diff_mll"] > eps)].copy()

print("=====")
print("[A] RESUMEN")
print("=====")
print(f"groups total: {len(g):,}")
print(f"groups mismatch: {len(bad):,} ({len(bad)/max(len(g),1):.4%}")

if len(bad) == 0:
    print("OK: no hay mismatch.")
    return

print("=====")
print("[B] HIPÓTESIS RÁPIDA (por qué falla)")
print("=====")
print("Promedios en grupos mismatch:")
cols_show = ["n_grados", "n_rows", "base_nan_rate", "mll_nan_rate", "share_base_nan", "share_mll_nan", "share_ba
se_sum", "share_mll_sum"]
print(bad[cols_show].describe().to_string())

# Top casos: mayor diff
top = bad.sort_values(["diff_mll", "diff_base"], ascending=False).head(30)
print("=====")
print("[C] TOP 30 GRUPOS PEOR DIF")
print("=====")
show_cols = grp + ["n_grados", "n_rows", "diff_base", "diff_mll", "base_nan_rate", "mll_nan_rate", "share_base_n

```

```

an", "share_mll_nan", "share_base_sum", "share_mll_sum"]
print(top[show_cols].to_string(index=False))

# Ejemplo detallado del peor
worst = top.iloc[0][grp].to_dict()
print("=====")
print("[D] DETALLE DEL PEOR GRUPO (filas por grado)")
print("=====")
sub = df[
    (df["fecha"] == worst["fecha"]) &
    (df["bloque_base"] == worst["bloque_base"]) &
    (df["variedad_canon"] == worst["variedad_canon"])
].copy()

cols_det = ["grado", col_base, col_mll]
for c in ["share_grado_baseline", "share_grado_mll", col_base_dia, col_mll_dia]:
    if c and c in sub.columns:
        cols_det.append(c)

sub = sub[cols_det].sort_values("grado")
print(sub.to_string(index=False))

if __name__ == "__main__":
    main()

-----
[11/164] FILE: \src\audit\audit_mll_curva_clipping_impact.py
-----
# src/audit/audit_mll_curva_clipping_impact.py
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

# =====
# Paths
# =====
IN_CURVA = Path("data/gold/pred_factor_curva_mll.parquet")
IN_GRID = Path("data/gold/universe_harvest_grid_mll.parquet") # para rel_pos_pred/day_in_harvest_pred/n_harvest_days_pred

OUT_REPORT = Path("data/audit/audit_mll_curva_clipping_report.parquet")
OUT_RELPOS = Path("data/audit/audit_mll_curva_clipping_by_relpos.parquet")
OUT_CYCLES = Path("data/audit/audit_mll_curva_clipping_by_cycle.parquet")

# =====
# Helpers
# =====
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

```

```

def _coalesce(df: pd.DataFrame, out_col: str, candidates: list[str]) -> None:
    base = df[out_col] if out_col in df.columns else pd.Series([pd.NA] * len(df), index=df.index)
    for c in candidates:
        if c in df.columns:
            base = base.where(base.notna(), df[c])
    df[out_col] = base

def _safe_bool(s: pd.Series) -> pd.Series:
    if s.dtype == bool:
        return s
    x = s.copy()
    if x.dtype.kind in "if":
        return x.fillna(0).astype(float).ne(0)
    return x.astype(str).str.strip().str.lower().isin(["1", "true", "t", "yes", "y"])

def _bin_repos(rel: pd.Series, nbins: int = 20) -> tuple[pd.Series, pd.Series, pd.Series]:
    """
    Devuelve:
    - bin_label (string)   e.g. "[0.00,0.05]"
    - bin_lo (float)
    - bin_hi (float)
    """
    r = pd.to_numeric(rel, errors="coerce").clip(lower=0.0, upper=1.0)

    bins = np.linspace(0.0, 1.0, nbins + 1)
    cut = pd.cut(r, bins=bins, include_lowest=True)

    # cut es Interval; lo convertimos a string para que pyarrow lo soporte
    lbl = cut.astype(str)

    # extraer bounds numéricos (para análisis)
    lo = cut.map(lambda x: float(x.left) if pd.notna(x) else np.nan)
    hi = cut.map(lambda x: float(x.right) if pd.notna(x) else np.nan)

    return lbl, lo, hi

def _q(s: pd.Series, q: float) -> float:
    x = pd.to_numeric(s, errors="coerce")
    return float(x.quantile(q)) if x.notna().any() else float("nan")

# =====
# Main
# =====
def main() -> None:
    created_at = pd.Timestamp.utcnow()

    curva = read_parquet(IN_CURVA).copy()
    _require(
        curva,
        ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "factor_curva_m11", "factor_curva_m11_raw"],
        "pred_factor_curva_m11",
    )

    # canon keys
    curva["ciclo_id"] = curva["ciclo_id"].astype(str)
    curva["fecha"] = _to_date(curva["fecha"])
    curva["bloque_base"] = _canon_int(curva["bloque_base"])
    curva["variedad_canon"] = _canon_str(curva["variedad_canon"])

    # merge grid para rel_pos_pred y day_in_harvest_pred (si existen)
    if IN_GRID.exists():
        grid = read_parquet(IN_GRID).copy()
        _require(grid, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_m11")

        grid["ciclo_id"] = grid["ciclo_id"].astype(str)
        grid["fecha"] = _to_date(grid["fecha"])
        grid["bloque_base"] = _canon_int(grid["bloque_base"])

```

```

grid["variedad_canon"] = _canon_str(grid["variedad_canon"])

key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
take = key + [c for c in [
    "rel_pos_pred", "day_in_harvest_pred", "n_harvest_days_pred",
    "rel_pos", "day_in_harvest", "n_harvest_days"
] if c in grid.columns]
grid2 = grid[take].drop_duplicates(subset=key)

curva = curva.merge(grid2, on=key, how="left", suffixes=("", "_grid"))

_coalesce(curva, "rel_pos_eff", ["rel_pos_eff", "rel_pos_pred", "rel_pos"])
_coalesce(curva, "day_in_harvest_eff", ["day_in_harvest_eff", "day_in_harvest_pred", "day_in_harvest"])
)
_coalesce(curva, "n_harvest_days_eff", ["n_harvest_days_eff", "n_harvest_days_pred", "n_harvest_days"])
)

else:
    curva["rel_pos_eff"] = pd.NA
    curva["day_in_harvest_eff"] = pd.NA
    curva["n_harvest_days_eff"] = pd.NA

raw = pd.to_numeric(curva["factor_curva_mll_raw"], errors="coerce")
fac = pd.to_numeric(curva["factor_curva_mll"], errors="coerce")

eps = 1e-12
was_clipped = raw.notna() & fac.notna() & (raw - fac).abs().gt(eps)

was_capped_pre = _safe_bool(curva["was_capped_pre"]) if "was_capped_pre" in curva.columns else pd.Series([
False] * len(curva))
was_capped_post = _safe_bool(curva["was_capped_post"]) if "was_capped_post" in curva.columns else pd.Series([
False] * len(curva))

# hits a extremos típicos (no asumimos, solo detectamos)
min_hits = fac.notna() & fac.le(0.2000000001)
max_hits = fac.notna() & fac.ge(4.9999999999)

# rel_pos bins (STRING + bounds)
lbl, lo, hi = _bin_relpos(curva["rel_pos_eff"], nbins=20)
curva["rel_pos_bin"] = lbl
curva["rel_pos_bin_lo"] = lo
curva["rel_pos_bin_hi"] = hi

# =====
# Report global
# =====
n = len(curva)
report = {
    "created_at": created_at,
    "rows": int(n),
    "raw_notna_rate": float(raw.notna().mean()) if n else float("nan"),
    "factor_notna_rate": float(fac.notna().mean()) if n else float("nan"),
    "clipped_rate_raw_vs_factor": float(was_clipped.mean()) if n else float("nan"),
    "was_capped_pre_rate": float(was_capped_pre.mean()) if n else float("nan"),
    "was_capped_post_rate": float(was_capped_post.mean()) if n else float("nan"),
    "min_hit_rate_factor": float(min_hits.mean()) if n else float("nan"),
    "max_hit_rate_factor": float(max_hits.mean()) if n else float("nan"),
    "raw_p01": _q(raw, 0.01),
    "raw_p50": _q(raw, 0.50),
    "raw_p99": _q(raw, 0.99),
    "factor_p01": _q(fac, 0.01),
    "factor_p50": _q(fac, 0.50),
    "factor_p99": _q(fac, 0.99),
}
df_report = pd.DataFrame([report])

# =====
# Relpos aggregation (group by STRING label)
# =====
by_rel = (
    curva.assign(
        _raw=raw,
        _fac=fac,

```

```

        _clipped=was_clipped,
        _cap_pre=was_capped_pre,
        _cap_post=was_capped_post,
        _min_hit=min_hits,
        _max_hit=max_hits,
    )
.groupby(["rel_pos_bin", "rel_pos_bin_lo", "rel_pos_bin_hi"], dropna=False)
.agg(
    rows=("ciclo_id", "size"),
    clipped_rate=("_clipped", "mean"),
    cap_pre_rate=("_cap_pre", "mean"),
    cap_post_rate=("_cap_post", "mean"),
    min_hit_rate=("_min_hit", "mean"),
    max_hit_rate=("_max_hit", "mean"),
    raw_p50=("_raw", "median"),
    raw_p95=("_raw", lambda s: _q(s, 0.95)),
    factor_p50=("_fac", "median"),
    factor_p95=("_fac", lambda s: _q(s, 0.95)),
)
.reset_index()
.sort_values(["rel_pos_bin_lo"], ascending=[True])
)

# =====
# Cycle aggregation (top offenders)
# =====
by_cyc = (
    curva.assign(
        _raw=raw,
        _fac=fac,
        _clipped=was_clipped,
        _cap_pre=was_capped_pre,
        _cap_post=was_capped_post,
        _min_hit=min_hits,
        _max_hit=max_hits,
    )
    .groupby(["ciclo_id"], dropna=False)
    .agg(
        days=("fecha", "count"),
        clipped_days=("_clipped", "sum"),
        clipped_rate=("_clipped", "mean"),
        cap_pre_days=("_cap_pre", "sum"),
        cap_post_days=("_cap_post", "sum"),
        min_hit_days=("_min_hit", "sum"),
        max_hit_days=("_max_hit", "sum"),
        raw_max=("_raw", "max"),
        raw_p95=("_raw", lambda s: _q(s, 0.95)),
        factor_max=("_fac", "max"),
        factor_p95=("_fac", lambda s: _q(s, 0.95)),
    )
    .reset_index()
    .sort_values(["clipped_days", "clipped_rate", "raw_max"], ascending=[False, False, False])
)

# =====
# Optional: impact share_pred_in vs share_curva_ml1 (si existe)
# =====
if "share_pred_in" in curva.columns and "share_curva_ml1" in curva.columns:
    sp = pd.to_numeric(curva["share_pred_in"], errors="coerce")
    sm = pd.to_numeric(curva["share_curva_ml1"], errors="coerce")

    tmp = curva[["ciclo_id"]].copy()
    tmp["_abs"] = (sp - sm).abs()
    l1 = tmp.groupby("ciclo_id", dropna=False)[ "_abs"].sum().rename("l1_share_pred_vs_final").reset_index()
)

by_cyc = by_cyc.merge(l1, on="ciclo_id", how="left")
df_report["l1_share_pred_vs_final_median"] = float(l1["l1_share_pred_vs_final"].median()) if len(l1) > 0 else np.nan
df_report["l1_share_pred_vs_final_p90"] = float(l1["l1_share_pred_vs_final"].quantile(0.90)) if len(l1) > 0 else np.nan

```

```

# =====
# Persist
# =====
OUT_REPORT.parent.mkdir(parents=True, exist_ok=True)
write_parquet(df_report, OUT_REPORT)
write_parquet(by_rel, OUT_RELPOS)
write_parquet(by_cyc, OUT_CYCLES)

# =====
# Print summary
# =====
print("\n=== CLIPPING REPORT (global) ===")
print(df_report.to_string(index=False))

print("\n=== CLIPPING BY REL_POS (top 12 bins by clipped_rate) ===")
show = by_rel.sort_values(["clipped_rate", "rows"], ascending=[False, False]).head(12)
print(show.to_string(index=False))

print("\n=== TOP 20 CYCLES (most clipped days) ===")
print(by_cyc.head(20).to_string(index=False))

print(f"\nOK -> {OUT_REPORT}")
print(f"OK -> {OUT_RELPOS}")
print(f"OK -> {OUT_CYCLES}")

if __name__ == "__main__":
    main()

-----
[12/164] FILE: \src\audit\audit_mll_curva_share_vs_real.py
-----
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd

from common.io import read_parquet

# -----
# Paths
# -----
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_mll.parquet")
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")

# Baseline día (para comparar forma)
OFERTA_PATH = Path("data/preds/pred_oferta_dia.parquet")

# Predicciones nuevas por share (SALIDA del apply_curva_share_dia)
# OJO: si tu apply escribe en pred_factor_curva_mll.parquet, úsalo.
PRED_FACTOR_PATH = Path("data/gold/pred_factor_curva_mll.parquet")

# Para reconstruir tallos_mll_dia: baseline * factor (compat downstream)
OUT_REPORT = Path("data/audit/audit_mll_curva_share_report.parquet")

# -----
# Helpers
# -----
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:

```

```

return s.astype(str).str.upper().str.strip()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Cols={list(df.columns)}")

def _safe_div(a: np.ndarray, b: np.ndarray, eps: float = 1e-12) -> np.ndarray:
    return a / (b + eps)

def _cycle_metrics(df: pd.DataFrame) -> pd.Series:
    """
    df: rows de un ciclo con columnas:
    - tallos_real_dia (>=0)
    - tallos_mll_dia (>=0)
    - tallos_base_dia (>=0)
    """
    r = df["tallos_real_dia"].to_numpy(dtype=float)
    m = df["tallos_mll_dia"].to_numpy(dtype=float)

    sr = r.sum()
    sm = m.sum()

    if sr <= 0:
        return pd.Series(
            {
                "has_real": False,
                "ll_share": np.nan,
                "ks_cdf": np.nan,
                "peak_pos_err_days": np.nan,
                "mass_early_diff": np.nan,
                "mass_tail_diff": np.nan,
                "n_days": int(len(df)),
            }
        )

    # shares
    pr = r / sr
    pm = np.where(sm > 0, m / sm, 0.0)

    # L1 share distance
    ll = float(np.abs(pm - pr).sum())

    # KS on CDF
    cdf_r = np.cumsum(pr)
    cdf_m = np.cumsum(pm)
    ks = float(np.max(np.abs(cdf_m - cdf_r)))

    # Peak position error (argmax day index)
    peak_r = int(np.argmax(pr))
    peak_m = int(np.argmax(pm))
    peak_err = peak_m - peak_r

    # Early/tail mass diffs (primer 20% y último 20%)
    n = len(df)
    k = max(1, int(np.ceil(0.20 * n)))
    early_r = float(pr[:k].sum())
    early_m = float(pm[:k].sum())
    tail_r = float(pr[-k:].sum())
    tail_m = float(pm[-k:].sum())

    return pd.Series(
        {
            "has_real": True,
            "ll_share": ll,
            "ks_cdf": ks,
            "peak_pos_err_days": peak_err,
            "mass_early_diff": early_m - early_r,
            "mass_tail_diff": tail_m - tail_r,
        }
    )

```



```

        "n_days": int(n),
    }
)

def main() -> None:
    created_at = pd.Timestamp.utcnow()

    # -----
    # Read
    # -----
    uni = read_parquet(UNIVERSE_PATH).copy()
    prog = read_parquet(PROG_PATH).copy()
    oferta = read_parquet(OFERTA_PATH).copy()
    pred = read_parquet(PRED_FACTOR_PATH).copy()

    # -----
    # Canon keys
    # -----
    _require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe")
    uni["ciclo_id"] = uni["ciclo_id"].astype(str)
    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_int(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    uni_k = uni[key].drop_duplicates()

    # PROG: viene variedad raw => canonizar mínimo (XLENCE->XL, CLOUD->CLO).
    _require(prog, ["ciclo_id", "fecha", "bloque_base", "variedad", "tallos_real_dia"], "prog")
    prog["ciclo_id"] = prog["ciclo_id"].astype(str)
    prog["fecha"] = _to_date(prog["fecha"])
    prog["bloque_base"] = _canon_int(prog["bloque_base"])
    prog["variedad_raw"] = _canon_str(prog["variedad"])
    prog["variedad_canon"] = prog["variedad_raw"].replace({"XLENCE": "XL", "CLOUD": "CLO"})
    prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce").fillna(0.0).astype(float)
)

    prog_k = prog[["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_real_dia"]].drop_duplicates(subset=key)

    # OFERTA baseline
    _require(oferta, ["ciclo_id", "fecha", "bloque_base", "stage", "tallos_pred"], "oferta")
    oferta["ciclo_id"] = oferta["ciclo_id"].astype(str)
    oferta["fecha"] = _to_date(oferta["fecha"])
    oferta["bloque_base"] = _canon_int(oferta["bloque_base"])
    oferta["stage"] = _canon_str(oferta["stage"])
    # canon variedad (si ya existe en oferta, ok; si no, fallback de tu pipeline)
    if "variedad_canon" in oferta.columns:
        oferta["variedad_canon"] = _canon_str(oferta["variedad_canon"])
    elif "variedad" in oferta.columns:
        oferta["variedad_canon"] = _canon_str(oferta["variedad"]).replace({"XLENCE": "XL", "CLOUD": "CLO"})
    else:
        oferta["variedad_canon"] = "UNKNOWN"
    oferta = oferta[oferta["stage"].eq("HARVEST")].copy()
    oferta["tallos_base_dia"] = pd.to_numeric(oferta["tallos_pred"], errors="coerce").fillna(0.0).astype(float)
)

    oferta_k = (
        oferta.groupby(key, as_index=False)
        .agg(tallos_base_dia=("tallos_base_dia", "sum"),
            tallos_proy=("tallos_proy", "max") if "tallos_proy" in oferta.columns else ("tallos_base_dia", "sum"))
    )

    # PRED factor / share
    _require(pred, ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "factor_curva_ml1"], "pred_factor")
    pred["ciclo_id"] = pred["ciclo_id"].astype(str)
    pred["fecha"] = _to_date(pred["fecha"])
    pred["bloque_base"] = _canon_int(pred["bloque_base"])
    pred["variedad_canon"] = _canon_str(pred["variedad_canon"])
    pred["factor_curva_ml1"] = pd.to_numeric(pred["factor_curva_ml1"], errors="coerce").fillna(1.0).astype(float)
    at)

```

```

pred_k = pred[key + ["factor_curva_m11"]].drop_duplicates(subset=key)

# -----
# Panel
# -----
panel = (
    uni_k.merge(oferta_k, on=key, how="left")
        .merge(pred_k, on=key, how="left")
        .merge(prog_k, on=key, how="left")
)

panel["tallos_base_dia"] = pd.to_numeric(panel["tallos_base_dia"], errors="coerce").fillna(0.0).astype(float)
panel["factor_curva_m11"] = pd.to_numeric(panel["factor_curva_m11"], errors="coerce").fillna(1.0).astype(float)
grp = ["ciclo_id"]
panel["_f"] = panel["factor_curva_m11"].clip(lower=0.0)

s = panel.groupby(grp, dropna=False)["_f"].transform("sum")
panel["_w"] = np.where(s > 0, panel["_f"] / s, 0.0)

panel["tallos_m11_dia"] = panel["_w"] * panel["tallos_proy"]

panel["tallos_real_dia"] = pd.to_numeric(panel["tallos_real_dia"], errors="coerce").fillna(0.0).astype(float)

# Coverage
universe_rows = int(len(uni_k))
miss_oferta = int(panel["tallos_base_dia"].isna().sum()) # (debería ser 0 tras fillna, pero dejamos)
miss_pred = int(panel["factor_curva_m11"].isna().sum())
cov_real = float((panel["tallos_real_dia"] > 0).mean())

print("=== COVERAGE ===")
print(
    pd.DataFrame([
        "created_at": created_at,
        "universe_rows": universe_rows,
        "coverage_real_gt0": cov_real,
        "miss_oferta_rows": miss_oferta,
        "miss_pred_rows": miss_pred,
    ])).to_string(index=False)
)

# -----
# Invariants
# -----
# Mass balance M11 vs proy (si tallos_proy existe y es consistente)
if "tallos_proy" in panel.columns:
    cyc = panel.groupby("ciclo_id", dropna=False).agg(
        proy=("tallos_proy", "max"),
        m11_sum=("tallos_m11_dia", "sum"),
    )
    cyc["abs_diff"] = (cyc["proy"].astype(float) - cyc["m11_sum"].astype(float)).abs()
    print("\n=== MASS BALANCE (cycle) ===")
    print(f"cycles={len(cyc):,} | max abs diff m11 vs proy: {float(cyc['abs_diff'].max()):.12f}")

# -----
# Shape metrics (cycle-level) for cycles with real>0
# -----
# Orden por fecha dentro de ciclo
panel = panel.sort_values(["ciclo_id", "fecha"]).reset_index(drop=True)

cyc_metrics = panel.groupby("ciclo_id", dropna=False).apply(_cycle_metrics).reset_index()

n_total = int(cyc_metrics["ciclo_id"].nunique())
n_real = int(cyc_metrics["has_real"].sum())
print("\n=== SHAPE (cycle) ===")
print(f"cycles total={n_total:,} | cycles with real={n_real:,}")

if n_real > 0:
    cols_show = ["l1_share", "ks_cdf", "peak_pos_err_days", "mass_early_diff", "mass_tail_diff"]
    for c in cols_show:

```

```

        s = cyc_metrics.loc[cyc_metrics["has_real"], c].astype(float)
        print(f"\n{c}:")
        print(s.describe().to_string())

# -----
# Save detailed report
# -----
panel["created_at_audit"] = created_at
OUT_REPORT.parent.mkdir(parents=True, exist_ok=True)
write = panel # full panel for drill-down
# guardamos también métricas por ciclo en JSON sidecar
cyc_out = cyc_metrics.copy()
cyc_out["created_at_audit"] = created_at
write.to_parquet(OUT_REPORT, index=False)

cyc_path = OUT_REPORT.with_name("audit_mll_curva_share_cycle_metrics.parquet")
cyc_out.to_parquet(cyc_path, index=False)

print(f"\nOK -> {OUT_REPORT}")
print(f"OK -> {cyc_path}")

if __name__ == "__main__":
    main()

-----
[13/164] FILE: \src\audit\audit_mll_curva_shares.py
-----
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

# =====
# Paths
# =====
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_mll.parquet")
PRED_FULL_PATH = Path("data/gold/pred_tallos_grado_dia_mll_full.parquet")

CURVA_PATH = Path("data/gold/pred_factor_curva_mll.parquet")
DIST_PATH = Path("data/gold/pred_dist_grado_mll.parquet")
OFERTA_PATH = Path("data/preds/pred_oferta_dia.parquet")
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")

DIM_VAR_PATH = Path("data/silver/dim_variedad_canon.parquet")

AUDIT_DIR = Path("data/audit/mll_status")

# =====
# Helpers
# =====
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:

```

```

        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _load_var_map() -> dict[str, str]:
    if not DIM_VAR_PATH.exists():
        # fallback duro (tu convención)
        return {"XLENCE": "XL", "XL": "XL", "CLOUD": "CLO", "CLO": "CLO"}
    dv = read_parquet(DIM_VAR_PATH).copy()
    _require(dv, ["variedad_raw", "variedad_canon"], "dim_variedad_canon")
    dv["raw"] = _canon_str(dv["variedad_raw"])
    dv["canon"] = _canon_str(dv["variedad_canon"])
    m = dict(zip(dv["raw"], dv["canon"]))
    # asegurar mínimos
    m.setdefault("XLENCE", "XL")
    m.setdefault("XL", "XL")
    m.setdefault("CLOUD", "CLO")
    m.setdefault("CLO", "CLO")
    return m

def _canon_var_from_col(df: pd.DataFrame, col: str, var_map: dict[str, str]) -> pd.Series:
    raw = _canon_str(df[col]) if col in df.columns else pd.Series(["UNKNOWN"] * len(df))
    return raw.map(var_map).fillna(raw)

def _ensure_relpos(uni: pd.DataFrame) -> pd.DataFrame:
    out = uni.copy()
    if "rel_pos" not in out.columns and "rel_pos_pred" in out.columns:
        out["rel_pos"] = pd.to_numeric(out["rel_pos_pred"], errors="coerce")
    if "day_in_harvest" not in out.columns and "day_in_harvest_pred" in out.columns:
        out["day_in_harvest"] = pd.to_numeric(out["day_in_harvest_pred"], errors="coerce").astype("Int64")
    if "n_harvest_days" not in out.columns and "n_harvest_days_pred" in out.columns:
        out["n_harvest_days"] = pd.to_numeric(out["n_harvest_days_pred"], errors="coerce").astype("Int64")
    return out

def _cycle_shape_metrics(sub: pd.DataFrame) -> dict:
    # requiere: fecha, share_mll, share_real opcional, rel_pos opcional
    sub = sub.sort_values("fecha").copy()

    mll = pd.to_numeric(sub["share_mll"], errors="coerce").fillna(0.0).to_numpy(float)
    mll = np.clip(mll, 0.0, None)
    mll = mll / (mll.sum() + 1e-12)

    out = {
        "n_days": int(len(sub)),
        "has_real": 0,
        "peak_idx_mll": int(np.argmax(mll)) if len(mll) else pd.NA,
        "peak_share_mll": float(np.max(mll)) if len(mll) else pd.NA,
        "l1_share": pd.NA,
        "ks_cdf": pd.NA,
        "peak_pos_err_days": pd.NA,
        "mass_early_mll": pd.NA,
        "mass_tail_mll": pd.NA,
        "mass_early_real": pd.NA,
        "mass_tail_real": pd.NA,
        "mass_early_diff": pd.NA,
        "mass_tail_diff": pd.NA,
    }

    if "rel_pos" in sub.columns:
        rel = pd.to_numeric(sub["rel_pos"], errors="coerce").to_numpy(float)
        out["mass_early_mll"] = float(mll[rel <= 0.15].sum())
        out["mass_tail_mll"] = float(mll[rel >= 0.85].sum())

    if "share_real" not in sub.columns or sub["share_real"].notna().sum() == 0:
        return out

    real = pd.to_numeric(sub["share_real"], errors="coerce").fillna(0.0).to_numpy(float)
    real = np.clip(real, 0.0, None)
    real = real / (real.sum() + 1e-12)

```

```

out["has_real"] = 1
peak_real = int(np.argmax(real)) if len(real) else pd.NA
out["peak_pos_err_days"] = (out["peak_idx_m11"] - peak_real) if pd.notna(peak_real) else pd.NA
out["l1_share"] = float(np.abs(real - m11).sum())
out["ks_cdf"] = float(np.max(np.abs(np.cumsum(real) - np.cumsum(m11))))

if "rel_pos" in sub.columns:
    rel = pd.to_numeric(sub["rel_pos"], errors="coerce").to_numpy(float)
    m_early_real = float(real[rel <= 0.15].sum())
    m_tail_real = float(real[rel >= 0.85].sum())
    out["mass_early_real"] = m_early_real
    out["mass_tail_real"] = m_tail_real
    out["mass_early_diff"] = float(out["mass_early_m11"] - m_early_real) if pd.notna(out["mass_early_m11"])
) else pd.NA
    out["mass_tail_diff"] = float(out["mass_tail_m11"] - m_tail_real) if pd.notna(out["mass_tail_m11"]) el
se pd.NA

return out

# =====
# Main
# =====
def main() -> None:
    AUDIT_DIR.mkdir(parents=True, exist_ok=True)
    created_at = pd.Timestamp.utcnow()
    var_map = _load_var_map()

    # ----- Load
    uni = read_parquet(UNIVERSE_PATH).copy()
    curva = read_parquet(CURVA_PATH).copy()
    dist = read_parquet(DIST_PATH).copy()
    oferta = read_parquet(OFERTA_PATH).copy()
    full = read_parquet(PRED_FULL_PATH).copy()
    prog = read_parquet(PROG_PATH).copy() if PROG_PATH.exists() else pd.DataFrame()

    # ----- Canon UNIVERSE
    _require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_m11")
    uni["ciclo_id"] = uni["ciclo_id"].astype(str)
    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_int(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])
    if "stage" in uni.columns:
        uni["stage"] = _canon_str(uni["stage"])
        uni = uni[uni["stage"].eq("HARVEST")].copy()
    uni = _ensure_relpos(uni)

    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    uni_k = uni[key].drop_duplicates()

    # ----- Canon CURVA
    _require(curva, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "pred_factor_curva_m11")
    curva["ciclo_id"] = curva["ciclo_id"].astype(str)
    curva["fecha"] = _to_date(curva["fecha"])
    curva["bloque_base"] = _canon_int(curva["bloque_base"])
    curva["variedad_canon"] = _canon_str(curva["variedad_canon"])
    curva_k = curva[key].drop_duplicates()

    # ----- Canon DIST (día-level coverage; grade not needed here)
    _require(dist, ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_m11"], "pred_d
ist_grado_m11")
    dist["ciclo_id"] = dist["ciclo_id"].astype(str)
    dist["fecha"] = _to_date(dist["fecha"])
    dist["bloque_base"] = _canon_int(dist["bloque_base"])
    dist["variedad_canon"] = _canon_str(dist["variedad_canon"])
    dist["grado"] = _canon_int(dist["grado"])
    dist_day_k = dist[key].drop_duplicates()

    # ----- Canon OFERTA (baseline)
    _require(oferta, ["ciclo_id", "fecha", "bloque_base", "tallos_pred"], "pred_oferta_dia")
    oferta["ciclo_id"] = oferta["ciclo_id"].astype(str)
    oferta["fecha"] = _to_date(oferta["fecha"])

```

```

oferta["bloque_base"] = _canon_int(oferta["bloque_base"])
if "stage" in oferta.columns:
    oferta["stage"] = _canon_str(oferta["stage"])
    oferta = oferta[oferta["stage"].eq("HARVEST")].copy()
# variedad_canon
if "variedad_canon" in oferta.columns:
    oferta["variedad_canon"] = _canon_str(oferta["variedad_canon"])
elif "variedad" in oferta.columns:
    oferta["variedad_canon"] = _canon_var_from_col(oferta, "variedad", var_map)
else:
    oferta["variedad_canon"] = "UNKNOWN"

oferta["tallos_pred_baseline_dia"] = pd.to_numeric(oferta["tallos_pred"], errors="coerce").fillna(0.0)
oferta_k = oferta[key + ["tallos_pred_baseline_dia"]].drop_duplicates(subset=key)

# ----- Canon PROG (real)
has_prog = (len(prog) > 0)
if has_prog:
    _require(prog, ["ciclo_id", "fecha", "bloque_base", "variedad", "tallos_real_dia"], "dim_cosecha_progr
ess_bloque_fecha")
    prog["ciclo_id"] = prog["ciclo_id"].astype(str)
    prog["fecha"] = _to_date(prog["fecha"])
    prog["bloque_base"] = _canon_int(prog["bloque_base"])
    prog["variedad_canon"] = _canon_var_from_col(prog, "variedad", var_map)
    prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce")
    prog_k = prog[key + ["tallos_real_dia", "pct_avance_real"]].drop_duplicates(subset=key)
else:
    prog_k = pd.DataFrame(columns=key + ["tallos_real_dia", "pct_avance_real"])

# =====
# 1) COVERAGE
# =====
miss_curva = uni_k.merge(curva_k, on=key, how="left", indicator=True)
miss_curva = miss_curva[miss_curva["_merge"].eq("left_only")].drop(columns=["_merge"])

miss_dist = uni_k.merge(dist_day_k, on=key, how="left", indicator=True)
miss_dist = miss_dist[miss_dist["_merge"].eq("left_only")].drop(columns=["_merge"])

miss_oferta = uni_k.merge(oferta_k[key].drop_duplicates(), on=key, how="left", indicator=True)
miss_oferta = miss_oferta[miss_oferta["_merge"].eq("left_only")].drop(columns=["_merge"])

miss_prog = uni_k.merge(prog_k[key].drop_duplicates(), on=key, how="left", indicator=True)
miss_prog = miss_prog[miss_prog["_merge"].eq("left_only")].drop(columns=["_merge"])

cov = pd.DataFrame([
    "created_at": created_at,
    "universe_rows": int(len(uni_k)),
    "curva_rows": int(len(curva_k)),
    "dist_day_rows": int(len(dist_day_k)),
    "oferta_rows": int(len(oferta_k)),
    "prog_rows": int(len(prog_k)),
    "miss_curva_rows": int(len(miss_curva)),
    "miss_dist_rows": int(len(miss_dist)),
    "miss_oferta_rows": int(len(miss_oferta)),
    "miss_prog_rows": int(len(miss_prog)),
    "miss_curva_rate": float(len(miss_curva) / max(len(uni_k), 1)),
    "miss_dist_rate": float(len(miss_dist) / max(len(uni_k), 1)),
    "miss_oferta_rate": float(len(miss_oferta) / max(len(uni_k), 1)),
    "miss_prog_rate": float(len(miss_prog) / max(len(uni_k), 1)),
])
write_parquet(cov, AUDIT_DIR / "coverage.parquet")

# =====
# 2) INVARIANTS on FINAL
# =====
_require(full, ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"], "pred_tallos_grado_dia_mll
_full")
for c in ["tallos_pred_baseline_dia", "tallos_pred_mll_dia",
    "tallos_pred_baseline_grado_dia", "tallos_pred_mll_grado_dia"]:
    if c in full.columns:
        full[c] = pd.to_numeric(full[c], errors="coerce")

```

```

full["ciclo_id"] = full["ciclo_id"].astype(str)
full["fecha"] = _to_date(full["fecha"])
full["bloque_base"] = _canon_int(full["bloque_base"])
full["variedad_canon"] = _canon_str(full["variedad_canon"])
full["grado"] = _canon_int(full["grado"])

g_day = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
sum_base = full.groupby(g_day, dropna=False)["tallos_pred_baseline_grado_dia"].sum().rename("sum_grado_base")
sum_mll = full.groupby(g_day, dropna=False)["tallos_pred_mll_grado_dia"].sum().rename("sum_grado_mll")

day_first = full.drop_duplicates(subset=g_day)[g_day + ["tallos_pred_baseline_dia", "tallos_pred_mll_dia"]
].set_index(g_day)
day_chk = day_first.join(sum_base).join(sum_mll).reset_index()

eps = 1e-6
day_chk["mismatch_base"] = (day_chk["sum_grado_base"] - day_chk["tallos_pred_baseline_dia"]).abs() > eps
day_chk["mismatch_mll"] = (day_chk["sum_grado_mll"] - day_chk["tallos_pred_mll_dia"]).abs() > eps

inv = pd.DataFrame([
    "created_at": created_at,
    "pct_day_mismatch_base": float(day_chk["mismatch_base"].mean()) if len(day_chk) else np.nan,
    "pct_day_mismatch_mll": float(day_chk["mismatch_mll"].mean()) if len(day_chk) else np.nan,
    "pct_day_mll_dia_nan": float(day_chk["tallos_pred_mll_dia"].isna().mean()) if len(day_chk) else np.nan,
    "pct_day_mll_dia_zero": float((day_chk["tallos_pred_mll_dia"].fillna(0.0) == 0.0).mean()) if len(day_c
hk) else np.nan,
])
write_parquet(inv, AUDIT_DIR / "invariants_day.parquet")

# Mass-balance per cycle (needs tallos_proy; take from oferta if present)
if "tallos_proy" in oferta.columns:
    oferta_tp = oferta[["ciclo_id", "tallos_proy"]].copy()
    oferta_tp["ciclo_id"] = oferta_tp["ciclo_id"].astype(str)
    oferta_tp["tallos_proy"] = pd.to_numeric(oferta_tp["tallos_proy"], errors="coerce")
    tproy = oferta_tp.groupby("ciclo_id", dropna=False)["tallos_proy"].max()
else:
    tproy = pd.Series(dtype=float)

cyc = day_chk.groupby("ciclo_id", dropna=False).agg(
    mll_sum=("tallos_pred_mll_dia", "sum"),
    base_sum=("tallos_pred_baseline_dia", "sum"),
    days=("ciclo_id", "size"),
)
if len(tproy):
    cyc = cyc.join(tproy.rename("tallos_proy"), how="left")
    cyc["abs_diff_mll_vs_proy"] = (cyc["mll_sum"] - cyc["tallos_proy"]).abs()
    cyc["rel_diff_mll_vs_proy"] = np.where(
        cyc["tallos_proy"].fillna(0).astype(float) > 0,
        cyc["abs_diff_mll_vs_proy"] / cyc["tallos_proy"].astype(float),
        np.nan,
    )
else:
    cyc["tallos_proy"] = np.nan
    cyc["abs_diff_mll_vs_proy"] = np.nan
    cyc["rel_diff_mll_vs_proy"] = np.nan

cyc = cyc.reset_index()
write_parquet(cyc, AUDIT_DIR / "mass_balance_cycle.parquet")

# =====
# 3) SHAPE vs REAL (curve)
# =====
# We approximate MLL share from final output:
# share_mll_day = tallos_pred_mll_dia / sum_cycle(tallos_pred_mll_dia)
# share_real_day = tallos_real_dia / sum_cycle(tallos_real_dia)
day_panel = day_chk.merge(
    uni.drop_duplicates(subset=g_day)[g_day + [c for c in ["rel_pos", "day_in_harvest", "n_harvest_days",
"month"] if c in uni.columns]],
    on=g_day,
    how="left",
).merge(

```

```

        prog_k[g_day + ["tallos_real_dia", "pct_avance_real"]],
        on=g_day,
        how="left",
    )

denom_mll = day_panel.groupby("ciclo_id", dropna=False)["tallos_pred_mll_dia"].transform("sum")
day_panel["share_mll"] = np.where(denom_mll > 0, day_panel["tallos_pred_mll_dia"] / denom_mll, 0.0)

denom_real = day_panel.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform(
    lambda s: np.nansum(pd.to_numeric(s, errors="coerce").to_numpy(dtype=float))
)
day_panel["share_real"] = np.where(denom_real > 0, day_panel["tallos_real_dia"] / denom_real, np.nan)

write_parquet(day_panel, AUDIT_DIR / "day_panel.parquet")

cyc_metrics = []
for cid, sub in day_panel.groupby("ciclo_id", dropna=False):
    m = _cycle_shape_metrics(sub)
    m["ciclo_id"] = cid
    # tags for segmentation
    v = sub["variedad_canon"].dropna()
    m["variedad_canon"] = v.iloc[0] if len(v) else pd.NA
    b = sub["bloque_base"].dropna()
    m["bloque_base"] = b.iloc[0] if len(b) else pd.NA
    mo = sub["month"].dropna() if "month" in sub.columns else pd.Series([], dtype="Int64")
    m["month"] = int(mo.iloc[0]) if len(mo) else pd.NA
    cyc_metrics.append(m)

cyc_shape = pd.DataFrame(cyc_metrics)
write_parquet(cyc_shape, AUDIT_DIR / "shape_cycle.parquet")

# segmentation summary (only where has_real)
with_real = cyc_shape[cyc_shape["has_real"].eq(1)].copy()
seg_rows = []
if len(with_real):
    for by, tag in [
        (["variedad_canon"], "by_variedad"),
        (["bloque_base"], "by_bloque"),
        (["variedad_canon", "month"], "by_variedad_month"),
    ]:
        g = with_real.groupby(by, dropna=False).agg(
            n_cycles=("ciclo_id", "count"),
            ll_median=("ll_share", "median"),
            ks_median=("ks_cdf", "median"),
            peak_err_median=("peak_pos_err_days", "median"),
            early_diff_median=("mass_early_diff", "median"),
            tail_diff_median=("mass_tail_diff", "median"),
        ).reset_index()
        g["segment_tag"] = tag
        seg_rows.append(g.sort_values("n_cycles", ascending=False))
seg = pd.concat(seg_rows, ignore_index=True) if seg_rows else pd.DataFrame()
write_parquet(seg, AUDIT_DIR / "shape_segmentation.parquet")

# =====
# Prints (state snapshot)
# =====
print("\n=== COVERAGE ===")
print(cov.to_string(index=False))

print("\n=== INVARIANTS (day) ===")
print(inv.to_string(index=False))

if len(cyc):
    mx = float(pd.to_numeric(cyc["abs_diff_mll_vs_proy"], errors="coerce").max())
    print("\n=== MASS BALANCE (cycle) ===")
    print(f"cycles={len(cyc):,} | max abs diff mll vs proy: {mx:.12f}")

print("\n=== SHAPE (cycle) ===")
print(f"cycles total={len(cyc_shape):,} | cycles with real={int(cyc_shape['has_real'].sum()):,}")
if len(with_real):
    print("\nll share (lower is better):")
    print(with_real["ll_share"].describe().to_string())

```



```

print("\nKS CDF (lower is better):")
print(with_real["ks_cdf"].describe().to_string())
print("\nPeak position error (days):")
print(pd.to_numeric(with_real["peak_pos_err_days"], errors="coerce").describe().to_string())
print("\nMass early diff (ML1 - real):")
print(pd.to_numeric(with_real["mass_early_diff"], errors="coerce").describe().to_string())
print("\nMass tail diff (ML1 - real):")
print(pd.to_numeric(with_real["mass_tail_diff"], errors="coerce").describe().to_string())
else:
    print("[WARN] No hay ciclos con real en el join (revisa PROG).")

print(f"\nOK -> auditoría escrita en: {AUDIT_DIR}")

if __name__ == "__main__":
    main()

-----
[14/164] FILE: \src\audit\audit_peso_tallo_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"

IN_FACT = EVAL / "backtest_factor_ml2_peso_tallo_grado_dia.parquet"
IN_TALLOS = SILVER / "fact_cosecha_real_grado_dia.parquet"

TS = datetime.now().strftime("%Y%m%d_%H%M%S")

OUT_KPI_GLOBAL = EVAL / f"audit_peso_tallo_ml2_kpi_global_{TS}.parquet"
OUT_DIST = EVAL / f"audit_peso_tallo_ml2_ratio_dist_{TS}.parquet"
OUT_BY_GRADO = EVAL / f"audit_peso_tallo_ml2_kpi_by_grado_{TS}.parquet"
OUT_EXAMPLES = EVAL / f"audit_peso_tallo_ml2_examples_10x2_{TS}.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else np.nan

def wape_weighted(err_abs: pd.Series, w: pd.Series) -> float:
    err_abs = pd.to_numeric(err_abs, errors="coerce").fillna(0.0)
    w = pd.to_numeric(w, errors="coerce").fillna(0.0)
    denom = float(np.sum(w))
    if denom <= 0:
        return np.nan
    return float(np.sum(err_abs * w) / denom)

```

```

def main() -> None:
    df = read_parquet(IN_FACT).copy()

    # Tallos reales para ponderar (impacto en kg y wape ponderado)
    tl = read_parquet(IN_TALLOS).copy()
    tl["fecha"] = _to_date(tl["fecha"])
    tl["grado"] = _canon_str(tl["grado"]) if "grado" in tl.columns else tl["grado"]

    if "bloque_base" in tl.columns:
        tl["bloque_base"] = _canon_str(tl["bloque_base"])
    elif "bloque_padre" in tl.columns:
        tl["bloque_base"] = _canon_str(tl["bloque_padre"])
    elif "bloque" in tl.columns:
        tl["bloque_base"] = _canon_str(tl["bloque"])
    else:
        raise KeyError("fact_cosecha_real_grado_dia no tiene bloque_base/bloque_padre/bloque")

    tl["tallos_real"] = pd.to_numeric(tl["tallos_real"], errors="coerce").fillna(0.0)
    tl = tl.groupby(["fecha", "bloque_base", "grado"], as_index=False).agg(tallos_real=("tallos_real", "sum"))

    # Canon fact
    df["fecha"] = _to_date(df["fecha"])
    df["bloque_base"] = _canon_str(df["bloque_base"])
    df["grado"] = _canon_str(df["grado"])

    for c in ["peso_tallo_ml1_g", "peso_tallo_real_g", "peso_tallo_final_g", "pred_ratio_peso"]:
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce")

    # Merge tallos weights
    df = df.merge(tl, on=["fecha", "bloque_base", "grado"], how="left")
    df["tallos_real"] = pd.to_numeric(df["tallos_real"], errors="coerce").fillna(0.0)

    # Keep only rows where we have real
    df = df[df["peso_tallo_real_g"].notna()].copy()

    # Errors (g/tallo)
    df["err_ml1_g"] = df["peso_tallo_real_g"] - df["peso_tallo_ml1_g"]
    df["err_ml2_g"] = df["peso_tallo_real_g"] - df["peso_tallo_final_g"]

    df["abs_err_ml1_g"] = df["err_ml1_g"].abs()
    df["abs_err_ml2_g"] = df["err_ml2_g"].abs()

    # Equivalent absolute kg error = abs_err_g * tallos / 1000
    df["abs_err_ml1_kg_equiv"] = (df["abs_err_ml1_g"] * df["tallos_real"]) / 1000.0
    df["abs_err_ml2_kg_equiv"] = (df["abs_err_ml2_g"] * df["tallos_real"]) / 1000.0

    # === KPI GLOBAL ===
    kpi_g = pd.DataFrame([
        "n_rows": int(len(df)),
        "n_cycles": int(df["ciclo_id"].nunique()) if "ciclo_id" in df.columns else np.nan,
        "mae_ml1_g": mae(df["err_ml1_g"]),
        "mae_ml2_g": mae(df["err_ml2_g"]),
        "wape_wt_ml1_g": wape_weighted(df["abs_err_ml1_g"], df["tallos_real"]),
        "wape_wt_ml2_g": wape_weighted(df["abs_err_ml2_g"], df["tallos_real"]),
        "kg_abs_err_ml1": float(df["abs_err_ml1_kg_equiv"].sum()),
        "kg_abs_err_ml2": float(df["abs_err_ml2_kg_equiv"].sum()),
        "improvement_abs_mae_g": mae(df["err_ml1_g"]) - mae(df["err_ml2_g"]),
        "improvement_abs_wape_wt_g": wape_weighted(df["abs_err_ml1_g"], df["tallos_real"]) - wape_weighted(df["abs_err_ml2_g"], df["tallos_real"]),
        "improvement_abs_kg": float(df["abs_err_ml1_kg_equiv"].sum() - df["abs_err_ml2_kg_equiv"].sum()),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    ])
    write_parquet(kpi_g, OUT_KPI_GLOBAL)

    # === DISTRIBUCIÓN DE RATIOS ===
    r = pd.to_numeric(df["pred_ratio_peso"], errors="coerce")
    dist = pd.DataFrame([
        "n": int(r.notna().sum()),
        "ratio_min": float(np.nanmin(r)),
        "ratio_p05": float(np.nanpercentile(r, 5)),
    ])

```

```

        "ratio_p25": float(np.nanpercentile(r, 25)),
        "ratio_median": float(np.nanmedian(r)),
        "ratio_p75": float(np.nanpercentile(r, 75)),
        "ratio_p95": float(np.nanpercentile(r, 95)),
        "ratio_max": float(np.nanmax(r)),
    })
write_parquet(dist, OUT_DIST)

# === KPI POR GRADO (para ver sesgos por calidad) ===
rows = []
for grado, g in df.groupby("grado"):
    rows.append({
        "grado": str(grado),
        "n_rows": int(len(g)),
        "mae_ml1_g": mae(g["err_ml1_g"]),
        "mae_ml2_g": mae(g["err_ml2_g"]),
        "wape_wt_ml1_g": wape_weighted(g["abs_err_ml1_g"], g["tallos_real"]),
        "wape_wt_ml2_g": wape_weighted(g["abs_err_ml2_g"], g["tallos_real"]),
        "kg_abs_err_ml1": float(g["abs_err_ml1_kg_equiv"].sum()),
        "kg_abs_err_ml2": float(g["abs_err_ml2_kg_equiv"].sum()),
        "improvement_abs_kg": float(g["abs_err_ml1_kg_equiv"].sum() - g["abs_err_ml2_kg_equiv"].sum()),
    })
by_grado = pd.DataFrame(rows).sort_values("improvement_abs_kg", ascending=False)
write_parquet(by_grado, OUT_BY_GRADO)

# === EJEMPLOS 10x2 (mejora vs empeora) por ciclo usando impacto kg ===
if "ciclo_id" in df.columns:
    by_cycle = (
        df.groupby("ciclo_id", as_index=False)
        .agg(
            kg_abs_err_ml1=("abs_err_ml1_kg_equiv", "sum"),
            kg_abs_err_ml2=("abs_err_ml2_kg_equiv", "sum"),
            n_rows=("fecha", "count"),
        )
    )
    by_cycle["improvement_kg"] = by_cycle["kg_abs_err_ml1"] - by_cycle["kg_abs_err_ml2"]

    top_good = by_cycle.sort_values("improvement_kg", ascending=False).head(10).assign(sample_group="TOP_IMPROVE_10")
    top_bad = by_cycle.sort_values("improvement_kg", ascending=True).head(10).assign(sample_group="TOP_WORSE_10")
    ex = pd.concat([top_good, top_bad], ignore_index=True)

    # attach some context (bloque_base, variedad_canon) if available
    ctx_cols = [c for c in ["ciclo_id", "bloque_base", "variedad_canon", "area", "tipo_sp", "estado"] if c
in df.columns]
    if ctx_cols:
        ctx = df[ctx_cols].drop_duplicates("ciclo_id")
        ex = ex.merge(ctx, on="ciclo_id", how="left")

    write_parquet(ex, OUT_EXAMPLES)
else:
    ex = pd.DataFrame()
    write_parquet(ex, OUT_EXAMPLES)

# === PRINT ===
print("\n=== ML2 PESO POR TALLO AUDIT ===")
print(f"KPI global parquet : {OUT_KPI_GLOBAL}")
print(f"Ratio dist parquet : {OUT_DIST}")
print(f"KPI por grado : {OUT_BY_GRADO}")
print(f"Examples 10x2 : {OUT_EXAMPLES}")
print("\n--- KPI GLOBAL ---")
print(kpi_g.to_string(index=False))
print("\n--- RATIO DIST ---")
print(dist.to_string(index=False))
print("\n--- KPI POR GRADO (top 10 por mejora kg) ---")
print(by_grado.head(10).to_string(index=False))
if not ex.empty:
    print("\n--- EXAMPLES 10x2 ---")
    print(ex.to_string(index=False))

```

```

if __name__ == "__main__":
    main()

-----
[15/164] FILE: \src\audit\audit_poscosecha_ml2_baseline.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
GOLD = DATA / "gold"
SILVER = DATA / "silver"

IN_GLOBAL = EVAL / "ml2_poscosecha_baseline_eval_global.parquet"
IN_BY_DEST = EVAL / "ml2_poscosecha_baseline_eval_by_destino.parquet"
IN_BY_GR = EVAL / "ml2_poscosecha_baseline_eval_by_grado.parquet"
IN_COV = EVAL / "ml2_poscosecha_baseline_eval_coverage.parquet"

# Para ejemplos (top improve/worse) calculamos con el join directo
IN_PRED_FULL = GOLD / "pred_poscosecha_ml2_full_grado_dia_bloque_destino.parquet"
IN_REAL_HIDR_DH = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"
IN_REAL_MERMA_AJ = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

OUT_KPI_GLOBAL = EVAL / "audit_poscosecha_baseline_kpi_global_{ts}.parquet"
OUT_KPI_DEST = EVAL / "audit_poscosecha_baseline_kpi_by_destino_{ts}.parquet"
OUT_KPI_GR = EVAL / "audit_poscosecha_baseline_kpi_by_grado_{ts}.parquet"
OUT_COV = EVAL / "audit_poscosecha_baseline_coverage_{ts}.parquet"
OUT_EXAMPLES = EVAL / "audit_poscosecha_baseline_examples_10x2_{ts}.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _as_of_date_default() -> pd.Timestamp:
    return pd.Timestamp.now().normalize() - pd.Timedelta(days=1)

def _safe_div(a: pd.Series, b: pd.Series) -> pd.Series:
    a = pd.to_numeric(a, errors="coerce")
    b = pd.to_numeric(b, errors="coerce")
    return a / b.replace(0, np.nan)

def main() -> None:
    ts = datetime.now().strftime("%Y%m%d_%H%M%S")
    aod = _as_of_date_default()

    g = read_parquet(IN_GLOBAL).copy()
    bd = read_parquet(IN_BY_DEST).copy()

```

```

bg = read_parquet(IN_BY_GR).copy()
cov = read_parquet(IN_COV).copy()

# Persist snapshot
write_parquet(g, Path(str(OUT_KPI_GLOBAL).format(ts=ts)))
write_parquet(bd, Path(str(OUT_KPI_DEST).format(ts=ts)))
write_parquet(bg, Path(str(OUT_KPI_GR).format(ts=ts)))
write_parquet(cov, Path(str(OUT_COV).format(ts=ts)))

# -----
# Examples (top improve/worse) sobre error de cajas_postcosecha_m11 vs real "factor-based"
# Nota: como no tenemos cajas_post real aquí, usamos un proxy robusto:
# score = |log_ratio_hidr| + |log_ratio_desp| + |log_ratio_ajuste| + (abs_err_dh_days/7)
# y mostramos TOP 10 mejores (menor score) y TOP 10 peores (mayor score)
# -----
pred = read_parquet(IN_PRED_FULL).copy()
pred.columns = [str(c).strip() for c in pred.columns]
pred["fecha"] = _to_date(pred["fecha"])
pred["grado"] = _canon_int(pred["grado"])
pred["destino"] = _canon_str(pred["destino"])

# columnas pred
dh_col = None
for c in ["dh_dias_m11", "dh_dias_pred_m11", "dh_dias"]:
    if c in pred.columns:
        dh_col = c
        break
fecha_post_pred_col = None
for c in ["fecha_post_pred_m11", "fecha_post_pred", "fecha_post_m11"]:
    if c in pred.columns:
        fecha_post_pred_col = c
        break
if fecha_post_pred_col:
    pred[fecha_post_pred_col] = _to_date(pred[fecha_post_pred_col])

hidr_col = None
for c in ["factor_hidr_m11", "factor_hidr"]:
    if c in pred.columns:
        hidr_col = c
        break
desp_col = None
for c in ["factor_desp_m11", "factor_desp"]:
    if c in pred.columns:
        desp_col = c
        break
aj_col = None
for c in ["ajuste_m11", "factor_ajuste_m11", "factor_ajuste"]:
    if c in pred.columns:
        aj_col = c
        break

pred = pred[(pred["fecha"] <= aod)].copy()
if fecha_post_pred_col:
    pred = pred[(pred[fecha_post_pred_col] <= aod)].copy()

real_hd = read_parquet(IN_REAL_HIDR_DH).copy()
real_hd.columns = [str(c).strip() for c in real_hd.columns]
real_hd["fecha_cosecha"] = _to_date(real_hd["fecha_cosecha"])
real_hd["fecha_post"] = _to_date(real_hd["fecha_post"])
real_hd["grado"] = _canon_int(real_hd["grado"])
real_hd["destino"] = _canon_str(real_hd["destino"])
real_hd["tallos"] = pd.to_numeric(real_hd["tallos"], errors="coerce").fillna(0.0)
real_hd["dh_dias"] = pd.to_numeric(real_hd["dh_dias"], errors="coerce")
real_hd["peso_base_g"] = pd.to_numeric(real_hd["peso_base_g"], errors="coerce")
real_hd["peso_post_g"] = pd.to_numeric(real_hd["peso_post_g"], errors="coerce")
real_hd["factor_hidr_real"] = _safe_div(real_hd["peso_post_g"], real_hd["peso_base_g"])
real_hd = real_hd[(real_hd["fecha_cosecha"] <= aod) & (real_hd["fecha_post"] <= aod)].copy()

rg = (
    real_hd.groupby(["fecha_cosecha", "grado", "destino"], as_index=False)
    .agg(
        tallos=("tallos", "sum"),

```

```

        dh_real=("dh_dias", "median"),
        factor_hidr_real=("factor_hidr_real", "median"),
        fecha_post_real=("fecha_post", "median"),
    )
)

real_ma = read_parquet(IN_REAL_MERMA_AJ).copy()
real_ma.columns = [str(c).strip() for c in real_ma.columns]
real_ma["fecha_post"] = _to_date(real_ma["fecha_post"])
real_ma["destino"] = _canon_str(real_ma["destino"])
real_ma["factor_desp"] = pd.to_numeric(real_ma["factor_desp"], errors="coerce")
real_ma["factor_ajuste"] = pd.to_numeric(real_ma["factor_ajuste"], errors="coerce")
real_ma = real_ma[real_ma["fecha_post"] <= aod].copy()

df = pred.merge(
    rg,
    left_on=["fecha", "grado", "destino"],
    right_on=["fecha_cosecha", "grado", "destino"],
    how="left",
)

df["fecha_post_key"] = df["fecha_post_real"]
if fecha_post_pred_col:
    df["fecha_post_key"] = df["fecha_post_key"].where(df["fecha_post_key"].notna(), df[fecha_post_pred_col])
])

df = df.merge(
    real_ma.rename(columns={"fecha_post": "fecha_post_key"}),
    on=["fecha_post_key", "destino"],
    how="left",
)

# métricas por fila
df["dh_pred"] = pd.to_numeric(df[dh_col], errors="coerce") if dh_col else np.nan
df["abs_err_dh"] = (pd.to_numeric(df["dh_pred"], errors="coerce") - pd.to_numeric(df["dh_real"], errors="coerce")).abs()

df["hidr_pred"] = pd.to_numeric(df[hidr_col], errors="coerce") if hidr_col else np.nan
df["log_ratio_hidr"] = np.log(_safe_div(df["factor_hidr_real"], df["hidr_pred"])).clip(lower=1e-6)

df["desp_pred"] = pd.to_numeric(df[desp_col], errors="coerce") if desp_col else np.nan
df["log_ratio_desp"] = np.log(_safe_div(df["factor_desp"], df["desp_pred"])).clip(lower=1e-6)

df["aj_pred"] = pd.to_numeric(df[aj_col], errors="coerce") if aj_col else np.nan
df["log_ratio_aj"] = np.log(_safe_div(df["factor_ajuste"], df["aj_pred"])).clip(lower=1e-6)

# score proxy (menor = mejor)
df["score"] = (
    df["abs_err_dh"].fillna(0.0) / 7.0
    + df["log_ratio_hidr"].abs().fillna(0.0)
    + df["log_ratio_desp"].abs().fillna(0.0)
    + df["log_ratio_aj"].abs().fillna(0.0)
)

# ponderar por tallos para ranking
df["wscore"] = df["score"] * pd.to_numeric(df["tallos"], errors="coerce").fillna(0.0).clip(lower=0.0)

# agregamos por (fecha, grado, destino) para evitar que bloque domine por cantidad de filas
key = ["fecha", "grado", "destino"]
ex = (
    df.groupby(key, dropna=False, as_index=False)
    .agg(
        tallos=("tallos", "sum"),
        score=("score", "mean"),
        wscore=("wscore", "sum"),
        abs_err_dh=("abs_err_dh", "mean"),
        log_ratio_hidr=("log_ratio_hidr", "mean"),
        log_ratio_desp=("log_ratio_desp", "mean"),
        log_ratio_aj=("log_ratio_aj", "mean"),
        fecha_post_real=("fecha_post_real", "first"),
        fecha_post_key=("fecha_post_key", "first"),
    )
)

```

```

top_best = ex.sort_values(["wscore", "tallos"], ascending=[True, False]).head(10).copy()
top_best["sample_group"] = "TOP_BEST_10"
top_worst = ex.sort_values(["wscore", "tallos"], ascending=[False, False]).head(10).copy()
top_worst["sample_group"] = "TOP_WORST_10"

examples = pd.concat([top_best, top_worst], ignore_index=True)
out_ex_path = Path(str(OUT_EXAMPLES).format(ts=ts))
write_parquet(examples, out_ex_path)

# -----
# Print
# -----
print("\n=== POSCOSECHA ML2 BASELINE AUDIT ===")
print(f"KPI global parquet : {Path(str(OUT_KPI_GLOBAL).format(ts=ts))}")
print(f"KPI por destino : {Path(str(OUT_KPI_DEST).format(ts=ts))}")
print(f"KPI por grado : {Path(str(OUT_KPI_GR).format(ts=ts))}")
print(f"Coverage parquet : {Path(str(OUT_COV).format(ts=ts))}")
print(f"Examples 10x2 : {out_ex_path}")

print("\n--- KPI GLOBAL ---")
print(g.to_string(index=False))

print("\n--- COVERAGE ---")
print(cov.to_string(index=False))

print("\n--- KPI DESTINO (head) ---")
print(bd.head(10).to_string(index=False))

print("\n--- KPI GRADO (head) ---")
print(bg.head(10).to_string(index=False))

print("\n--- EXAMPLES (10x2) ---")
print(examples.to_string(index=False))

if __name__ == "__main__":
    main()

-----
[16/164] FILE: \src\audit\audit_poscosecha_seed.py
-----

from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

IN_GD_BD = Path("data/gold/pred_poscosecha_seed_grado_dia_bloque_destino.parquet")
IN_DD = Path("data/gold/pred_poscosecha_seed_dia_destino.parquet")
IN_DT = Path("data/gold/pred_poscosecha_seed_dia_total.parquet")

OUT_CHECKS = Path("data/audit/audit_poscosecha_seed_checks.parquet")
OUT_SUMMARY = Path("data/audit/audit_poscosecha_seed_summary.json")

def _json_safe(obj):
    if isinstance(obj, dict):
        return {k: _json_safe(v) for k, v in obj.items()}
    if isinstance(obj, list):
        return [_json_safe(v) for v in obj]
    if isinstance(obj, (pd.Timestamp,)):
        return obj.isoformat()
    if isinstance(obj, (np.integer,)):
        return int(obj)
    if isinstance(obj, (np.floating,)):
        return float(obj)
    return obj

```

```

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def _pct(x: float) -> float:
    return float(np.round(x * 100.0, 4))

def main() -> None:
    gd = read_parquet(IN_GD_BD).copy()
    dd = read_parquet(IN_DD).copy()
    dt = read_parquet(IN_DT).copy()

    gd.columns = [str(c).strip() for c in gd.columns]
    dd.columns = [str(c).strip() for c in dd.columns]
    dt.columns = [str(c).strip() for c in dt.columns]

    need_gd = [
        "fecha", "fecha_post_pred", "bloque_base", "variedad_canon", "grado", "destino",
        "cajas_mll_grado_dia", "cajas_split_grado_dia", "cajas_post_seed",
        "factor_hidr_seed", "factor_desp_seed", "factor_ajuste_seed", "dh_dias"
    ]
    miss = [c for c in need_gd if c not in gd.columns]
    if miss:
        raise ValueError(f"gold seed GD_BD sin columnas: {miss}")

    # numeric
    for c in ["cajas_mll_grado_dia", "cajas_split_grado_dia", "cajas_post_seed",
              "factor_hidr_seed", "factor_desp_seed", "factor_ajuste_seed"]:
        gd[c] = _to_num(gd[c])

    # 1) Checks de integridad y rangos
    gd["chk_nonneg_split"] = gd["cajas_split_grado_dia"].fillna(0) >= -1e-9
    gd["chk_nonneg_post"] = gd["cajas_post_seed"].fillna(0) >= -1e-9

    gd["chk_hidr_range"] = gd["factor_hidr_seed"].between(0.60, 3.00, inclusive="both")
    gd["chk_desp_range"] = gd["factor_desp_seed"].between(0.05, 1.00, inclusive="both")
    gd["chk_ajus_range"] = gd["factor_ajuste_seed"].between(0.50, 2.00, inclusive="both")

    gd["chk_dh_range"] = pd.to_numeric(gd["dh_dias"], errors="coerce").between(0, 30, inclusive="both")

    # 2) Mass-balance: sum por key_supply debe igualar cajas_mll (antes del split)
    key_supply = ["fecha", "bloque_base", "variedad_canon", "grado"]
    mb = (
        gd.groupby(key_supply, dropna=False, as_index=False)
        .agg(
            cajas_mll=("cajas_mll_grado_dia", "max"),
            sum_split=("cajas_split_grado_dia", "sum"),
        )
    )
    mb["abs_diff"] = (mb["sum_split"] - mb["cajas_mll"]).abs()
    mb_ok_rate = float((mb["abs_diff"] <= 1e-9).mean()) if len(mb) else float("nan")
    mb_max_abs = float(mb["abs_diff"].max()) if len(mb) else float("nan")

    # 3) Relación post vs split (factor efectivo)
    gd["factor_efectivo_post_vs_split"] = np.where(
        gd["cajas_split_grado_dia"].abs() > 1e-12,
        gd["cajas_post_seed"] / gd["cajas_split_grado_dia"],
        np.nan
    )

    # 4) Agregados por destino
    by_dest = (
        gd.groupby("destino", dropna=False)
        .agg(
            n=("destino", "size"),
            cajas_split=("cajas_split_grado_dia", "sum"),
            cajas_post=("cajas_post_seed", "sum"),
            hidr_med=("factor_hidr_seed", "median"),
            desp_med=("factor_desp_seed", "median"),
        )
    )

```



```

        ajus_med=("factor_ajuste_seed","median"),
        dh_med=("dh_dias","median"),
        eff_med=("factor_efectivo_post_vs_split","median"),
    )
    .reset_index()
)

# 5) Top outliers
outliers = (
    gd.assign(abs_eff_dev=(gd["factor_efectivo_post_vs_split"] - gd["factor_efectivo_post_vs_split"].median())
    ).abs()
    .sort_values("abs_eff_dev", ascending=False)
    .head(200)
    .copy()
)

# 6) Construir checks fila a fila (útil para debug)
checks = gd[
    ["fecha","bloque_base","variedad_canon","grado","destino","fecha_post_pred",
    "cajas_mll_grado_dia","cajas_split_grado_dia","cajas_post_seed",
    "factor_hidr_seed","factor_desp_seed","factor_ajuste_seed","dh_dias",
    "factor_efectivo_post_vs_split",
    "chk_nonneg_split","chk_nonneg_post","chk_hidr_range","chk_desp_range","chk_ajus_range","chk_dh_range"]
].copy()

write_parquet(checks, OUT_CHECKS)

summary = {
    "rows_gd_bd": int(len(gd)),
    "rows_dd": int(len(dd)),
    "rows_dt": int(len(dt)),
    "mass_balance_split_ok_rate": _pct(mb_ok_rate) if np.isfinite(mb_ok_rate) else None,
    "mass_balance_split_max_abs_diff": mb_max_abs,
    "range_ok_rates": {
        "hidr_range_ok": _pct(float(checks["chk_hidr_range"].mean())),
        "desp_range_ok": _pct(float(checks["chk_desp_range"].mean())),
        "ajus_range_ok": _pct(float(checks["chk_ajus_range"].mean())),
        "dh_range_ok": _pct(float(checks["chk_dh_range"].mean())),
    },
    "post_vs_split_factor_quantiles": {
        "p01": float(np.nanquantile(gd["factor_efectivo_post_vs_split"].values, 0.01)),
        "p50": float(np.nanquantile(gd["factor_efectivo_post_vs_split"].values, 0.50)),
        "p99": float(np.nanquantile(gd["factor_efectivo_post_vs_split"].values, 0.99)),
    },
    "by_destino": by_dest.to_dict(orient="records"),
    "top_outliers_preview": outliers[
        ["fecha","bloque_base","variedad_canon","grado","destino","fecha_post_pred",
        "cajas_split_grado_dia","cajas_post_seed","factor_hidr_seed","factor_desp_seed","factor_ajuste_se
ed",
        "factor_efectivo_post_vs_split"]
    ].head(30).to_dict(orient="records"),
}

OUT_SUMMARY.parent.mkdir(parents=True, exist_ok=True)
with open(OUT_SUMMARY, "w", encoding="utf-8") as f:
    json.dump(_json_safe(summary), f, ensure_ascii=False, indent=2)

print(f"OK -> {OUT_CHECKS}")
print(f"OK -> {OUT_SUMMARY}")
print(f"[AUDIT] mass_balance_ok_rate={mb_ok_rate:.4f} max_abs_diff={mb_max_abs:.10f}")

if __name__ == "__main__":
    main()

```

```

-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

IN_FACTOR = EVAL / "backtest_factor_ml2_share_grado.parquet"
IN_FINAL = EVAL / "backtest_pred_tallos_grado_dia_ml2_final.parquet"
IN_REAL = SILVER / "fact_cosecha_real_grado_dia.parquet"
IN_PRED_ML1 = GOLD / "pred_tallos_grado_dia_ml1_full.parquet"

STAMP = datetime.now().strftime("%Y%m%d_%H%M%S")

OUT_KPI_GLOBAL = EVAL / f"audit_share_grado_ml2_kpi_global_{STAMP}.parquet"
OUT_KPI_BY_ESTADO = EVAL / f"audit_share_grado_ml2_kpi_by_estado_{STAMP}.parquet"
OUT_RATIO_DIST = EVAL / f"audit_share_grado_ml2_ratio_dist_{STAMP}.parquet"
OUT_SANITY = EVAL / f"audit_share_grado_ml2_sanity_{STAMP}.parquet"
OUT_EXAMPLES_10x2 = EVAL / f"audit_share_grado_ml2_examples_10x2_{STAMP}.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _resolve_block_base(df: pd.DataFrame) -> pd.DataFrame:
    if "bloque_base" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_base"])
        return df
    if "bloque_padre" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_padre"])
        return df
    if "bloque" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque"])
        return df
    raise KeyError("No encuentro columna de bloque: bloque_base / bloque_padre / bloque")

def _safe_float(x) -> float:
    try:
        return float(x)
    except Exception:
        return float("nan")

def _wape(abs_err: pd.Series, w: pd.Series) -> float:
    a = pd.to_numeric(abs_err, errors="coerce").fillna(0.0).values
    ww = pd.to_numeric(w, errors="coerce").fillna(0.0).values
    denom = float(np.sum(ww))
    if denom <= 0:
        return float("nan")
    return float(np.sum(a * ww) / denom)

```

```

def main() -> None:
    fac = read_parquet(IN_FACTOR).copy()
    fin = read_parquet(IN_FINAL).copy()
    real = read_parquet(IN_REAL).copy()
    mll = read_parquet(IN_PRED_ML1).copy()

    # Canon
    for df in (fac, fin, mll):
        df["fecha"] = _to_date(df["fecha"])
        df["bloque_base"] = _canon_str(df["bloque_base"])
        df["grado"] = _canon_str(df["grado"])
        if "variedad_canon" in df.columns:
            df["variedad_canon"] = _canon_str(df["variedad_canon"])

    real["fecha"] = _to_date(real["fecha"])
    real = _resolve_block_base(real)
    real["grado"] = _canon_str(real["grado"])
    real["tallos_real"] = pd.to_numeric(real["tallos_real"], errors="coerce").fillna(0.0)

    # ---- Share real por día/bloque ----
    rg = real.groupby(["fecha", "bloque_base", "grado"], as_index=False).agg(tallos_real=("tallos_real", "sum"))
    tot = rg.groupby(["fecha", "bloque_base"], as_index=False).agg(tallos_real_dia=("tallos_real", "sum"))
    rg = rg.merge(tot, on=["fecha", "bloque_base"], how="left")
    rg["share_real"] = np.where(rg["tallos_real_dia"] > 0, rg["tallos_real"] / rg["tallos_real_dia"], np.nan)

    # ---- ML1 (share + tallos pred por grado) ----
    mll_need = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_mll", "tallos_pred_mll_grado_dia"]
    mll = mll[mll_need].copy()
    mll["share_grado_mll"] = pd.to_numeric(mll["share_grado_mll"], errors="coerce")
    mll["tallos_pred_mll_grado_dia"] = pd.to_numeric(mll["tallos_pred_mll_grado_dia"], errors="coerce").fillna(0.0)

    # ---- Factor: baseline share_mll y share_final ----
    fac_cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"]
    if "share_mll" in fac.columns:
        fac_use = fac[fac_cols + ["share_mll"]].copy()
    elif "share_grado_mll" in fac.columns:
        fac_use = fac.rename(columns={"share_grado_mll": "share_mll"})[fac_cols + ["share_mll"]].copy()
    else:
        raise KeyError(f"No encuentro share_mll/share_grado_mll en factor. Columnas: {list(fac.columns)}")

    # fin YA trae share_final (del apply)
    need_fin = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_final", "tallos_final_grado_dia"]
    for c in need_fin:
        if c not in fin.columns:
            raise KeyError(f"Falta columna en FINAL backtest: {c}. Columnas: {list(fin.columns)}")
    fin_use = fin[need_fin].copy()

    # ---- Dataset audit a grano ----
    df = fin_use.merge(fac_use, on=fac_cols, how="left")
    df = df.merge(mll, on=fac_cols, how="left")
    df = df.merge(rg[["fecha", "bloque_base", "grado", "share_real", "tallos_real", "tallos_real_dia"]],
                  on=["fecha", "bloque_base", "grado"], how="left")

    # Bring estado if present in factor (optional)
    if "estado" in fac.columns:
        tmp = fac[fac_cols + ["estado"]].drop_duplicates(fac_cols)
        tmp["estado"] = _canon_str(tmp["estado"])
        df = df.merge(tmp, on=fac_cols, how="left")
    else:
        df["estado"] = "NA"

    # Cast
    df["share_mll"] = pd.to_numeric(df["share_mll"], errors="coerce")
    df["share_final"] = pd.to_numeric(df["share_final"], errors="coerce")
    df["share_real"] = pd.to_numeric(df["share_real"], errors="coerce")
    df["tallos_pred_mll_grado_dia"] = pd.to_numeric(df["tallos_pred_mll_grado_dia"], errors="coerce")
    df["tallos_final_grado_dia"] = pd.to_numeric(df["tallos_final_grado_dia"], errors="coerce")
    df["tallos_real"] = pd.to_numeric(df["tallos_real"], errors="coerce")

```

```

df["tallos_real_dia"] = pd.to_numeric(df["tallos_real_dia"], errors="coerce").fillna(0.0)

# Filter where share_real exists
m = df["share_real"].notna() & df["share_ml1"].notna() & df["share_final"].notna()
d = df.loc[m].copy()

# Errors
d["err_share_ml1"] = d["share_real"] - d["share_ml1"]
d["err_share_ml2"] = d["share_real"] - d["share_final"]
d["abs_err_share_ml1"] = d["err_share_ml1"].abs()
d["abs_err_share_ml2"] = d["err_share_ml2"].abs()

d["abs_err_tallos_ml1"] = (d["tallos_real"] - d["tallos_pred_ml1_grado_dia"]).abs()
d["abs_err_tallos_ml2"] = (d["tallos_real"] - d["tallos_final_grado_dia"]).abs()

# Ratios (diagnóstico)
eps = 1e-6
d["ratio_share_final_over_ml1"] = (d["share_final"] + eps) / (d["share_ml1"] + eps)

# --- KPI GLOBAL ---
w = d["tallos_real_dia"]
kpi_global = pd.DataFrame([
    "n_rows": int(len(d)),
    "n_cycles": int(d["ciclo_id"].nunique()),
    "mae_share_ml1": _safe_float(np.nanmean(d["abs_err_share_ml1"])),
    "mae_share_ml2": _safe_float(np.nanmean(d["abs_err_share_ml2"])),
    "wape_share_ml1_wt_day": _wape(d["abs_err_share_ml1"], w),
    "wape_share_ml2_wt_day": _wape(d["abs_err_share_ml2"], w),
    "mae_abs_tallos_ml1": _safe_float(np.nanmean(d["abs_err_tallos_ml1"])),
    "mae_abs_tallos_ml2": _safe_float(np.nanmean(d["abs_err_tallos_ml2"])),
    "improvement_abs_mae_share": _safe_float(np.nanmean(d["abs_err_share_ml1"]) - np.nanmean(d["abs_err_sh
are_ml2"])),
    "improvement_abs_mae_abs_tallos": _safe_float(np.nanmean(d["abs_err_tallos_ml1"]) - np.nanmean(d["abs_
err_tallos_ml2"])),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
])

# --- KPI POR ESTADO ---
rows = []
for estado, g in d.groupby("estado"):
    ww = g["tallos_real_dia"]
    rows.append({
        "estado": str(estado),
        "n_rows": int(len(g)),
        "n_cycles": int(g["ciclo_id"].nunique()),
        "mae_share_ml1": _safe_float(np.nanmean(g["abs_err_share_ml1"])),
        "mae_share_ml2": _safe_float(np.nanmean(g["abs_err_share_ml2"])),
        "wape_share_ml1_wt_day": _wape(g["abs_err_share_ml1"], ww),
        "wape_share_ml2_wt_day": _wape(g["abs_err_share_ml2"], ww),
        "mae_abs_tallos_ml1": _safe_float(np.nanmean(g["abs_err_tallos_ml1"])),
        "mae_abs_tallos_ml2": _safe_float(np.nanmean(g["abs_err_tallos_ml2"])),
        "improvement_abs_mae_share": _safe_float(np.nanmean(g["abs_err_share_ml1"]) - np.nanmean(g["abs_er
r_share_ml2"])),
    })
kpi_by_estado = pd.DataFrame(rows).sort_values("improvement_abs_mae_share", ascending=False)

# --- RATIO DIST ---
r = d["ratio_share_final_over_ml1"].replace([np.inf, -np.inf], np.nan).dropna()
ratio_dist = pd.DataFrame([
    "n": int(len(r)),
    "ratio_min": _safe_float(r.min()),
    "ratio_p05": _safe_float(r.quantile(0.05)),
    "ratio_p25": _safe_float(r.quantile(0.25)),
    "ratio_median": _safe_float(r.median()),
    "ratio_p75": _safe_float(r.quantile(0.75)),
    "ratio_p95": _safe_float(r.quantile(0.95)),
    "ratio_max": _safe_float(r.max()),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
])

# --- SANITY: sum shares == 1 ---
if "share_final" not in fin.columns:

```

```

        raise KeyError("FINAL no tiene share_final para sanity.")
s = fin.groupby(["ciclo_id", "fecha"], as_index=False).agg(sum_share_final=("share_final", "sum"))
sanity = pd.DataFrame([
    "n_days": int(len(s)),
    "sum_share_min": _safe_float(s["sum_share_final"].min()),
    "sum_share_p25": _safe_float(s["sum_share_final"].quantile(0.25)),
    "sum_share_median": _safe_float(s["sum_share_final"].median()),
    "sum_share_p75": _safe_float(s["sum_share_final"].quantile(0.75)),
    "sum_share_max": _safe_float(s["sum_share_final"].max()),
    "pct_close_lpm_1e-6": _safe_float(np.mean(np.isclose(s["sum_share_final"].values, 1.0, atol=1e-6))),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
])

# --- EXAMPLES 10x2 (Top improve / Top worse) por reducción de error tallos ---
# agregamos por ciclo_id el impacto total
agg = d.groupby(["ciclo_id", "estado"], as_index=False).agg(
    abs_err_tallos_ml1=("abs_err_tallos_ml1", "sum"),
    abs_err_tallos_ml2=("abs_err_tallos_ml2", "sum"),
    n_rows=("ciclo_id", "size"),
)
agg["improvement_abs_tallos"] = agg["abs_err_tallos_ml1"] - agg["abs_err_tallos_ml2"]

top = agg.sort_values("improvement_abs_tallos", ascending=False).head(10).copy()
top["sample_group"] = "TOP_IMPROVE_10"
worst = agg.sort_values("improvement_abs_tallos", ascending=True).head(10).copy()
worst["sample_group"] = "TOP_WORSE_10"

examples = pd.concat([top, worst], ignore_index=True)

# attach representative bloque_base/variedad
rep = fin[["ciclo_id", "bloque_base", "variedad_canon"]].drop_duplicates("ciclo_id")
examples = examples.merge(rep, on="ciclo_id", how="left")

# Write
EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(kpi_global, OUT_KPI_GLOBAL)
write_parquet(kpi_by_estado, OUT_KPI_BY_ESTADO)
write_parquet(ratio_dist, OUT_RATIO_DIST)
write_parquet(sanity, OUT_SANITY)
write_parquet(examples, OUT_EXAMPLES_10x2)

print("\n=== ML2 SHARE GRADO AUDIT ===")
print(f"KPI global parquet : {OUT_KPI_GLOBAL}")
print(f"KPI por estado      : {OUT_KPI_BY_ESTADO}")
print(f"Ratio dist parquet   : {OUT_RATIO_DIST}")
print(f"Sanity parquet       : {OUT_SANITY}")
print(f"Examples 10x2        : {OUT_EXAMPLES_10x2}")

print("\n--- KPI GLOBAL ---")
print(kpi_global.to_string(index=False))

print("\n--- KPI POR ESTADO ---")
print(kpi_by_estado.to_string(index=False))

print("\n--- RATIO DIST ---")
print(ratio_dist.to_string(index=False))

print("\n--- SANITY ---")
print(sanity.to_string(index=False))

print("\n--- EXAMPLES 10x2 ---")
print(examples.sort_values(["sample_group", "improvement_abs_tallos"], ascending=[True, False]).to_string(
index=False))

if __name__ == "__main__":
    main()

```

```

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"

IN_FACT = EVAL / "backtest_factor_ml2_tallos_curve_dia.parquet"
IN_CICLO = SILVER / "fact_ciclo_maestro.parquet"

OUT_KPI_GLOBAL = EVAL / f"audit_tallos_curve_ml2_kpi_global_{datetime.now():%Y%m%d_%H%M%S}.parquet"
OUT_KPI_ESTADO = EVAL / f"audit_tallos_curve_ml2_kpi_by_estado_{datetime.now():%Y%m%d_%H%M%S}.parquet"
OUT_DIST = EVAL / f"audit_tallos_curve_ml2_adjust_dist_{datetime.now():%Y%m%d_%H%M%S}.parquet"
OUT_EXAMPLES = EVAL / f"audit_tallos_curve_ml2_examples_10x2_{datetime.now():%Y%m%d_%H%M%S}.parquet"

def mae(s: pd.Series) -> float:
    return float(np.nanmean(np.abs(s))) if len(s) else np.nan

def wape(y: pd.Series, yhat: pd.Series) -> float:
    denom = np.sum(np.abs(y))
    return float(np.sum(np.abs(y - yhat)) / denom) if denom > 0 else np.nan

def main() -> None:
    df = read_parquet(IN_FACT).copy()
    ciclo = read_parquet(IN_CICLO)[["ciclo_id", "estado"]].drop_duplicates("ciclo_id")

    ciclo["estado"] = ciclo["estado"].astype(str).str.upper().str.strip()
    df = df.merge(ciclo, on="ciclo_id", how="left")

    # Core numeric
    for c in ["tallos_real_dia", "tallos_pred_ml1_dia", "tallos_final_ml2_dia", "pred_ratio"]:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)

    df = df[(df["tallos_real_dia"] > 0) | (df["tallos_pred_ml1_dia"] > 0)].copy()

    # Errors
    df["err_ml1"] = df["tallos_real_dia"] - df["tallos_pred_ml1_dia"]
    df["err_ml2"] = df["tallos_real_dia"] - df["tallos_final_ml2_dia"]

    # === KPI GLOBAL ===
    kpi_g = pd.DataFrame([
        {"n_rows": len(df),
         "n_cycles": df["ciclo_id"].nunique(),
         "mae_ml1": mae(df["err_ml1"]),
         "mae_ml2": mae(df["err_ml2"]),
         "wape_ml1": wape(df["tallos_real_dia"], df["tallos_pred_ml1_dia"]),
         "wape_ml2": wape(df["tallos_real_dia"], df["tallos_final_ml2_dia"]),
         "improvement_abs_mae": mae(df["err_ml1"]) - mae(df["err_ml2"]),
         "improvement_abs_wape": wape(df["tallos_real_dia"], df["tallos_pred_ml1_dia"])
         - wape(df["tallos_real_dia"], df["tallos_final_ml2_dia"]),
         "created_at": pd.Timestamp(datetime.now()).normalize(),
        }])
    write_parquet(kpi_g, OUT_KPI_GLOBAL)

    # === DISTRIBUCIÓN AJUSTES ===
    r = df["pred_ratio"]
    dist = pd.DataFrame([

```

```

    "n": len(r),
    "min": r.min(),
    "p05": np.percentile(r, 5),
    "p25": np.percentile(r, 25),
    "median": np.median(r),
    "p75": np.percentile(r, 75),
    "p95": np.percentile(r, 95),
    "max": r.max(),
    "pct_clip": float(((r <= np.exp(-1.5)) | (r >= np.exp(1.5))).mean()),
  })
]
write_parquet(dist, OUT_DIST)

```

```

# === KPI POR ESTADO ===
rows = []
for estado, g in df.groupby("estado"):
    rows.append({
        "estado": estado,
        "n_rows": len(g),
        "n_cycles": g["ciclo_id"].nunique(),
        "mae_ml1": mae(g["err_ml1"]),
        "mae_ml2": mae(g["err_ml2"]),
        "wape_ml1": wape(g["tallos_real_dia"], g["tallos_pred_ml1_dia"]),
        "wape_ml2": wape(g["tallos_real_dia"], g["tallos_final_ml2_dia"]),
        "improvement_abs_wape": wape(g["tallos_real_dia"], g["tallos_pred_ml1_dia"])
        - wape(g["tallos_real_dia"], g["tallos_final_ml2_dia"]),
    })
write_parquet(pd.DataFrame(rows), OUT_KPI_ESTADO)

```

```

# === EJEMPLOS 10x2 ===
by_cycle = (
    df.groupby("ciclo_id", as_index=False)
    .agg(
        estado=("estado", "first"),
        wape_ml1=("tallos_pred_ml1_dia", lambda x: wape(df.loc[x.index, "tallos_real_dia"], x)),
        wape_ml2=("tallos_final_ml2_dia", lambda x: wape(df.loc[x.index, "tallos_real_dia"], x)),
    )
)
by_cycle["improvement"] = by_cycle["wape_ml1"] - by_cycle["wape_ml2"]

top = pd.concat([
    by_cycle.sort_values("improvement", ascending=False).head(10),
    by_cycle.sort_values("improvement", ascending=True).head(10),
])
write_parquet(top, OUT_EXAMPLES)

```

```

# === PRINT ===
print("\n=== AUDIT ML2 ? CURVA DIARIA TALLOS ===")
print(f"KPI global      : {OUT_KPI_GLOBAL}")
print(f"KPI por estado   : {OUT_KPI_ESTADO}")
print(f"Dist ajustes      : {OUT_DIST}")
print(f"Ejemplos 10x2     : {OUT_EXAMPLES}")
print("\n--- KPI GLOBAL ---")
print(kpi_g.to_string(index=False))
print("\n--- DIST AJUSTES ---")
print(dist.to_string(index=False))

```

```

if __name__ == "__main__":
    main()

```

```

-----
[19/164] FILE: \src\audit\audit_universe_harvest_grid_ml1.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
import json
import pandas as pd
from common.io import read_parquet, write_parquet

```

```

IN_GRID = Path("data/gold/universe_harvest_grid_ml1.parquet")

```

```

OUT_AUDIT = Path("data/audit/audit_universe_harvest_grid_mll_checks.parquet")
OUT_SUMMARY = Path("data/audit/audit_universe_harvest_grid_mll_summary.json")

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def main() -> None:
    created_at = pd.Timestamp.utcnow()
    grid = read_parquet(IN_GRID).copy()
    grid.columns = [str(c).strip() for c in grid.columns]

    need = {"ciclo_id", "fecha", "bloque_base", "variedad_canon"}
    miss = need - set(grid.columns)
    if miss:
        raise ValueError(f"universe_harvest_grid_mll sin columnas: {sorted(miss)}")

    grid["ciclo_id"] = grid["ciclo_id"].astype(str)
    grid["fecha"] = _to_date(grid["fecha"])

    key_dia = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    dup = int(grid.duplicated(subset=key_dia).sum())
    dup_rate = float(dup / max(len(grid), 1))

    rows = [
        {"metric": "rows_grid", "value": int(len(grid)), "level": "INFO", "hint": "", "created_at": created_at},
        {"metric": "dup_count_key_dia", "value": dup, "level": "WARN" if dup > 0 else "OK", "hint": "Duplicados por (ci",
        {"metric": "dup_rate_key_dia", "value": dup_rate, "level": "WARN" if dup_rate > 0 else "OK", "hint": "", "create
    ]
    audit = pd.DataFrame(rows)
    OUT_AUDIT.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(audit, OUT_AUDIT)

    summary = {
        "created_at_utc": created_at.isoformat(),
        "rows_grid": int(len(grid)),
        "dup_count_key_dia": dup,
        "dup_rate_key_dia": dup_rate,
        "horizon": {
            "min_fecha": str(pd.to_datetime(grid["fecha"].min()).date()) if len(grid) else None,
            "max_fecha": str(pd.to_datetime(grid["fecha"].max()).date()) if len(grid) else None,
        },
    }
    OUT_SUMMARY.parent.mkdir(parents=True, exist_ok=True)
    with open(OUT_SUMMARY, "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)

    print(f"OK -> {OUT_AUDIT}")
    print(f"OK -> {OUT_SUMMARY}")
    print(f"[AUDIT] dup_count_key_dia={dup:,} dup_rate={dup_rate:.6f}")

if __name__ == "__main__":
    main()

```

```

[20/164] FILE: \src\bronze\build_balanza_lc_raw.py

```

```

# src/bronze/build_balanza_lc_raw.py
from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import pandas as pd
import pyodbc
import yaml

```

```

from common.io import write_parquet

```

```

def load_settings() -> dict:

```



```

with open("config/settings.yaml", "r", encoding="utf-8") as f:
    return yaml.safe_load(f)

def _to_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce")

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def _warn(msg: str) -> None:
    print(f"[WARN] {msg}")

def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    bronze_dir.mkdir(parents=True, exist_ok=True)

    src = cfg.get("sources", {})
    server = src.get("sql_server", "")
    db = src.get("sql_db", "")
    schema = src.get("sql_schema", "dbo")
    driver = src.get("odbc_driver", "ODBC Driver 17 for SQL Server")
    user = src.get("sql_user", "")
    password = src.get("sql_password", "")

    view_lc = src.get("sql_view_balanza_lc", "BAL_View_Balanza_1C")

    if not (server and db and view_lc):
        raise ValueError("Config: define sources.sql_server, sources.sql_db y sources.sql_view_balanza_lc")
    if not (user and password):
        raise ValueError("Config: define sources.sql_user y sources.sql_password en settings.yaml")

    dt_min = pd.to_datetime(cfg.get("bronze", {}).get("balanza_fecha_min", "2024-01-01"))

    conn_str = (
        f"DRIVER={{driver}};"
        f"SERVER={server};"
        f"DATABASE={db};"
        f"UID={user};"
        f"PWD={password};"
        "TrustServerCertificate=yes;"
    )

    q_lc = f"""
    SELECT *
    FROM {schema}.{view_lc}
    WHERE Fecha >= ?
    """

    _info(f"Leyendo {schema}.{view_lc} >= {dt_min.date()} ...")
    with pyodbc.connect(conn_str) as conn:
        df = pd.read_sql(q_lc, conn, params=[dt_min])

    df.columns = [str(c).strip() for c in df.columns]

    # Tipado técnico mínimo permitido en Bronze
    if "Fecha" in df.columns:
        df["Fecha"] = _to_dt(df["Fecha"])
        bad = int(df["Fecha"].isna().sum())
        if bad > 0:
            _warn(f"Hay {bad} filas con Fecha no parseable. Se eliminan en Bronze (imposibilidad técnica).")
            df = df[df["Fecha"].notna()].copy()

    df["bronze_source"] = view_lc
    df["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")

    out = bronze_dir / "balanza_lc_raw.parquet"

```

```

write_parquet(df, out)

_info(f"OK: balanza_lc_raw={len(df)} -> {out}")
if "Fecha" in df.columns:
    _info(f"Rango fechas: {df['Fecha'].min()} -> {df['Fecha'].max()}")

if __name__ == "__main__":
    main()

-----
[21/164] FILE: \src\bronze\build_balanza_cosecha_raw.py
-----
# src/bronze/build_balanza_cosecha_raw.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import pyodbc
import yaml

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce")

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def _warn(msg: str) -> None:
    print(f"[WARN] {msg}")

def _get_existing_columns(conn: pyodbc.Connection, schema: str, table: str) -> set[str]:
    cols = pd.read_sql(
        """
        SELECT COLUMN_NAME
        FROM INFORMATION_SCHEMA.COLUMNS
        WHERE TABLE_SCHEMA = ? AND TABLE_NAME = ?
        """,
        conn,
        params=[schema, table],
    )
    return set(cols["COLUMN_NAME"].astype(str).tolist())

def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    bronze_dir.mkdir(parents=True, exist_ok=True)

    src = cfg.get("sources", {})
    server = src.get("sql_server", "")
    db = src.get("sql_db", "")
    schema = src.get("sql_schema", "dbo")
    table = src.get("sql_table_balanza", "")
    driver = src.get("odbc_driver", "ODBC Driver 17 for SQL Server")
    user = src.get("sql_user", "")
    password = src.get("sql_password", "")

```

```

if not (server and db and table):
    raise ValueError("Config: define sources.sql_server, sources.sql_db y sources.sql_table_balanza")
if not (user and password):
    raise ValueError("Config: define sources.sql_user y sources.sql_password en settings.yaml")

dist_cfg = cfg.get("dist_grado", {})
fecha_min = pd.to_datetime(dist_cfg.get("fecha_min", "2024-01-01"))

conn_str = (
    f"DRIVER={{driver}};"
    f"SERVER={server};"
    f"DATABASE={db};"
    f"UID={user};"
    f"PWD={password};"
    "TrustServerCertificate=yes;"
)

wanted = [
    "Fecha",
    "Variedad",
    "Bloque",
    "Grado",
    "Destino",
    "Tallos",
    "peso_menos_vegetativo", # <-- requerido para peso_tallo
]

_info(f"Leyendo BALANZA cosecha raw desde {db}.{schema}.{table} >= {fecha_min.date()} ...")
with pyodbc.connect(conn_str) as conn:
    existing = _get_existing_columns(conn, schema, table)

    if "Fecha" not in existing:
        raise ValueError(f"{schema}.{table} no tiene columna 'Fecha' (obligatoria).")

    selected = [c for c in wanted if c in existing]
    missing = [c for c in wanted if c not in existing]
    if missing:
        _warn(f"{schema}.{table}: columnas ausentes (se crearán como NaN en Bronze): {missing}")

    select_sql = ", ".join(selected) if selected else "Fecha"
    query = f"""
        SELECT {select_sql}
        FROM {schema}.{table}
        WHERE Fecha >= ?
    """
    df = pd.read_sql(query, conn, params=[fecha_min])

df.columns = [str(c).strip() for c in df.columns]

# Asegurar presencia de todas las columnas wanted (si faltaron en SQL)
for c in wanted:
    if c not in df.columns:
        df[c] = np.nan

# Normalizaciones técnicas permitidas (Bronze):
df["Fecha"] = _to_dt(df["Fecha"])
bad = int(df["Fecha"].isna().sum())
if bad > 0:
    _warn(f"Hay {bad} filas con Fecha no parseable. Se eliminan en Bronze por imposibilidad técnica.")
    df = df[df["Fecha"].notna()].copy()

# Casteos técnicos a string (evita mezcla rara de tipos al escribir parquet)
for c in ["Variedad", "Bloque", "Destino"]:
    if c in df.columns:
        df[c] = df[c].astype("string")

# El resto (Grado, Tallos, peso_menos_vegetativo) queda ?tal cual venga?
# (sin reglas de negocio ni caps; Silver decide)

df["bronze_source"] = "balanza_cosecha_sql"
df["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")

```

```

out_path = bronze_dir / "balanza_cosecha_raw.parquet"
write_parquet(df, out_path)

_info(f"OK: bronze balanza_cosecha_raw={len(df)} filas -> {out_path}")
_info(f"Rango fechas: {df['Fecha'].min()} -> {df['Fecha'].max()}")

if __name__ == "__main__":
    main()

-----
[22/164] FILE: \src\bronze\build_balanza_mermas_sources.py
-----
# src/bronze/build_balanza_mermas_sources.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import pyodbc
import yaml

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce")

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    bronze_dir.mkdir(parents=True, exist_ok=True)

    src = cfg.get("sources", {})
    server = src.get("sql_server", "")
    db = src.get("sql_db", "")
    schema = src.get("sql_schema", "dbo")
    driver = src.get("odbc_driver", "ODBC Driver 17 for SQL Server")
    user = src.get("sql_user", "")
    password = src.get("sql_password", "")

    if not (server and db):
        raise ValueError("Config: define sources.sql_server y sources.sql_db en settings.yaml")
    if not (user and password):
        raise ValueError("Config: define sources.sql_user y sources.sql_password en settings.yaml")

    # Recorte temporal permitido en BRONZE (solo por volumen, no por negocio)
    dt_min = pd.to_datetime(cfg.get("bronze", {}).get("balanza_fecha_min", "2025-01-01"))

    conn_str = (
        f"DRIVER={{driver}};"
        f"SERVER={server};"
        f"DATABASE={db};"
        f"UID={user};"
        f"PWD={password};"
        "TrustServerCertificate=yes;"
    )

```

```

q_2a = f"""
SELECT
    Fecha,
    Origen,
    Seccion,
    Variedad,
    codigo_actividad,
    Grado,
    peso_balanza,
    tallos,
    num_bunches
FROM {schema}.BAL_View_Balanza_2A
WHERE Fecha >= ?
"""

q_2 = f"""
SELECT
    fecha_entrega,
    Lote,
    Grado,
    Destino,
    Tallos,
    peso_netto,
    variedad,
    tipo_pelado,
    Origen,
    producto
FROM dbo.BAL_View_Balanza_2
WHERE fecha_entrega >= ?
"""

with pyodbc.connect(conn_str) as conn:
    _info(f"Leyendo BAL_View_Balanza_2A >= {dt_min.date()} ...")
    df2a = pd.read_sql(q_2a, conn, params=[dt_min])

    _info(f"Leyendo BAL_View_Balanza_2 >= {dt_min.date()} ...")
    df2 = pd.read_sql(q_2, conn, params=[dt_min])

df2a.columns = [str(c).strip() for c in df2a.columns]
df2.columns = [str(c).strip() for c in df2.columns]

# Tipos mínimos (permitidos en BRONZE)
df2a["Fecha"] = _to_dt(df2a["Fecha"])
df2 = df2[df2["fecha_entrega"].notna()].copy()
df2["fecha_entrega"] = _to_dt(df2["fecha_entrega"])

df2a["bronze_source"] = "BAL_View_Balanza_2A"
df2a["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")

df2["bronze_source"] = "BAL_View_Balanza_2"
df2["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")

out2a = bronze_dir / "balanza_2a_raw.parquet"
out2 = bronze_dir / "balanza_2_raw.parquet"

write_parquet(df2a, out2a)
write_parquet(df2, out2)

_info(f"OK: balanza_2a_raw={len(df2a)} -> {out2a}")
_info(f"OK: balanza_2_raw={len(df2)} -> {out2}")

if __name__ == "__main__":
    main()

-----
[23/164] FILE: \src\bronze\build_conteo_tallos_alturas_gyp2419_raw.py
-----
from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import pandas as pd
import yaml
import numpy as np

from openpyxl import load_workbook

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _table_to_df(path: Path, table_name: str) -> pd.DataFrame:
    """
    Lee una Excel Table (ListObject) por nombre (ej: 'Tabla6') desde cualquier hoja.
    Devuelve un DataFrame con headers correctos.
    """
    wb = load_workbook(filename=str(path), read_only=False, data_only=True)

    for ws in wb.worksheets:
        # ws.tables es dict: {table_name: Table}
        if table_name in ws.tables:
            tab = ws.tables[table_name]
            ref = tab.ref # rango tipo "A1:H120"

            cells = ws[ref]
            data = [[c.value for c in row] for row in cells]
            if not data or len(data) < 2:
                return pd.DataFrame()

            header = [str(x).strip() if x is not None else "" for x in data[0]]
            rows = data[1:]

            df = pd.DataFrame(rows, columns=header)

            # limpieza básica de columnas vacías tipo "None" o ""
            df.columns = [str(c).strip() for c in df.columns]
            df = df.loc[:, ~pd.Series(df.columns).astype(str).str.match(r"^(none)?$", case=False)]
            return df

    raise ValueError(
        f"No encontré la tabla '{table_name}' en el archivo {path.name}. "
        f"Abre el Excel y confirma el nombre exacto de la tabla."
    )

def main():
    cfg = load_settings()
    src = cfg.get("sources", {})

    p_in = Path(src.get("conteo_tallos_alturas_gyp2419_path", ""))
    if not str(p_in).strip():
        raise ValueError("Config: falta sources.conteo_tallos_alturas_gyp2419_path")
    if not p_in.exists():
        raise FileNotFoundError(f"No existe archivo: {p_in}")

    table_name = src.get("conteo_tallos_alturas_gyp2419_table", "")
    if not table_name:
        raise ValueError("Config: falta sources.conteo_tallos_alturas_gyp2419_table (ej: 'Tabla6')")

    # output
    bronze_dir = Path(cfg["paths"]["bronze"])
    bronze_dir.mkdir(parents=True, exist_ok=True)
    out_path = bronze_dir / "conteo_tallos_alturas_gyp2419_raw.parquet"

    df = _table_to_df(p_in, table_name)

    # metadata

```

```

df["__source_file"] = str(p_in)
df["__source_table"] = table_name
df["__ingested_at"] = datetime.now().isoformat(timespec="seconds")

# limpieza mínima de strings
for c in df.columns:
    if df[c].dtype == object:
        df[c] = df[c].astype(str).str.strip().replace({"None": np.nan, "nan": np.nan})

write_parquet(df, out_path)
print(f"OK: {out_path} filas={len(df)} cols={len(df.columns)}")

if __name__ == "__main__":
    main()

```

```

-----
[24/164] FILE: \src\bronze\build_fenograma_raw.py
-----

```

```

<EMPTY FILE>

```

```

-----
[25/164] FILE: \src\bronze\build_fenograma_sources.py
-----

```

```

# src/bronze/build_fenograma_sources.py
from __future__ import annotations

import os
from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd
import pyodbc
import yaml

from common.io import write_parquet

# -----
# Helpers
# -----
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def _warn(msg: str) -> None:
    print(f"[WARN] {msg}")

def _to_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce")

def _norm_cols(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df.columns = [str(c).strip() for c in df.columns]
    return df

def _read_excel_any_header(path: str, sheet_name: str, skiprows: int = 0) -> pd.DataFrame:
    df = pd.read_excel(path, sheet_name=sheet_name, skiprows=skiprows, engine="openpyxl")
    return _norm_cols(df)

def _read_indices_sheet_raw(path: str, sheet_name: str) -> pd.DataFrame:

```

```

"""
BRONZE: leer sheet COMPLETO como raw (header=None).
Requisito: NO inferir tipos. Todo se persiste como string para evitar fallos de parquet.
"""

raw = pd.read_excel(path, sheet_name=sheet_name, engine="openpyxl", header=None)
raw.columns = [f"col_{i}" for i in range(raw.shape[1])]
return raw

def _sql_connect(server: str, db: str, driver: str, user: str, password: str) -> pyodbc.Connection:
    conn_str = (
        f"DRIVER={{driver}};"
        f"SERVER={server};"
        f"DATABASE={db};"
        f"UID={user};"
        f"PWD={password};"
        "TrustServerCertificate=yes;"
    )
    return pyodbc.connect(conn_str)

def _read_balanza_filtrada(
    server: str,
    db: str,
    schema: str,
    table: str,
    driver: str,
    user: str,
    password: str,
    bloques: list[str],
    fecha_min: pd.Timestamp,
) -> pd.DataFrame:
    if not bloques:
        return pd.DataFrame(columns=["Bloque", "Fecha"])

    bloques = [str(b).strip() for b in bloques if str(b).strip() != ""]
    seen = set()
    bloques_u = []
    for b in bloques:
        if b not in seen:
            seen.add(b)
            bloques_u.append(b)
    bloques = bloques_u

    out = []
    fecha_min = pd.to_datetime(fecha_min)

    with _sql_connect(server, db, driver, user, password) as conn:
        chunk_size = 800
        for i in range(0, len(bloques), chunk_size):
            chunk = bloques[i : i + chunk_size]
            chunk_esc = [c.replace("'", "'") for c in chunk]
            chunk_sql = ", ".join([f"'{c}'" for c in chunk_esc])

            q = f"""
            SELECT Bloque, Fecha
            FROM {schema}.{table}
            WHERE Bloque IN ({chunk_sql})
            AND Fecha >= ?
            """
            tmp = pd.read_sql(q, conn, params=[fecha_min])
            out.append(tmp)

    bal = pd.concat(out, ignore_index=True) if out else pd.DataFrame(columns=["Bloque", "Fecha"])
    bal = _norm_cols(bal)
    if "Bloque" in bal.columns:
        bal["Bloque"] = bal["Bloque"].astype(str)
    if "Fecha" in bal.columns:
        bal["Fecha"] = _to_dt(bal["Fecha"])
    bal = bal[bal["Fecha"].notna()].copy()
    return bal

```



```

def _force_all_to_string(df: pd.DataFrame) -> pd.DataFrame:
    """
    BRONZE RAW (índices): forzar 100% a string.
    Esto evita inferencias tipo int64/float en columnas con encabezados/etiquetas mezcladas.
    """
    df = df.copy()
    for c in df.columns:
        df[c] = df[c].astype("string")
    return df

def _force_safe_types_for_parquet(df: pd.DataFrame, prefer_str_cols: list[str] | None = None) -> pd.DataFrame:
    """
    Fenograma XLSM: tipado seguro para parquet.
    """
    df = df.copy()
    prefer_str_cols = prefer_str_cols or []

    for c in prefer_str_cols:
        if c in df.columns:
            df[c] = df[c].astype("string")

    obj_cols = [c for c in df.columns if df[c].dtype == "object"]
    for c in obj_cols:
        s = df[c]
        sample = s.dropna()
        if sample.empty:
            df[c] = s.astype("string")
            continue

        types = sample.map(lambda x: type(x)).value_counts()
        if len(types) > 1:
            df[c] = s.astype("string")
            continue

        if types.index[0] is str:
            df[c] = s.astype("string")
            continue

        try:
            df[c] = pd.to_numeric(s, errors="raise")
        except Exception:
            df[c] = s.astype("string")

    return df

def _profile_object_cols(df: pd.DataFrame, top_n: int = 5) -> None:
    obj_cols = [c for c in df.columns if str(df[c].dtype) in ("object", "string")]
    if not obj_cols:
        return
    _info(f"Perfil columnas tipo object/string (n={len(obj_cols)}):")
    for c in obj_cols[: min(len(obj_cols), 25)]:
        vals = df[c].dropna().astype(str).head(top_n).tolist()
        _info(f" - {c}: dtype={df[c].dtype} ejemplos={vals}")

def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    bronze_dir.mkdir(parents=True, exist_ok=True)

    # -----
    # Fenograma XLSM (activo) - BRONZE raw con tipado seguro
    # -----
    ruta_xlsm = cfg["sources"].get("fenograma_path", "")
    hoja_fenograma = cfg["sources"].get("fenograma_sheet", "")
    skip_rows = int(cfg.get("sources", {}).get("fenograma_skiprows", 7))

    if not (ruta_xlsm and hoja_fenograma):

```

```

        raise ValueError("Config: define sources.fenograma_path y sources.fenograma_sheet")

df_xlsm = _read_excel_any_header(ruta_xlsm, hoja_fenograma, skiprows=skip_rows)

prefer_str = ["Fiesta", "Bloques", "Bloque", "Area", "Area ", "Variedad", "S/P", "Pruebas", "Pruebas "]
df_xlsm = _force_safe_types_for_parquet(df_xlsm, prefer_str_cols=prefer_str)

df_xlsm["bronze_source"] = "fenograma_xlsm"
df_xlsm["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")

_profile_object_cols(df_xlsm)

out_xlsm = bronze_dir / "fenograma_xlsm_raw.parquet"
write_parquet(df_xlsm, out_xlsm)
_info(f"OK: fenograma_xlsm_raw={len(df_xlsm)} -> {out_xlsm}")

bloques = (
    df_xlsm.get("Bloques", pd.Series([], dtype="string"))
    .dropna()
    .astype(str)
    .str.strip()
    .replace("", np.nan)
    .dropna()
    .unique()
    .tolist()
)

# -----
# Índices históricos XL/CLO (raw 100% string)
# -----
ruta_indices = cfg.get("sources", {}).get("indices_path", "")
sheet_xl = cfg.get("sources", {}).get("indices_sheet_xl", "")
sheet_clo = cfg.get("sources", {}).get("indices_sheet_clo", "")

if ruta_indices and sheet_xl:
    raw_xl = _read_indices_sheet_raw(ruta_indices, sheet_xl)
    raw_xl = _force_all_to_string(raw_xl)
    raw_xl["bronze_source"] = "indices_xl"
    raw_xl["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")
    out_xl = bronze_dir / "indices_xl_raw.parquet"
    write_parquet(raw_xl, out_xl)
    _info(f"OK: indices_xl_raw={len(raw_xl)} -> {out_xl}")
else:
    _warn("No se generó indices_xl_raw (faltan sources.indices_path o sources.indices_sheet_xl).")

if ruta_indices and sheet_clo:
    raw_clo = _read_indices_sheet_raw(ruta_indices, sheet_clo)
    raw_clo = _force_all_to_string(raw_clo)
    raw_clo["bronze_source"] = "indices_clo"
    raw_clo["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")
    out_clo = bronze_dir / "indices_clo_raw.parquet"
    write_parquet(raw_clo, out_clo)
    _info(f"OK: indices_clo_raw={len(raw_clo)} -> {out_clo}")
else:
    _warn("No se generó indices_clo_raw (faltan sources.indices_path o sources.indices_sheet_clo).")

# -----
# Balanza SQL (raw filtrado)
# -----
fecha_min_hist = pd.to_datetime(cfg.get("sources", {}).get("fecha_min_hist", "2024-01-01"))

sql_server = cfg["sources"].get("sql_server", "")
sql_db = cfg["sources"].get("sql_db", "")
sql_schema = cfg["sources"].get("sql_schema", "dbo")
sql_table = cfg["sources"].get("sql_table", "")
odbc_driver = cfg["sources"].get("odbc_driver", "ODBC Driver 17 for SQL Server")

sql_user = cfg["sources"].get("sql_user", "") or os.environ.get("SQL_USER", "")
sql_password = cfg["sources"].get("sql_password", "") or os.environ.get("SQL_PASSWORD", "")

if sql_server and sql_db and sql_table and sql_user and sql_password:
    bal = _read_balanza_filtrada(

```

```

        server=sql_server,
        db=sql_db,
        schema=sql_schema,
        table=sql_table,
        driver=odbc_driver,
        user=sql_user,
        password=sql_password,
        bloques=bloques,
        fecha_min=fecha_min_hist,
    )
    bal["bronze_source"] = "balanza_sql"
    bal["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")

    out_bal = bronze_dir / "balanza_bloque_fecha_raw.parquet"
    write_parquet(bal, out_bal)
    _info(f"OK: balanza_bloque_fecha_raw={len(bal)} -> {out_bal}")
else:
    _warn(
        "No se generó balanza_bloque_fecha_raw. "
        "Revisa sources.sql_server/sql_db/sql_table y variables de entorno SQL_USER/SQL_PASSWORD."
    )

if __name__ == "__main__":
    main()

-----
[26/164] FILE: \src\bronze\build_ghu_maestro_horas.py
-----
# src/bronze/build_ghu_maestro_horas.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import pyodbc
import yaml

from common.io import write_parquet

# -----
# Config / helpers
# -----
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce")

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def _warn(msg: str) -> None:
    print(f"[WARN] {msg}")

def _read_available_columns(conn: pyodbc.Connection, db: str, schema: str, table_or_view: str) -> set[str]:
    q = """
        SELECT COLUMN_NAME
        FROM INFORMATION_SCHEMA.COLUMNS
        WHERE TABLE_CATALOG = ? AND TABLE_SCHEMA = ? AND TABLE_NAME = ?
    """
    cols = pd.read_sql(q, conn, params=[db, schema, table_or_view])
    return set(cols["COLUMN_NAME"].astype(str).tolist())

```

```

# -----
# Main
# -----
def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    bronze_dir.mkdir(parents=True, exist_ok=True)

    src = cfg.get("sources", {})
    server = src.get("sql_server", "")
    driver = src.get("odbc_driver", "ODBC Driver 17 for SQL Server")
    user = src.get("sql_user", "")
    password = src.get("sql_password", "")

    db_gh = src.get("sql_db_gh", "GestionHumana")
    schema_gh = src.get("sql_schema_ghu", "dbo")
    view_ghu = src.get("sql_view_ghu_maestro_horas", "GHU_View_Maestro_Horas")

    if not server:
        raise ValueError("Config: define sources.sql_server en settings.yaml")
    if not (user and password):
        raise ValueError("Config: define sources.sql_user y sources.sql_password en settings.yaml")

    dt_min = pd.to_datetime(cfg.get("bronze", {}).get("ghu_fecha_min", "2024-01-01"))

    conn_str = (
        f"DRIVER={{driver}};"
        f"SERVER={server};"
        f"DATABASE={db_gh};"
        f"UID={user};"
        f"PWD={password};"
        "TrustServerCertificate=yes;"
    )

    _info(f"Leyendo {db_gh}.{schema_gh}.{view_ghu} desde {dt_min.date()} ...")

    with pyodbc.connect(conn_str) as conn:
        existing = _read_available_columns(conn, db_gh, schema_gh, view_ghu)

        # Base columns (las que ya usabas)
        wanted = [
            "fecha",
            "codigo_personal",
            "nombres",
            "actividad",
            "codigo_actividad",
            "tipo_actividad",
            "area_trabajada",
            "area_original",
            "horas_acumula",
            "bloque",
            "hora_ingreso",
            "hora_salida",
            "horas_presenciales",
            "unidades_producidas",
        ]

        # Nueva columna requerida por Silver (si existe en la vista)
        # Si tu vista la nombra distinto, agrega alias en settings o aquí.
        extra_candidates = ["unidad_medida"]
        for c in extra_candidates:
            if c not in wanted:
                wanted.append(c)

        selected = [c for c in wanted if c in existing]
        missing = [c for c in wanted if c not in existing]
        if missing:
            _warn(f"{db_gh}.{schema_gh}.{view_ghu}: columnas ausentes (se crearán NaN): {missing}")

        select_sql = ", ".join(selected) if selected else ""

```

```

query = f"""
    SELECT {select_sql}
    FROM {schema_gh}.{view_ghu}
    WHERE fecha >= ?
"""

df = pd.read_sql(query, conn, params=[dt_min])

df.columns = [str(c).strip() for c in df.columns]

# Asegurar presencia de columnas ausentes como NaN (robusto)
for c in wanted:
    if c not in df.columns:
        df[c] = np.nan

# Normalizaciones mínimas BRONZE
df["fecha"] = _to_dt(df["fecha"])
n_bad = int(df["fecha"].isna().sum())
if n_bad > 0:
    _warn(f"Hay {n_bad} filas con fecha no parseable. Se eliminan en BRONZE por imposibilidad técnica.")
    df = df[df["fecha"].notna()].copy()

# Metadatos
df["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")

out_path = bronze_dir / "ghu_maestro_horas.parquet"
write_parquet(df, out_path)

_info(f"OK: bronze ghu_maestro_horas={len(df)} filas -> {out_path}")
_info(f"Rango fechas: {df['fecha'].min()} -> {df['fecha'].max()}")
if "tipo_actividad" in df.columns:
    _info(f"Tipo_actividad (top): {df['tipo_actividad'].astype(str).value_counts(dropna=False).head(10).to_dict()}")

if __name__ == "__main__":
    main()

-----
[27/164] FILE: \src\bronze\build_personal_sources.py
-----
# src/bronze/build_personal_sources.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import pyodbc
import yaml

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_int64(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])

```

```

bronze_dir.mkdir(parents=True, exist_ok=True)

src = cfg.get("sources", {})
server = src.get("sql_server", "")
driver = src.get("odbc_driver", "ODBC Driver 17 for SQL Server")
user = src.get("sql_user", "")
password = src.get("sql_password", "")

db = src.get("sql_db_gh", src.get("sql_db_ghu", "GestionHumana"))
schema = src.get("sql_schema_ghu", "dbo")
table_personal = src.get("sql_table_personal", "Personal")

if not server:
    raise ValueError("Config: define sources.sql_server en settings.yaml")
if not (user and password):
    raise ValueError("Config: define sources.sql_user y sources.sql_password en settings.yaml")

conn_str = (
    f"DRIVER={{driver}};"
    f"SERVER={server};"
    f"DATABASE={db};"
    f"UID={user};"
    f"PWD={password};"
    "TrustServerCertificate=yes;"
)

query = f"""
SELECT
    Codigo_Personal AS codigo_personal,
    Activo_o_Inactivo
FROM {schema}.{table_personal}
"""

_info(f"Leyendo Personal desde {db}.{schema}.{table_personal} ...")
with pyodbc.connect(conn_str) as conn:
    df = pd.read_sql(query, conn)

df.columns = [str(c).strip() for c in df.columns]

if "codigo_personal" not in df.columns:
    raise ValueError("Personal: no se obtuvo columna 'codigo_personal'. Revisa query/tabla.")
if "Activo_o_Inactivo" not in df.columns:
    raise ValueError("Personal: no se obtuvo columna 'Activo_o_Inactivo'. Revisa query/tabla.")

df["codigo_personal"] = _to_int64(df["codigo_personal"])
df["Activo_o_Inactivo"] = df["Activo_o_Inactivo"].astype(str).str.strip()

# Quitar nulos imposibles técnicamente (sin código)
df = df[df["codigo_personal"].notna()].copy()

# Metadatos bronze
df["bronze_source"] = "personal_sql"
df["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")

out_path = bronze_dir / "personal_raw.parquet"
write_parquet(df, out_path)

_info(f"OK: bronze personal_raw={len(df)} filas -> {out_path}")
_info(f"Activo_o_Inactivo (top): {df['Activo_o_Inactivo'].value_counts(dropna=False).head(10).to_dict()}")

if __name__ == "__main__":
    main()

-----
[28/164] FILE: \src\bronze\build_ventas_sources.py
-----
# src/bronze/build_ventas_sources.py
from __future__ import annotations

from pathlib import Path

```

```

from datetime import datetime
import pandas as pd
import yaml

from common.io import write_parquet

# -----
# Helpers
# -----
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def _read_excel_raw(path: str, sheet_name: str) -> pd.DataFrame:
    """
    BRONZE ventas:
    - header=None
    - no inferir tipos
    - columnas col_0..col_n
    """
    raw = pd.read_excel(path, sheet_name=sheet_name, header=None, engine="openpyxl")
    raw.columns = [f"col_{i}" for i in range(raw.shape[1])]
    return raw

def _force_all_string(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    for c in df.columns:
        df[c] = df[c].astype("string")
    return df

def main() -> None:
    cfg = load_settings()
    bronze_dir = Path(cfg["paths"]["bronze"])
    bronze_dir.mkdir(parents=True, exist_ok=True)

    ventas_cfg = cfg.get("ventas", {})
    sheet = ventas_cfg.get("ventas_sheet", "FULLES 10K")

    paths = {
        "ventas_2025": ventas_cfg.get("ventas_2025_path", ""),
        "ventas_2026": ventas_cfg.get("ventas_2026_path", "")
    }

    for tag, path in paths.items():
        if not path:
            _info(f"{tag}: no definido, se omite.")
            continue

        df = _read_excel_raw(path, sheet_name=sheet)
        df = _force_all_string(df)

        df["bronze_source"] = tag
        df["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")

        out = bronze_dir / f"{tag}_raw.parquet"
        write_parquet(df, out)

        _info(f"OK: {tag}_raw={len(df)} filas -> {out}")

if __name__ == "__main__":
    main()

```

```
-----  
[29/164] FILE: \src\bronze\build_weather_hour_a4.py  
-----
```

```
# src/bronze/build_weather_hour_a4.py  
from __future__ import annotations
```

```
from pathlib import Path  
from datetime import datetime  
import pandas as pd  
import numpy as np  
import pyodbc  
import yaml
```

```
from common.io import write_parquet
```

```
def load_settings() -> dict:  
    with open("config/settings.yaml", "r", encoding="utf-8") as f:  
        return yaml.safe_load(f)
```

```
def _to_dt(s: pd.Series) -> pd.Series:  
    return pd.to_datetime(s, errors="coerce")
```

```
def _to_num(s: pd.Series) -> pd.Series:  
    return pd.to_numeric(s, errors="coerce")
```

```
def _info(msg: str) -> None:  
    print(f"[INFO] {msg}")
```

```
def _warn(msg: str) -> None:  
    print(f"[WARN] {msg}")
```

```
def _read_weather_table(  
    conn: pyodbc.Connection,  
    schema: str,  
    table: str,  
    dt_min: pd.Timestamp,  
) -> pd.DataFrame:
```

```
    wanted = [  
        "fecha",  
        "rainfall_mm",  
        "et",  
        "temp_avg",  
        "wind_dir_of_avg",  
        "wind_run",  
        "solar_rad_avg",  
        "solar_rad_hi",  
        "solar_energy",  
        "hum_last",  
        "uv_index_avg",  
        "wind_speed_avg",  
    ]
```

```
    cols_df = pd.read_sql(  
        """  
        SELECT COLUMN_NAME  
        FROM INFORMATION_SCHEMA.COLUMNS  
        WHERE TABLE_SCHEMA = ? AND TABLE_NAME = ?  
        """,  
        conn,  
        params=[schema, table],  
    )
```

```
    existing = set(cols_df["COLUMN_NAME"].astype(str).tolist())  
    if "fecha" not in existing:  
        raise ValueError(f"{schema}.{table} no tiene columna 'fecha' (obligatoria).")
```

```
    selected = [c for c in wanted if c in existing]
```



```

missing = [c for c in wanted if c not in existing]
if missing:
    _warn(f"{schema}.{table}: columnas ausentes (se crearán NaN): {missing}")

select_sql = ", ".join(selected) if selected else "fecha"
q = f"""
    SELECT {select_sql}
    FROM {schema}.{table}
    WHERE fecha >= ?
    ORDER BY fecha
"""
df = pd.read_sql(q, conn, params=[dt_min])
df.columns = [str(c).strip() for c in df.columns]

for c in wanted:
    if c not in df.columns:
        df[c] = np.nan

df["fecha"] = _to_dt(df["fecha"])
df = df[df["fecha"].notna()].copy()

for c in [
    "rainfall_mmm", "et", "temp_avg", "wind_run",
    "solar_rad_avg", "solar_rad_hi", "solar_energy",
    "hum_last", "uv_index_avg", "wind_speed_avg",
]:
    df[c] = _to_num(df[c])

return df

def main() -> None:
    cfg = load_settings()
    bronze_dir = Path(cfg["paths"]["bronze"])
    bronze_dir.mkdir(parents=True, exist_ok=True)

    src = cfg.get("sources", {})
    server = src.get("sql_server", "")
    driver = src.get("odbc_driver", "ODBC Driver 17 for SQL Server")
    user = src.get("sql_user", "")
    password = src.get("sql_password", "")

    db_weather = src.get("sql_db_weather", "WeatherStation")
    schema = src.get("sql_schema_weather", "dbo")
    table_a4 = src.get("sql_table_weather_a4", "Weather_Station_Hour_A4")

    if not server:
        raise ValueError("Config: define sources.sql_server en settings.yaml")
    if not (user and password):
        raise ValueError("Config: define sources.sql_user y sources.sql_password en settings.yaml")

    dt_min = pd.to_datetime(cfg.get("bronze", {}).get("weather_fecha_min", "2024-01-01 01:00:00"))

    conn_str = (
        f"DRIVER={{driver}};"
        f"SERVER={server};"
        f"DATABASE={db_weather};"
        f"UID={user};"
        f"PWD={password};"
        "TrustServerCertificate=yes;"
    )

    _info(f"Leyendo clima A4 desde {db_weather}.{schema}.{table_a4} >= {dt_min} ...")
    with pyodbc.connect(conn_str) as conn:
        try:
            df = _read_weather_table(conn, schema, table_a4, dt_min)
        except Exception as e:
            # BRONZE debe fallar si tú lo decides; aquí lo dejo estricto por defecto.
            raise RuntimeError(f"No pude leer A4 ({schema}.{table_a4}). Error: {e}") from e

    df["station"] = "A4"
    df["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")

```

```

out_path = bronze_dir / "weather_hour_a4.parquet"
write_parquet(df, out_path)

_info(f"OK: bronze weather_hour_a4={len(df)} filas -> {out_path}")
_info(f"Rango fechas: {df['fecha'].min()} -> {df['fecha'].max()}")

if __name__ == "__main__":
    main()

-----
[30/164] FILE: \src\bronze\build_weather_hour_main.py
-----
# src/bronze/build_weather_hour_main.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import pyodbc
import yaml

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce")

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def _warn(msg: str) -> None:
    print(f"[WARN] {msg}")

def _read_weather_table(
    conn: pyodbc.Connection,
    schema: str,
    table: str,
    dt_min: pd.Timestamp,
) -> pd.DataFrame:
    wanted = [
        "fecha",
        "rainfall_mm",
        "et",
        "temp_avg",
        "wind_dir_of_avg",
        "wind_run",
        "solar_rad_avg",
        "solar_rad_hi",
        "solar_energy",
        "hum_last",
        "uv_index_avg",
        "wind_speed_avg",
    ]

    cols_df = pd.read_sql(
        """

```

```

SELECT COLUMN_NAME
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA = ? AND TABLE_NAME = ?
""",
conn,
params=[schema, table],
)
existing = set(cols_df["COLUMN_NAME"].astype(str).tolist())
if "fecha" not in existing:
    raise ValueError(f"{schema}.{table} no tiene columna 'fecha' (obligatoria).")

selected = [c for c in wanted if c in existing]
missing = [c for c in wanted if c not in existing]
if missing:
    _warn(f"{schema}.{table}: columnas ausentes (se crearán NaN): {missing}")

select_sql = ", ".join(selected) if selected else "fecha"
q = f"""
SELECT {select_sql}
FROM {schema}.{table}
WHERE fecha >= ?
ORDER BY fecha
"""
df = pd.read_sql(q, conn, params=[dt_min])
df.columns = [str(c).strip() for c in df.columns]

# Asegurar presencia de todas las wanted
for c in wanted:
    if c not in df.columns:
        df[c] = np.nan

df["fecha"] = _to_dt(df["fecha"])
df = df[df["fecha"].notna()].copy()

for c in [
    "rainfall_mm", "et", "temp_avg", "wind_run",
    "solar_rad_avg", "solar_rad_hi", "solar_energy",
    "hum_last", "uv_index_avg", "wind_speed_avg",
]:
    df[c] = _to_num(df[c])

return df

```

```

def main() -> None:
    cfg = load_settings()
    bronze_dir = Path(cfg["paths"]["bronze"])
    bronze_dir.mkdir(parents=True, exist_ok=True)

    src = cfg.get("sources", {})
    server = src.get("sql_server", "")
    driver = src.get("odbc_driver", "ODBC Driver 17 for SQL Server")
    user = src.get("sql_user", "")
    password = src.get("sql_password", "")

    db_weather = src.get("sql_db_weather", "WeatherStation")
    schema = src.get("sql_schema_weather", "dbo")
    table_main = src.get("sql_table_weather_main", "Weather_Station_Hour")

    if not server:
        raise ValueError("Config: define sources.sql_server en settings.yaml")
    if not (user and password):
        raise ValueError("Config: define sources.sql_user y sources.sql_password en settings.yaml")

    dt_min = pd.to_datetime(cfg.get("bronze", {}).get("weather_fecha_min", "2024-01-01 01:00:00"))

    conn_str = (
        f"DRIVER={{driver}};"
        f"SERVER={server};"
        f"DATABASE={db_weather};"
        f"UID={user};"
        f"PWD={password};"
    )

```

```

        "TrustServerCertificate=yes;"
    )

    _info(f"Leyendo clima MAIN desde {db_weather}.{schema}.{table_main} >= {dt_min} ...")
    with pyodbc.connect(conn_str) as conn:
        df = _read_weather_table(conn, schema, table_main, dt_min)

    df["station"] = "MAIN"
    df["bronze_extracted_at"] = datetime.now().isoformat(timespec="seconds")

    out_path = bronze_dir / "weather_hour_main.parquet"
    write_parquet(df, out_path)

    _info(f"OK: bronze weather_hour_main={len(df)} filas -> {out_path}")
    _info(f"Rango fechas: {df['fecha'].min()} -> {df['fecha'].max()}")

if __name__ == "__main__":
    main()

```

```

-----
[31/164] FILE: \src\common\ids.py
-----
from __future__ import annotations

import re
import hashlib
import pandas as pd

def norm_text(x: object) -> str:
    if x is None or (isinstance(x, float) and pd.isna(x)):
        return ""
    s = str(x).strip().upper()
    s = re.sub(r"\s+", " ", s)
    return s

def make_bloque_id(bloque: object) -> str:
    return norm_text(bloque)

def make_variedad_id(variedad: object) -> str:
    return norm_text(variedad)

def make_ciclo_id(bloque_id: str, variedad_id: str, tipo_sp: str, fecha_sp) -> str:
    # fecha_sp se espera como Timestamp/date.
    f = pd.to_datetime(fecha_sp).date().isoformat()
    raw = f"{bloque_id}|{variedad_id}|{tipo_sp}|{f}"
    return hashlib.shal(raw.encode("utf-8")).hexdigest()

-----
[32/164] FILE: \src\common\io.py
-----
from __future__ import annotations

from pathlib import Path
import pandas as pd

def ensure_parent_dir(path: Path) -> None:
    path.parent.mkdir(parents=True, exist_ok=True)

def read_parquet(path: Path) -> pd.DataFrame:
    return pd.read_parquet(path)

def write_parquet(df: pd.DataFrame, path: Path) -> None:
    ensure_parent_dir(path)
    df.to_parquet(path, index=False)

def read_excel(path: Path, sheet_name: str) -> pd.DataFrame:
    return pd.read_excel(path, sheet_name=sheet_name)

```

```
-----  
[33/164] FILE: \src\common\timegrid.py  
-----
```

```
from __future__ import annotations
```

```
import pandas as pd
```

```
def build_grid_ciclo_fecha(fact_ciclo_maestro: pd.DataFrame, horizon_days: int) -> pd.DataFrame:  
    # Espera columnas: ciclo_id, fecha_sp (date/timestamp)  
    base = fact_ciclo_maestro[["ciclo_id", "fecha_sp"]].copy()  
    base["fecha_sp"] = pd.to_datetime(base["fecha_sp"]).dt.normalize()  
  
    rows = []  
    for ciclo_id, fecha_sp in base.itertuples(index=False):  
        start = fecha_sp  
        end = start + pd.Timedelta(days=horizon_days)  
        fechas = pd.date_range(start=start, end=end, freq="D")  
        tmp = pd.DataFrame({"ciclo_id": ciclo_id, "fecha": fechas})  
        rows.append(tmp)  
  
    grid = pd.concat(rows, ignore_index=True) if rows else pd.DataFrame(columns=["ciclo_id", "fecha"])  
    # calendarios útiles  
    grid["fecha"] = pd.to_datetime(grid["fecha"]).dt.normalize()  
    # Semana_ (ISO o Sunday-based; ajustar a tu estándar)  
    # Aquí uso ISO week por defecto:  
    isocal = grid["fecha"].dt.isocalendar()  
    grid["anio"] = isocal["year"].astype(int)  
    grid["semana_iso"] = isocal["week"].astype(int)  
    grid["mes"] = grid["fecha"].dt.month.astype(int)  
    return grid
```

```
-----  
[34/164] FILE: \src\eval\build_kpis_ajuste_poscosecha_ml2.py  
-----
```

```
from __future__ import annotations
```

```
from pathlib import Path  
from datetime import datetime  
import numpy as np  
import pandas as pd
```

```
from src.common.io import read_parquet, write_parquet
```

```
def _project_root() -> Path:  
    return Path(__file__).resolve().parents[2]
```

```
ROOT = _project_root()  
DATA = ROOT / "data"  
EVAL = DATA / "eval" / "ml2"  
SILVER = DATA / "silver"  
GOLD = DATA / "gold"
```

```
IN_FINAL = GOLD / "pred_poscosecha_ml2_ajuste_grado_dia_bloque_destino_final.parquet"  
IN_REAL_MA = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"
```

```
OUT_G = EVAL / "ml2_ajuste_poscosecha_eval_global.parquet"  
OUT_D = EVAL / "ml2_ajuste_poscosecha_eval_by_destino.parquet"  
OUT_DIST = EVAL / "ml2_ajuste_poscosecha_eval_ratio_dist.parquet"
```

```
def _to_date(s: pd.Series) -> pd.Series:  
    return pd.to_datetime(s, errors="coerce").dt.normalize()
```

```
def _canon_str(s: pd.Series) -> pd.Series:  
    return s.astype(str).str.upper().str.strip()
```

```
def _resolve_fecha_post_pred(df: pd.DataFrame) -> str:
```

```

for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml1", "fecha_post_pred"]:  
    if c in df.columns:  
        return c  
raise KeyError("No encuentro fecha_post_pred_* en final.")

def _resolve_ajuste_ml1(df: pd.DataFrame) -> str:  
    for c in ["factor_ajuste_ml1", "ajuste_ml1", "factor_ajuste_seed", "factor_ajuste"]:  
        if c in df.columns:  
            return c  
    raise KeyError("No encuentro ajuste ML1 en final.")

def _weight_series(df: pd.DataFrame) -> pd.Series:  
    for c in ["tallos_w", "tallos", "tallos_total_ml2", "tallos_total"]:  
        if c in df.columns:  
            return pd.to_numeric(df[c], errors="coerce").fillna(0.0)  
    return pd.Series(1.0, index=df.index, dtype="float64")

def _wmae_log_ratio(ratio: pd.Series, w: pd.Series) -> float:  
    r = pd.to_numeric(ratio, errors="coerce")  
    w = pd.to_numeric(w, errors="coerce").fillna(0.0)  
    lr = np.log(r.replace(0, np.nan))  
    m = lr.notna() & np.isfinite(lr) & w.notna() & (w >= 0)  
    if not bool(m.any()):  
        return np.nan  
    denom = float(w[m].sum())  
    if denom <= 0:  
        return np.nan  
    return float((lr[m].abs() * w[m]).sum() / denom)

def main() -> None:  
    df = read_parquet(IN_FINAL).copy()  
    df.columns = [str(c).strip() for c in df.columns]  
  
    df["fecha"] = _to_date(df["fecha"])  
    df["destino"] = _canon_str(df["destino"])  
  
    fecha_post_col = _resolve_fecha_post_pred(df)  
    df[fecha_post_col] = _to_date(df[fecha_post_col])  
  
    aj_ml1_col = _resolve_ajuste_ml1(df)  
    df[aj_ml1_col] = pd.to_numeric(df[aj_ml1_col], errors="coerce")  
  
    if "factor_ajuste_final" not in df.columns:  
        raise KeyError("No encuentro factor_ajuste_final en pred_poscosecha_ml2_ajuste..._final.")  
  
    df["factor_ajuste_final"] = pd.to_numeric(df["factor_ajuste_final"], errors="coerce")  
  
    # real  
    real = read_parquet(IN_REAL_MA).copy()  
    real.columns = [str(c).strip() for c in real.columns]  
    real["fecha_post"] = _to_date(real["fecha_post"])  
    real["destino"] = _canon_str(real["destino"])  
  
    if "factor_ajuste" in real.columns:  
        rc = "factor_ajuste"  
    elif "ajuste" in real.columns:  
        rc = "ajuste"  
    else:  
        raise ValueError("Real MA no trae factor_ajuste ni ajuste.")  
    real[rc] = pd.to_numeric(real[rc], errors="coerce")  
  
    real2 = (  
        real.groupby(["fecha_post", "destino"], dropna=False, as_index=False)  
            .agg(factor_ajuste_real=(rc, "median"))  
    )  
  
    df = df.merge(  
        real2.rename(columns={"fecha_post": "fecha_post_key"}),

```

```

        left_on=[fecha_post_col, "destino"],
        right_on=["fecha_post_key", "destino"],
        how="left",
    )

w = _weight_series(df)

# solo con real
m = df["factor_ajuste_real"].notna() & df[aj_mll_col].notna() & df["factor_ajuste_final"].notna()
d = df.loc[m].copy()
if d.empty:
    raise ValueError("No hay filas con real para KPIs de ajuste.")

w2 = _weight_series(d)

# ratios (pred/real)
d["ratio_mll"] = d[aj_mll_col] / d["factor_ajuste_real"].replace(0, np.nan)
d["ratio_ml2"] = d["factor_ajuste_final"] / d["factor_ajuste_real"].replace(0, np.nan)

out_g = pd.DataFrame([
    {
        "n_rows": int(len(d)),
        "n_dates": int(pd.to_datetime(d[fecha_post_col], errors="coerce").dt.normalize().nunique()),
        "mae_log_mll_w": _wmae_log_ratio(d["ratio_mll"], w2),
        "mae_log_ml2_w": _wmae_log_ratio(d["ratio_ml2"], w2),
        "median_ratio_mll": float(np.nanmedian(pd.to_numeric(d["ratio_mll"], errors="coerce").values)),
        "median_ratio_ml2": float(np.nanmedian(pd.to_numeric(d["ratio_ml2"], errors="coerce").values)),
        "p05_ratio_ml2": float(np.nanpercentile(pd.to_numeric(d["ratio_ml2"], errors="coerce").dropna().values
, 5)),
        "p95_ratio_ml2": float(np.nanpercentile(pd.to_numeric(d["ratio_ml2"], errors="coerce").dropna().values
, 95)),
        "improvement_abs_mae_log_w": float(_wmae_log_ratio(d["ratio_mll"], w2) - _wmae_log_ratio(d["ratio_ml2"]
], w2)),
        "created_at": pd.Timestamp(datetime.utcnow()).normalize(),
    }
])

rows = []
for dest, g in d.groupby("destino"):
    ww = _weight_series(g)
    rows.append({
        "destino": str(dest),
        "n_rows": int(len(g)),
        "n_dates": int(pd.to_datetime(g[fecha_post_col], errors="coerce").dt.normalize().nunique()),
        "mae_log_mll_w": _wmae_log_ratio(g["ratio_mll"], ww),
        "mae_log_ml2_w": _wmae_log_ratio(g["ratio_ml2"], ww),
        "median_ratio_ml2": float(np.nanmedian(pd.to_numeric(g["ratio_ml2"], errors="coerce").values)),
        "improvement_abs_mae_log_w": float(_wmae_log_ratio(g["ratio_mll"], ww) - _wmae_log_ratio(g["ratio_
ml2"], ww)),
    })
out_d = pd.DataFrame(rows).sort_values("improvement_abs_mae_log_w", ascending=False)

r = pd.to_numeric(d["ratio_ml2"], errors="coerce")
out_dist = pd.DataFrame([
    {
        "n": int(len(d)),
        "ratio_min": float(np.nanmin(r.values)),
        "ratio_p05": float(np.nanpercentile(r.dropna().values, 5)),
        "ratio_p25": float(np.nanpercentile(r.dropna().values, 25)),
        "ratio_median": float(np.nanmedian(r.values)),
        "ratio_p75": float(np.nanpercentile(r.dropna().values, 75)),
        "ratio_p95": float(np.nanpercentile(r.dropna().values, 95)),
        "ratio_max": float(np.nanmax(r.values)),
        "created_at": pd.Timestamp(datetime.utcnow()).normalize(),
    }
])

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(out_g, OUT_G)
write_parquet(out_d, OUT_D)
write_parquet(out_dist, OUT_DIST)

print(f"[OK] Wrote global : {OUT_G}")
print(out_g.to_string(index=False))
print(f"\n[OK] Wrote destino: {OUT_D} rows={len(out_d)}")
print(out_d.to_string(index=False))

```

```

print(f"\n[OK] Wrote dist    : {OUT_DIST}")
print(out_dist.to_string(index=False))

if __name__ == "__main__":
    main()

-----
[35/164] FILE: \src\eval\build_kpis_desp_poscosecha_ml2.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"

IN_FINAL = DATA / "gold" / "pred_poscosecha_ml2_desp_grado_dia_bloque_destino_final.parquet"
IN_REAL = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

OUT_GLOBAL = EVAL / "ml2_desp_poscosecha_eval_global.parquet"
OUT_BY_DEST = EVAL / "ml2_desp_poscosecha_eval_by_destino.parquet"
OUT_DIST = EVAL / "ml2_desp_poscosecha_eval_ratio_dist.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def wmae_log_ratio(log_ratio: pd.Series, w: pd.Series) -> float:
    x = pd.to_numeric(log_ratio, errors="coerce")
    ww = pd.to_numeric(w, errors="coerce").fillna(0.0)
    m = x.notna() & (ww > 0)
    if not bool(m.any()):
        return float("nan")
    denom = float(ww[m].sum())
    if denom <= 0:
        return float(np.nanmean(np.abs(x[m].values)))
    return float((np.abs(x[m].values) * ww[m].values).sum() / denom)

def main() -> None:
    df = read_parquet(IN_FINAL).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha_post_pred_used", "destino", "factor_desp_ml1", "factor_desp_final"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"Final sin columnas: {sorted(miss)}")

    df["fecha_post_pred_used"] = _to_date(df["fecha_post_pred_used"])
    df["destino"] = _canon_str(df["destino"])
    df["factor_desp_ml1"] = pd.to_numeric(df["factor_desp_ml1"], errors="coerce")
    df["factor_desp_final"] = pd.to_numeric(df["factor_desp_final"], errors="coerce")

```



```

# peso
# peso (tallos). Si no existe tallos/tallos_w en el universo, usar peso=1.0 por fila
if "tallos_w" in df.columns:
    w = pd.to_numeric(df["tallos_w"], errors="coerce")
elif "tallos" in df.columns:
    w = pd.to_numeric(df["tallos"], errors="coerce")
else:
    w = pd.Series(1.0, index=df.index)

df["w"] = w.fillna(0.0)

# Real
real = read_parquet(IN_REAL).copy()
real.columns = [str(c).strip() for c in real.columns]
need_r = {"fecha_post", "destino", "factor_desp"}
miss_r = need_r - set(real.columns)
if miss_r:
    raise ValueError(f"Real sin columnas: {sorted(miss_r)}")

real["fecha_post"] = _to_date(real["fecha_post"])
real["destino"] = _canon_str(real["destino"])
real["factor_desp_real"] = pd.to_numeric(real["factor_desp"], errors="coerce")

real2 = (
    real.groupby(["fecha_post", "destino"], dropna=False, as_index=False)
    .agg(factor_desp_real=("factor_desp_real", "median"))
)

x = df.merge(
    real2,
    left_on=["fecha_post_pred_used", "destino"],
    right_on=["fecha_post", "destino"],
    how="left",
)

eps = 1e-12
x["ratio_ml1"] = (x["factor_desp_real"] + eps) / (x["factor_desp_ml1"] + eps)
x["ratio_ml2"] = (x["factor_desp_real"] + eps) / (x["factor_desp_final"] + eps)
x["log_ratio_ml1"] = np.log(x["ratio_ml1"])
x["log_ratio_ml2"] = np.log(x["ratio_ml2"])

m = x["factor_desp_real"].notna() & x["factor_desp_ml1"].notna() & x["factor_desp_final"].notna()
d = x.loc[m].copy()

out_g = pd.DataFrame(
    [
        {
            "n_rows": int(len(d)),
            "n_dates": int(d["fecha_post_pred_used"].nunique()),
            "mae_log_ml1_w": wmae_log_ratio(d["log_ratio_ml1"], d["w"]),
            "mae_log_ml2_w": wmae_log_ratio(d["log_ratio_ml2"], d["w"]),
            "median_ratio_ml1": float(np.nanmedian(d["ratio_ml1"].values)),
            "median_ratio_ml2": float(np.nanmedian(d["ratio_ml2"].values)),
            "p05_ratio_ml2": float(np.nanpercentile(d["ratio_ml2"].values, 5)),
            "p95_ratio_ml2": float(np.nanpercentile(d["ratio_ml2"].values, 95)),
            "improvement_abs_mae_log_w": wmae_log_ratio(d["log_ratio_ml1"], d["w"]) - wmae_log_ratio(d["log_ratio_ml2"], d["w"]),
            "created_at": pd.Timestamp(datetime.now()).normalize(),
        }
    ]
)

rows = []
for dest, g in d.groupby("destino"):
    rows.append(
        {
            "destino": str(dest),
            "n_rows": int(len(g)),
            "mae_log_ml1_w": wmae_log_ratio(g["log_ratio_ml1"], g["w"]),
            "mae_log_ml2_w": wmae_log_ratio(g["log_ratio_ml2"], g["w"]),
            "median_ratio_ml2": float(np.nanmedian(g["ratio_ml2"].values)),

```

```

        "improvement_abs_mae_log_w": wmae_log_ratio(g["log_ratio_ml1"], g["w"]) - wmae_log_ratio(g["log_ratio_ml2"], g["w"]),
    }
)
out_d = pd.DataFrame(rows).sort_values("improvement_abs_mae_log_w", ascending=False)

dist = pd.DataFrame(
    [
        {
            "n": int(len(d)),
            "ratio_min": float(np.nanmin(d["ratio_ml2"].values)),
            "ratio_p05": float(np.nanpercentile(d["ratio_ml2"].values, 5)),
            "ratio_p25": float(np.nanpercentile(d["ratio_ml2"].values, 25)),
            "ratio_median": float(np.nanmedian(d["ratio_ml2"].values)),
            "ratio_p75": float(np.nanpercentile(d["ratio_ml2"].values, 75)),
            "ratio_p95": float(np.nanpercentile(d["ratio_ml2"].values, 95)),
            "ratio_max": float(np.nanmax(d["ratio_ml2"].values)),
            "created_at": pd.Timestamp(datetime.now()).normalize(),
        }
    ]
)

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(out_g, OUT_GLOBAL)
write_parquet(out_d, OUT_BY_DEST)
write_parquet(dist, OUT_DIST)

print(f"[OK] Wrote global : {OUT_GLOBAL}")
print(out_g.to_string(index=False))
print(f"\n[OK] Wrote destino: {OUT_BY_DEST} rows={len(out_d)}")
print(out_d.to_string(index=False))
print(f"\n[OK] Wrote dist : {OUT_DIST}")
print(dist.to_string(index=False))

if __name__ == "__main__":
    main()

-----
[36/164] FILE: \src\eval\build_kpis_dh_poscosecha_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime

import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

IN_FINAL = GOLD / "pred_poscosecha_ml2_dh_grado_dia_bloque_destino_final.parquet"
IN_REAL = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"

OUT_GLOBAL = EVAL / "ml2_dh_poscosecha_eval_global.parquet"
OUT_BY_DESTINO = EVAL / "ml2_dh_poscosecha_eval_by_destino.parquet"
OUT_BY_GRADO = EVAL / "ml2_dh_poscosecha_eval_by_grado.parquet"
OUT_DIST = EVAL / "ml2_dh_poscosecha_eval_delta_dist.parquet"

```

```

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _mae(x: pd.Series, w: pd.Series | None = None) -> float:
    x = pd.to_numeric(x, errors="coerce")
    if w is None:
        return float(np.nanmean(np.abs(x))) if len(x) else np.nan
    w = pd.to_numeric(w, errors="coerce").fillna(0.0).astype(float)
    m = x.notna() & np.isfinite(w)
    if not m.any():
        return np.nan
    ww = w[m].values
    ww = np.where(ww <= 0, 1.0, ww)
    return float(np.sum(np.abs(x[m].values) * ww) / np.sum(ww))

def _dist(s: pd.Series) -> dict:
    s = pd.to_numeric(s, errors="coerce").dropna()
    if s.empty:
        return {"n": 0}
    return {
        "n": int(len(s)),
        "min": float(s.min()),
        "p05": float(s.quantile(0.05)),
        "p25": float(s.quantile(0.25)),
        "median": float(s.median()),
        "p75": float(s.quantile(0.75)),
        "p95": float(s.quantile(0.95)),
        "max": float(s.max()),
    }

def main() -> None:
    fin = read_parquet(IN_FINAL).copy()
    real = read_parquet(IN_REAL).copy()
    fin.columns = [str(c).strip() for c in fin.columns]
    real.columns = [str(c).strip() for c in real.columns]

    fin["fecha"] = _to_date(fin["fecha"])
    fin["destino"] = _canon_str(fin["destino"])
    fin["grado"] = _canon_int(fin["grado"])

    # Real
    real["fecha_cosecha"] = _to_date(real["fecha_cosecha"])
    real["destino"] = _canon_str(real["destino"])
    real["grado"] = _canon_int(real["grado"])
    real["tallos"] = pd.to_numeric(real.get("tallos"), errors="coerce").fillna(0.0)
    real["dh_dias"] = _canon_int(real.get("dh_dias"))

    real_g = (
        real.groupby(["fecha_cosecha", "grado", "destino"], dropna=False, as_index=False)
        .agg(dh_real=("dh_dias", "median"), tallos_real=("tallos", "sum"))
    )

    df = fin.merge(
        real_g,
        left_on=["fecha", "grado", "destino"],
        right_on=["fecha_cosecha", "grado", "destino"],
        how="left",
    )

    # Errores
    df["dh_m11"] = _canon_int(df.get("dh_m11"))

```

```

df["dh_dias_final"] = _canon_int(df.get("dh_dias_final"))
df["dh_real"] = _canon_int(df.get("dh_real"))
df["tallos_real"] = pd.to_numeric(df.get("tallos_real"), errors="coerce").fillna(0.0)

m = df["dh_real"].notna() & df["dh_ml1"].notna() & df["dh_dias_final"].notna()
d = df.loc[m].copy()

d["err_ml1_days"] = (d["dh_real"].astype(float) - d["dh_ml1"].astype(float))
d["err_ml2_days"] = (d["dh_real"].astype(float) - d["dh_dias_final"].astype(float))
d["improvement_abs_days"] = d["err_ml1_days"].abs() - d["err_ml2_days"].abs()

out_g = pd.DataFrame([
    "n_rows": int(len(d)),
    "n_dates": int(d["fecha"].nunique()),
    "mae_ml1_days": _mae(d["err_ml1_days"], w=d["tallos_real"]),
    "mae_ml2_days": _mae(d["err_ml2_days"], w=d["tallos_real"]),
    "bias_ml1_days": float(np.nanmean(d["err_ml1_days"])),
    "bias_ml2_days": float(np.nanmean(d["err_ml2_days"])),
    "improvement_abs_mae_days": _mae(d["err_ml1_days"], w=d["tallos_real"]) - _mae(d["err_ml2_days"], w=d["tallos_real"]),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
])

rows_dest = []
for dest, g in d.groupby("destino"):
    rows_dest.append({
        "destino": str(dest),
        "n_rows": int(len(g)),
        "mae_ml1_days": _mae(g["err_ml1_days"], w=g["tallos_real"]),
        "mae_ml2_days": _mae(g["err_ml2_days"], w=g["tallos_real"]),
        "bias_ml1_days": float(np.nanmean(g["err_ml1_days"])),
        "bias_ml2_days": float(np.nanmean(g["err_ml2_days"])),
        "improvement_abs_mae_days": _mae(g["err_ml1_days"], w=g["tallos_real"]) - _mae(g["err_ml2_days"], w=g["tallos_real"]),
    })
out_d = pd.DataFrame(rows_dest).sort_values("improvement_abs_mae_days", ascending=False)

rows_gr = []
for gr, g in d.groupby("grado"):
    rows_gr.append({
        "grado": int(gr) if pd.notna(gr) else None,
        "n_rows": int(len(g)),
        "mae_ml1_days": _mae(g["err_ml1_days"], w=g["tallos_real"]),
        "mae_ml2_days": _mae(g["err_ml2_days"], w=g["tallos_real"]),
        "bias_ml1_days": float(np.nanmean(g["err_ml1_days"])),
        "bias_ml2_days": float(np.nanmean(g["err_ml2_days"])),
        "improvement_abs_mae_days": _mae(g["err_ml1_days"], w=g["tallos_real"]) - _mae(g["err_ml2_days"], w=g["tallos_real"]),
    })
out_gd = pd.DataFrame(rows_gr).sort_values("improvement_abs_mae_days", ascending=False)

dist = _dist(pd.to_numeric(fin.get("err_dh_days_pred_ml2"), errors="coerce"))
out_dist = pd.DataFrame([
    **dist,
    "created_at": pd.Timestamp(datetime.now()).normalize(),
])

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(out_g, OUT_GLOBAL)
write_parquet(out_d, OUT_BY_DESTINO)
write_parquet(out_gd, OUT_BY_GRADO)
write_parquet(out_dist, OUT_DIST)

print(f"[OK] Wrote global : {OUT_GLOBAL}")
print(out_g.to_string(index=False))
print(f"\n[OK] Wrote destino: {OUT_BY_DESTINO} rows={len(out_d)}")
print(out_d.to_string(index=False))
print(f"\n[OK] Wrote grado : {OUT_BY_GRADO} rows={len(out_gd)}")
print(out_gd.head(10).to_string(index=False))
print(f"\n[OK] Wrote dist : {OUT_DIST}")
print(out_dist.to_string(index=False))

```

```

if __name__ == "__main__":
    main()

-----
[37/164] FILE: \src\eval\build_kpis_harvest_horizon_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import argparse
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"

IN_FINAL_PROD = GOLD_DIR / "pred_harvest_horizon_final_ml2.parquet"
IN_FINAL_BT = EVAL_DIR / "backtest_pred_harvest_horizon_final_ml2.parquet"

OUT_PROD = EVAL_DIR / "ml2_harvest_horizon_eval_prod.parquet"
OUT_BT = EVAL_DIR / "ml2_harvest_horizon_eval_backtest.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else float("nan")

def _bias(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(x)) if len(x) else float("nan")

def _parse_args() -> argparse.Namespace:
    p = argparse.ArgumentParser()
    p.add_argument("--mode", choices=["prod", "backtest"], default="backtest")
    return p.parse_args()

def main() -> None:
    args = _parse_args()

    ciclo = read_parquet(IN_CICLO).copy()
    ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])
    ciclo["fecha_fin_cosecha"] = _to_date(ciclo["fecha_fin_cosecha"])

    if args.mode == "prod":
        pred = read_parquet(IN_FINAL_PROD).copy()
        out_path = OUT_PROD
    else:
        pred = read_parquet(IN_FINAL_BT).copy()
        out_path = OUT_BT

```

```

# Canon dates
pred["harvest_end_pred"] = _to_date(pred["harvest_end_pred"])
pred["harvest_end_final"] = _to_date(pred["harvest_end_final"])

pred["harvest_start_pred"] = _to_date(pred["harvest_start_pred"])
pred["harvest_start_final"] = _to_date(pred["harvest_start_final"])

# Merge
df = ciclo.merge(pred, on="ciclo_id", how="inner")

# Real duration
df["n_harvest_days_real"] = (df["fecha_fin_cosecha"] - df["fecha_inicio_cosecha"]).dt.days + 1

# Filter to valid rows
df = df.loc[
    df["n_harvest_days_real"].notna()
    & df["n_harvest_days_pred"].notna()
    & df["n_harvest_days_final"].notna()
    & df["harvest_end_pred"].notna()
    & df["harvest_end_final"].notna(),
    :
].copy()

df["n_harvest_days_real"] = pd.to_numeric(df["n_harvest_days_real"], errors="coerce")
df["n_harvest_days_pred"] = pd.to_numeric(df["n_harvest_days_pred"], errors="coerce")
df["n_harvest_days_final"] = pd.to_numeric(df["n_harvest_days_final"], errors="coerce")

# Errors on duration
df["err_n_days_ml1"] = df["n_harvest_days_real"] - df["n_harvest_days_pred"]
df["err_n_days_ml2"] = df["n_harvest_days_real"] - df["n_harvest_days_final"]

# Errors on end-date (in days)
df["err_end_ml1_days"] = (df["fecha_fin_cosecha"] - df["harvest_end_pred"]).dt.days
df["err_end_ml2_days"] = (df["fecha_fin_cosecha"] - df["harvest_end_final"]).dt.days

# KPIs
mae_ml1_n = _mae(df["err_n_days_ml1"])
mae_ml2_n = _mae(df["err_n_days_ml2"])

mae_ml1_end = _mae(df["err_end_ml1_days"])
mae_ml2_end = _mae(df["err_end_ml2_days"])

out = pd.DataFrame([
    {
        "mode": args.mode,
        "n": int(len(df)),
        "mae_n_days_ml1": mae_ml1_n,
        "mae_n_days_ml2": mae_ml2_n,
        "bias_n_days_ml1": _bias(df["err_n_days_ml1"]),
        "bias_n_days_ml2": _bias(df["err_n_days_ml2"]),
        "improvement_abs_n_days": (mae_ml1_n - mae_ml2_n) if (pd.notna(mae_ml1_n) and pd.notna(mae_ml2_n)) else np.nan,
        "mae_end_days_ml1": mae_ml1_end,
        "mae_end_days_ml2": mae_ml2_end,
        "bias_end_days_ml1": _bias(df["err_end_ml1_days"]),
        "bias_end_days_ml2": _bias(df["err_end_ml2_days"]),
        "improvement_abs_end_days": (mae_ml1_end - mae_ml2_end) if (pd.notna(mae_ml1_end) and pd.notna(mae_ml2_end)) else np.nan,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }
])

EVAL_DIR.mkdir(parents=True, exist_ok=True)
write_parquet(out, out_path)

print(f"[OK] Wrote eval: {out_path}")
print(out.to_string(index=False))

if __name__ == "__main__":
    main()

```

[38/164] FILE: \src\eval\build_kpis_harvest_start_ml2.py

```
-----  
from __future__ import annotations
```

```
from pathlib import Path  
from datetime import datetime  
import numpy as np  
import pandas as pd
```

```
from src.common.io import read_parquet, write_parquet
```

```
def _project_root() -> Path:  
    # .../src/eval/file.py -> repo_root = parents[2]  
    return Path(__file__).resolve().parents[2]
```

```
ROOT = _project_root()  
DATA_DIR = ROOT / "data"  
SILVER_DIR = DATA_DIR / "silver"  
GOLD_DIR = DATA_DIR / "gold"  
EVAL_DIR = DATA_DIR / "eval" / "ml2"
```

```
IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"  
IN_FINAL = EVAL_DIR / "backtest_pred_harvest_start_final_ml2.parquet"
```

```
OUT_EVAL = EVAL_DIR / "ml2_harvest_start_eval.parquet"
```

```
def _to_date(s: pd.Series) -> pd.Series:  
    return pd.to_datetime(s, errors="coerce").dt.normalize()
```

```
def main() -> None:  
    ciclo = read_parquet(IN_CICLO).copy()  
    pred = read_parquet(IN_FINAL).copy()  
  
    ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])  
    pred["harvest_start_pred"] = _to_date(pred["harvest_start_pred"])  
    pred["harvest_start_final"] = _to_date(pred["harvest_start_final"])  
  
    df = ciclo.merge(pred, on="ciclo_id", how="inner")  
    df = df.loc[  
        df["fecha_inicio_cosecha"].notna()  
        & df["harvest_start_pred"].notna()  
        & df["harvest_start_final"].notna(),  
        :  
    ].copy()  
  
    df["err_ml1_days"] = (df["fecha_inicio_cosecha"] - df["harvest_start_pred"]).dt.days.astype(int)  
    df["err_ml2_days"] = (df["fecha_inicio_cosecha"] - df["harvest_start_final"]).dt.days.astype(int)
```

```
def _mae(x: pd.Series) -> float:  
    return float(np.mean(np.abs(x))) if len(x) else float("nan")
```

```
def _bias(x: pd.Series) -> float:  
    return float(np.mean(x)) if len(x) else float("nan")
```

```
mae_ml1 = _mae(df["err_ml1_days"])  
mae_ml2 = _mae(df["err_ml2_days"])
```

```
out = pd.DataFrame([  
    "n": int(len(df)),  
    "mae_ml1_days": mae_ml1,  
    "mae_ml2_days": mae_ml2,  
    "bias_ml1_days": _bias(df["err_ml1_days"]),  
    "bias_ml2_days": _bias(df["err_ml2_days"]),  
    "improvement_abs_days": (mae_ml1 - mae_ml2) if (pd.notna(mae_ml1) and pd.notna(mae_ml2)) else float("nan"),  
    "created_at": pd.Timestamp(datetime.now()).normalize(),  
])
```

```

EVAL_DIR.mkdir(parents=True, exist_ok=True)
write_parquet(out, OUT_EVAL)
print(f"[OK] Wrote eval: {OUT_EVAL}")

if __name__ == "__main__":
    main()

-----
[39/164] FILE: \src\eval\build_kpis_hidr_poscosecha_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

IN_FINAL = GOLD / "pred_poscosecha_ml2_hidr_grado_dia_bloque_destino_final.parquet"
IN_REAL = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"

OUT_GLOBAL = EVAL / "ml2_hidr_poscosecha_eval_global.parquet"
OUT_BY_DESTINO = EVAL / "ml2_hidr_poscosecha_eval_by_destino.parquet"
OUT_BY_GRADO = EVAL / "ml2_hidr_poscosecha_eval_by_grado.parquet"
OUT_RATIO_DIST = EVAL / "ml2_hidr_poscosecha_eval_ratio_dist.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _factor_from_hidr_pct(hidr_pct: pd.Series) -> pd.Series:
    x = pd.to_numeric(hidr_pct, errors="coerce")
    return np.where(x.isna(), np.nan, np.where(x > 3.5, 1.0 + x / 100.0, x)).astype(float)

def _mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else np.nan

def main() -> None:
    df = read_parquet(IN_FINAL).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"grado", "destino", "factor_hidr_ml1", "factor_hidr_final"}
    miss = need - set(df.columns)
    if miss:
        raise KeyError(f"Final sin columnas: {sorted(miss)}")

```



```

# fecha_post_pred usada
fpp = None
for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml2", "fecha_post_pred_ml1", "
fecha_post_pred"]:
```

```

    if c in df.columns:
        fpp = c
        break
if fpp is None:
    raise KeyError("No encuentro fecha_post_pred en final.")

df[fpp] = _to_date(df[fpp])
df["destino"] = _canon_str(df["destino"])
df["grado"] = _canon_int(df["grado"])
df["factor_hidr_ml1"] = pd.to_numeric(df["factor_hidr_ml1"], errors="coerce")
df["factor_hidr_final"] = pd.to_numeric(df["factor_hidr_final"], errors="coerce")

# real
real = read_parquet(IN_REAL).copy()
real.columns = [str(c).strip() for c in real.columns]
if "hidr_pct" in real.columns:
    real["factor_hidr_real"] = _factor_from_hidr_pct(real["hidr_pct"])
else:
    pb = pd.to_numeric(real.get("peso_base_g"), errors="coerce")
    pp = pd.to_numeric(real.get("peso_post_g"), errors="coerce")
    real["factor_hidr_real"] = np.where(pb > 0, pp / pb, np.nan)

real["fecha_post"] = _to_date(real["fecha_post"])
real["destino"] = _canon_str(real["destino"])
real["grado"] = _canon_int(real["grado"])

# agregación real por llave
if "tallos" in real.columns:
    real["tallos"] = pd.to_numeric(real["tallos"], errors="coerce").fillna(0.0)
    g = real.groupby(["fecha_post", "grado", "destino"], dropna=False)
    real2 = g.apply(
        lambda x: pd.Series({
            "factor_hidr_real": float(np.nansum(x["factor_hidr_real"] * x["tallos"]) / np.nansum(x["tallos"]
            "))) if np.nansum(x["tallos"]) > 0 else float(np.nanmedian(x["factor_hidr_real"])),
            "tallos_real_sum": float(np.nansum(x["tallos"])),
        })
    ).reset_index()
else:
    real2 = (
        real.groupby(["fecha_post", "grado", "destino"], dropna=False, as_index=False)
        .agg(factor_hidr_real=("factor_hidr_real", "median"))
    )
    real2["tallos_real_sum"] = np.nan

mrg = df.merge(
    real2,
    left_on=[fpp, "grado", "destino"],
    right_on=["fecha_post", "grado", "destino"],
    how="left",
).drop(columns=["fecha_post"], errors="ignore")

m = mrg["factor_hidr_real"].notna() & mrg["factor_hidr_ml1"].notna() & mrg["factor_hidr_final"].notna()
d = mrg.loc[m].copy()
if d.empty:
    raise ValueError("No hay match con real para evaluar hidratación.")

eps = 1e-9
d["ratio_ml1"] = d["factor_hidr_real"] / d["factor_hidr_ml1"].clip(lower=eps)
d["ratio_ml2"] = d["factor_hidr_real"] / d["factor_hidr_final"].clip(lower=eps)

d["log_ratio_ml1"] = np.log(d["ratio_ml1"].clip(lower=eps))
d["log_ratio_ml2"] = np.log(d["ratio_ml2"].clip(lower=eps))

# peso por tallos
w = pd.to_numeric(d.get("tallos_real_sum"), errors="coerce").fillna(1.0).clip(lower=0.0)
if (w.sum() <= 0) or (w.isna().all()):
    w = pd.Series(1.0, index=d.index)

```

```

def wmae(x):
    x = pd.to_numeric(x, errors="coerce")
    mm = x.notna()
    if not mm.any():
        return np.nan
    return float(np.sum(np.abs(x[mm]) * w[mm]) / np.sum(w[mm]))

out_g = pd.DataFrame([
    "n_rows": int(len(d)),
    "n_dates": int(d[fpp].nunique()),
    "mae_log_ml1_w": wmae(d["log_ratio_ml1"]),
    "mae_log_ml2_w": wmae(d["log_ratio_ml2"]),
    "median_ratio_ml1": float(np.nanmedian(d["ratio_ml1"])),
    "median_ratio_ml2": float(np.nanmedian(d["ratio_ml2"])),
    "p05_ratio_ml2": float(np.nanpercentile(d["ratio_ml2"], 5)),
    "p95_ratio_ml2": float(np.nanpercentile(d["ratio_ml2"], 95)),
    "improvement_abs_mae_log_w": wmae(d["log_ratio_ml1"]) - wmae(d["log_ratio_ml2"]),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
])

# by destino
rows = []
for dest, g in d.groupby("destino"):
    ww = pd.to_numeric(g.get("tallos_real_sum"), errors="coerce").fillna(1.0).clip(lower=0.0)
    if ww.sum() <= 0:
        ww = pd.Series(1.0, index=g.index)

    def _wmae(v):
        v = pd.to_numeric(v, errors="coerce")
        mm = v.notna()
        return float(np.sum(np.abs(v[mm]) * ww[mm]) / np.sum(ww[mm])) if mm.any() else np.nan

    rows.append({
        "destino": str(dest),
        "n_rows": int(len(g)),
        "mae_log_ml1_w": _wmae(g["log_ratio_ml1"]),
        "mae_log_ml2_w": _wmae(g["log_ratio_ml2"]),
        "median_ratio_ml2": float(np.nanmedian(g["ratio_ml2"])),
        "improvement_abs_mae_log_w": _wmae(g["log_ratio_ml1"]) - _wmae(g["log_ratio_ml2"]),
    })
out_bd = pd.DataFrame(rows).sort_values("improvement_abs_mae_log_w", ascending=False)

# by grado
rows = []
for gr, g in d.groupby("grado"):
    ww = pd.to_numeric(g.get("tallos_real_sum"), errors="coerce").fillna(1.0).clip(lower=0.0)
    if ww.sum() <= 0:
        ww = pd.Series(1.0, index=g.index)

    def _wmae(v):
        v = pd.to_numeric(v, errors="coerce")
        mm = v.notna()
        return float(np.sum(np.abs(v[mm]) * ww[mm]) / np.sum(ww[mm])) if mm.any() else np.nan

    rows.append({
        "grado": int(gr) if pd.notna(gr) else None,
        "n_rows": int(len(g)),
        "mae_log_ml1_w": _wmae(g["log_ratio_ml1"]),
        "mae_log_ml2_w": _wmae(g["log_ratio_ml2"]),
        "median_ratio_ml2": float(np.nanmedian(g["ratio_ml2"])),
        "improvement_abs_mae_log_w": _wmae(g["log_ratio_ml1"]) - _wmae(g["log_ratio_ml2"]),
    })
out_bg = pd.DataFrame(rows).sort_values("improvement_abs_mae_log_w", ascending=False)

# ratio dist ml2
out_dist = pd.DataFrame([
    "n": int(len(d)),
    "ratio_min": float(np.nanmin(d["ratio_ml2"])),
    "ratio_p05": float(np.nanpercentile(d["ratio_ml2"], 5)),
    "ratio_p25": float(np.nanpercentile(d["ratio_ml2"], 25)),
    "ratio_median": float(np.nanmedian(d["ratio_ml2"])),
    "ratio_p75": float(np.nanpercentile(d["ratio_ml2"], 75)),
])

```

```

    "ratio_p95": float(np.nanpercentile(d["ratio_ml2"], 95)),
    "ratio_max": float(np.nanmax(d["ratio_ml2"])),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
})]

```

```

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(out_g, OUT_GLOBAL)
write_parquet(out_bd, OUT_BY_DESTINO)
write_parquet(out_bg, OUT_BY_GRADO)
write_parquet(out_dist, OUT_RATIO_DIST)

print(f"[OK] Wrote global : {OUT_GLOBAL}")
print(out_g.to_string(index=False))
print(f"\n[OK] Wrote destino: {OUT_BY_DESTINO} rows={len(out_bd)}")
print(out_bd.to_string(index=False))
print(f"\n[OK] Wrote grado : {OUT_BY_GRADO} rows={len(out_bg)}")
print(out_bg.head(13).to_string(index=False))
print(f"\n[OK] Wrote dist : {OUT_RATIO_DIST}")
print(out_dist.to_string(index=False))

```

```

if __name__ == "__main__":
    main()

```

```

-----
[40/164] FILE: \src\eval\build_kpis_peso_tallo_ml2.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

```

```

from src.common.io import read_parquet, write_parquet

```

```

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

```

```

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"

```

```

IN_FACT = EVAL / "backtest_factor_ml2_peso_tallo_grado_dia.parquet"
IN_TALLOS = SILVER / "fact_cosecha_real_grado_dia.parquet"

```

```

OUT_GLOBAL = EVAL / "ml2_peso_tallo_eval_global.parquet"
OUT_ESTADO = EVAL / "ml2_peso_tallo_eval_by_estado.parquet"
OUT_RATIO = EVAL / "ml2_peso_tallo_eval_ratio_dist.parquet"

```

```

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

```

```

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

```

```

def _mae(x: pd.Series) -> float:
    return float(np.nanmean(np.abs(x))) if len(x) else np.nan

```

```

def _wape_weighted(err_abs: pd.Series, w: pd.Series) -> float:
    w = pd.to_numeric(w, errors="coerce").fillna(0.0)
    err_abs = pd.to_numeric(err_abs, errors="coerce").fillna(0.0)
    denom = float(np.sum(w))
    if denom <= 0:

```

```

        return np.nan
    return float(np.sum(err_abs * w) / denom)

def main() -> None:
    df = read_parquet(IN_FACT).copy()

    # Tallos reales (peso por tallo debe ponderarse por tallos)
    tl = read_parquet(IN_TALLOS).copy()
    tl["fecha"] = _to_date(tl["fecha"])
    tl["grado"] = _canon_str(tl["grado"]) if "grado" in tl.columns else tl["grado"]
    if "bloque_base" not in tl.columns:
        if "bloque_padre" in tl.columns:
            tl["bloque_base"] = _canon_str(tl["bloque_padre"])
        elif "bloque" in tl.columns:
            tl["bloque_base"] = _canon_str(tl["bloque"])
        else:
            raise KeyError("fact_cosecha_real_grado_dia no tiene bloque_base/bloque_padre/bloque")
    else:
        tl["bloque_base"] = _canon_str(tl["bloque_base"])

    tl["tallos_real"] = pd.to_numeric(tl["tallos_real"], errors="coerce").fillna(0.0)
    tl = tl.groupby(["fecha", "bloque_base", "grado"], as_index=False).agg(tallos_real=("tallos_real", "sum"))

    # Canon fact
    df["fecha"] = _to_date(df["fecha"])
    df["bloque_base"] = _canon_str(df["bloque_base"])
    df["grado"] = _canon_str(df["grado"])
    if "estado" in df.columns:
        df["estado"] = _canon_str(df["estado"])

    for c in ["peso_tallo_ml1_g", "peso_tallo_real_g", "peso_tallo_final_g", "pred_ratio_peso"]:
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce")

    df = df.merge(tl, on=["fecha", "bloque_base", "grado"], how="left")
    df["tallos_real"] = pd.to_numeric(df["tallos_real"], errors="coerce").fillna(0.0)

    # Keep rows where we have real peso_tallo and some weight
    df = df[df["peso_tallo_real_g"].notna()].copy()

    # Errors in g/tallo
    df["err_ml1_g"] = df["peso_tallo_real_g"] - df["peso_tallo_ml1_g"]
    df["err_ml2_g"] = df["peso_tallo_real_g"] - df["peso_tallo_final_g"]

    # Weighted absolute error (by tallos)
    df["abs_err_ml1_g"] = df["err_ml1_g"].abs()
    df["abs_err_ml2_g"] = df["err_ml2_g"].abs()

    # Impact kg (approx)
    df["abs_err_ml1_kg_equiv"] = (df["abs_err_ml1_g"] * df["tallos_real"]) / 1000.0
    df["abs_err_ml2_kg_equiv"] = (df["abs_err_ml2_g"] * df["tallos_real"]) / 1000.0

    out_g = pd.DataFrame([
        {
            "n_rows": int(len(df)),
            "n_cycles": int(df["ciclo_id"].nunique()) if "ciclo_id" in df.columns else np.nan,
            "mae_ml1_g": _mae(df["err_ml1_g"]),
            "mae_ml2_g": _mae(df["err_ml2_g"]),
            "wape_wt_ml1_g": _wape_weighted(df["abs_err_ml1_g"], df["tallos_real"]),
            "wape_wt_ml2_g": _wape_weighted(df["abs_err_ml2_g"], df["tallos_real"]),
            "kg_abs_err_ml1": float(df["abs_err_ml1_kg_equiv"].sum()),
            "kg_abs_err_ml2": float(df["abs_err_ml2_kg_equiv"].sum()),
            "improvement_abs_mae_g": _mae(df["err_ml1_g"]) - _mae(df["err_ml2_g"]),
            "improvement_abs_wape_wt_g": _wape_weighted(df["abs_err_ml1_g"], df["tallos_real"])
                - _wape_weighted(df["abs_err_ml2_g"], df["tallos_real"]),
            "improvement_abs_kg": float(df["abs_err_ml1_kg_equiv"].sum() - df["abs_err_ml2_kg_equiv"].sum()),
            "created_at": pd.Timestamp(datetime.now()).normalize(),
        }
    ])

    # Ratio dist sanity
    r = pd.to_numeric(df["pred_ratio_peso"], errors="coerce")
    out_r = pd.DataFrame([

```

```

        "n": int(r.notna().sum()),
        "ratio_min": float(np.nanmin(r)),
        "ratio_p05": float(np.nanpercentile(r, 5)),
        "ratio_p25": float(np.nanpercentile(r, 25)),
        "ratio_median": float(np.nanmedian(r)),
        "ratio_p75": float(np.nanpercentile(r, 75)),
        "ratio_p95": float(np.nanpercentile(r, 95)),
        "ratio_max": float(np.nanmax(r)),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    })

# By estado if exists
rows = []
if "estado" in df.columns and df["estado"].notna().any():
    for est, g in df.groupby("estado"):
        rows.append({
            "estado": str(est),
            "n_rows": int(len(g)),
            "mae_ml1_g": _mae(g["err_ml1_g"]),
            "mae_ml2_g": _mae(g["err_ml2_g"]),
            "wape_wt_ml1_g": _wape_weighted(g["abs_err_ml1_g"], g["tallos_real"]),
            "wape_wt_ml2_g": _wape_weighted(g["abs_err_ml2_g"], g["tallos_real"]),
            "kg_abs_err_ml1": float(g["abs_err_ml1_kg_equiv"].sum()),
            "kg_abs_err_ml2": float(g["abs_err_ml2_kg_equiv"].sum()),
            "improvement_abs_kg": float(g["abs_err_ml1_kg_equiv"].sum() - g["abs_err_ml2_kg_equiv"].sum())
        })

out_e = pd.DataFrame(rows)

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(out_g, OUT_GLOBAL)
write_parquet(out_r, OUT_RATIO)
write_parquet(out_e, OUT_ESTADO)

print(f"[OK] Wrote global: {OUT_GLOBAL}")
print(out_g.to_string(index=False))
print(f"\n[OK] Wrote ratio : {OUT_RATIO}")
print(out_r.to_string(index=False))
print(f"\n[OK] Wrote estado: {OUT_ESTADO} rows={len(out_e)}")
if not out_e.empty:
    print(out_e.sort_values('improvement_abs_kg', ascending=False).to_string(index=False))

if __name__ == "__main__":
    main()

-----
[41/164] FILE: \src\eval\build_kpis_poscosecha_ml2_baseline.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
SILVER = DATA / "silver"
EVAL = DATA / "eval" / "ml2"

# =====
# INPUTS

```

```

# =====
IN_PRED_FULL = GOLD / "pred_poscosecha_ml2_full_grado_dia_bloque_destino.parquet"

# Real hidr + DH (por fecha_cosecha/fecha_post, grado, destino)
IN_REAL_HIDR_DH = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"

# Real merma/ajuste (por fecha_post, destino)
IN_REAL_MERMA_AJ = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

# =====
# OUTPUTS
# =====
OUT_GLOBAL = EVAL / "ml2_poscosecha_baseline_eval_global.parquet"
OUT_BY_DESTINO = EVAL / "ml2_poscosecha_baseline_eval_by_destino.parquet"
OUT_BY_GRADO = EVAL / "ml2_poscosecha_baseline_eval_by_grado.parquet"
OUT_COVERAGE = EVAL / "ml2_poscosecha_baseline_eval_coverage.parquet"

# =====
# HELPERS
# =====
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _as_of_date_default() -> pd.Timestamp:
    # Regla operativa: "hoy" no se usa porque no está cerrado ? usar hoy-1
    return pd.Timestamp.now().normalize() - pd.Timedelta(days=1)

def _safe_div(a: pd.Series, b: pd.Series) -> pd.Series:
    a = pd.to_numeric(a, errors="coerce")
    b = pd.to_numeric(b, errors="coerce")
    return a / b.replace(0, np.nan)

def mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else np.nan

def wape(abs_err: pd.Series, y_true: pd.Series) -> float:
    abs_err = pd.to_numeric(abs_err, errors="coerce").fillna(0.0)
    y_true = pd.to_numeric(y_true, errors="coerce").fillna(0.0)
    denom = float(np.sum(np.abs(y_true)))
    if denom <= 0:
        return np.nan
    return float(np.sum(abs_err) / denom)

def _require_cols(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise KeyError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def main(as_of_date: str | None = None) -> None:
    AOD = _as_of_date_default() if as_of_date is None else pd.to_datetime(as_of_date).normalize()

    pred = read_parquet(IN_PRED_FULL).copy()
    pred.columns = [str(c).strip() for c in pred.columns]

    real_hd = read_parquet(IN_REAL_HIDR_DH).copy()
    real_hd.columns = [str(c).strip() for c in real_hd.columns]

```

```

real_ma = read_parquet(IN_REAL_MERMA_AJ).copy()
real_ma.columns = [str(c).strip() for c in real_ma.columns]

# =====
# Canon pred
# =====
_require_cols(pred, ["fecha", "grado", "destino"], "pred_poscosecha_ml2_full")

pred["fecha"] = _to_date(pred["fecha"])
pred["grado"] = _canon_int(pred["grado"])
pred["destino"] = _canon_str(pred["destino"])

# columnas ML1 esperadas (con fallback)
# DH
dh_col = None
for c in ["dh_dias_ml1", "dh_dias_pred_ml1", "dh_dias"]:
    if c in pred.columns:
        dh_col = c
        break

# fecha_post predicha (si existe, mejor)
fecha_post_pred_col = None
for c in ["fecha_post_pred_ml1", "fecha_post_pred", "fecha_post_ml1"]:
    if c in pred.columns:
        fecha_post_pred_col = c
        break

if fecha_post_pred_col:
    pred[fecha_post_pred_col] = _to_date(pred[fecha_post_pred_col])
elif dh_col:
    pred["fecha_post_pred_ml1"] = pred["fecha"] + pd.to_timedelta(
        pd.to_numeric(pred[dh_col], errors="coerce").fillna(0).astype(int), unit="D"
    )
    fecha_post_pred_col = "fecha_post_pred_ml1"
else:
    raise KeyError("No encuentro dh ni fecha_post_pred en pred. Espero dh_dias_ml1 o fecha_post_pred_ml1.")
)

# Hidr
hidr_col = None
for c in ["factor_hidr_ml1", "factor_hidr"]:
    if c in pred.columns:
        hidr_col = c
        break

# Desp
desp_col = None
for c in ["factor_desp_ml1", "factor_desp"]:
    if c in pred.columns:
        desp_col = c
        break

# Ajuste
aj_col = None
for c in ["ajuste_ml1", "factor_ajuste_ml1", "factor_ajuste"]:
    if c in pred.columns:
        aj_col = c
        break

# filtro as_of
pred = pred[pred["fecha"] <= AOD].copy()
pred = pred[pred[fecha_post_pred_col] <= AOD].copy()

# =====
# Canon real hidr/dh
# =====
_require_cols(
    real_hd,
    ["fecha_cosecha", "fecha_post", "dh_dias", "grado", "destino", "tallos", "peso_base_g", "peso_post_g"]
,
    "fact_hidratacion_real_post_grado_destino",

```

```

)
real_hd["fecha_cosecha"] = _to_date(real_hd["fecha_cosecha"])
real_hd["fecha_post"] = _to_date(real_hd["fecha_post"])
real_hd["grado"] = _canon_int(real_hd["grado"])
real_hd["destino"] = _canon_str(real_hd["destino"])
real_hd["tallos"] = pd.to_numeric(real_hd["tallos"], errors="coerce").fillna(0.0)
real_hd["dh_dias"] = pd.to_numeric(real_hd["dh_dias"], errors="coerce")

real_hd["peso_base_g"] = pd.to_numeric(real_hd["peso_base_g"], errors="coerce")
real_hd["peso_post_g"] = pd.to_numeric(real_hd["peso_post_g"], errors="coerce")
real_hd["factor_hidr_real"] = _safe_div(real_hd["peso_post_g"], real_hd["peso_base_g"])

# filtro as_of (NO usar hoy)
real_hd = real_hd[(real_hd["fecha_cosecha"] <= AOD) & (real_hd["fecha_post"] <= AOD)].copy()

# colapsar real a grano único por (fecha_cosecha, grado, destino)
# (si hay múltiples destinos/grado por día, sum tallos y ponderar hidratación)
rg = (
    real_hd.groupby(["fecha_cosecha", "grado", "destino"], as_index=False)
    .agg(
        tallos=("tallos", "sum"),
        dh_dias=("dh_dias", "median"),
        peso_base_g=("peso_base_g", "median"),
        peso_post_g=("peso_post_g", "median"),
        factor_hidr_real=("factor_hidr_real", "median"),
        fecha_post_real=("fecha_post", "median"),
    )
)

# =====
# Canon real merma/ajuste
# =====
_require_cols(
    real_ma,
    ["fecha_post", "destino", "factor_desp", "factor_ajuste"],
    "dim_mermas_ajuste_fecha_post_destino",
)
real_ma["fecha_post"] = _to_date(real_ma["fecha_post"])
real_ma["destino"] = _canon_str(real_ma["destino"])
real_ma["factor_desp"] = pd.to_numeric(real_ma["factor_desp"], errors="coerce")
real_ma["factor_ajuste"] = pd.to_numeric(real_ma["factor_ajuste"], errors="coerce")
real_ma = real_ma[real_ma["fecha_post"] <= AOD].copy()

# =====
# JOIN: pred vs real (DH/HIDR) por fecha_cosecha (pred.fecha)
# =====
df = pred.merge(
    rg,
    left_on=["fecha", "grado", "destino"],
    right_on=["fecha_cosecha", "grado", "destino"],
    how="left",
)

# JOIN merma/ajuste real por fecha_post_real (prefer) y destino.
# Si no hay fecha_post_real, usar fecha_post_pred (evaluación ?contra calendario?)
df["fecha_post_key"] = df["fecha_post_real"]
df["fecha_post_key"] = df["fecha_post_key"].where(df["fecha_post_key"].notna(), df["fecha_post_pred_col"])
df = df.merge(
    real_ma.rename(columns={"fecha_post": "fecha_post_key"}),
    on=["fecha_post_key", "destino"],
    how="left",
    suffixes=("", "_realma"),
)

# =====
# KPIs por componente
# =====
# Pesos: usar tallos reales como peso
w = pd.to_numeric(df["tallos"], errors="coerce").fillna(0.0)

# --- DH ---
if dh_col:

```



```

    df["dh_pred"] = pd.to_numeric(df[dh_col], errors="coerce")
else:
    df["dh_pred"] = pd.to_numeric(_safe_div((df[fecha_post_pred_col] - df["fecha"]).dt.days, 1.0), errors=
"coerce")

df["dh_real"] = pd.to_numeric(df["dh_dias"], errors="coerce")
m_dh = df["dh_pred"].notna() & df["dh_real"].notna()
df.loc[m_dh, "err_dh_days"] = df.loc[m_dh, "dh_pred"] - df.loc[m_dh, "dh_real"]
df.loc[m_dh, "abs_err_dh_days"] = df.loc[m_dh, "err_dh_days"].abs()

# --- HIDR ---
if hidr_col:
    df["hidr_pred_factor"] = pd.to_numeric(df[hidr_col], errors="coerce")
else:
    df["hidr_pred_factor"] = np.nan
df["hidr_real_factor"] = pd.to_numeric(df["factor_hidr_real"], errors="coerce")

m_h = df["hidr_pred_factor"].notna() & df["hidr_real_factor"].notna()
df.loc[m_h, "ratio_hidr"] = _safe_div(df.loc[m_h, "hidr_real_factor"], df.loc[m_h, "hidr_pred_factor"])
df.loc[m_h, "log_ratio_hidr"] = np.log(df.loc[m_h, "ratio_hidr"].clip(lower=1e-6))

# --- DESP ---
if desp_col:
    df["desp_pred_factor"] = pd.to_numeric(df[desp_col], errors="coerce")
else:
    df["desp_pred_factor"] = np.nan
df["desp_real_factor"] = pd.to_numeric(df["factor_desp"], errors="coerce")

m_d = df["desp_pred_factor"].notna() & df["desp_real_factor"].notna()
df.loc[m_d, "ratio_desp"] = _safe_div(df.loc[m_d, "desp_real_factor"], df.loc[m_d, "desp_pred_factor"])
df.loc[m_d, "log_ratio_desp"] = np.log(df.loc[m_d, "ratio_desp"].clip(lower=1e-6))

# --- AJUSTE ---
if aj_col:
    df["aj_pred_factor"] = pd.to_numeric(df[aj_col], errors="coerce")
else:
    df["aj_pred_factor"] = np.nan
df["aj_real_factor"] = pd.to_numeric(df["factor_ajuste"], errors="coerce")

m_a = df["aj_pred_factor"].notna() & df["aj_real_factor"].notna()
df.loc[m_a, "ratio_ajuste"] = _safe_div(df.loc[m_a, "aj_real_factor"], df.loc[m_a, "aj_pred_factor"])
df.loc[m_a, "log_ratio_ajuste"] = np.log(df.loc[m_a, "ratio_ajuste"].clip(lower=1e-6))

# =====
# GLOBAL KPIs
# =====
def _kpi_block(sub: pd.DataFrame) -> dict:
    ww = pd.to_numeric(sub["tallos"], errors="coerce").fillna(0.0)

    out = {
        "n_rows_pred": int(len(sub)),
        "n_rows_with_real_hd": int(sub["tallos"].notna().sum()),
        "as_of_date": AOD,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
        "dh_col_used": str(dh_col),
        "fecha_post_pred_col_used": str(fecha_post_pred_col),
        "hidr_col_used": str(hidr_col),
        "desp_col_used": str(desp_col),
        "ajuste_col_used": str(aj_col),
    }

    # DH
    dsub = sub[sub["err_dh_days"].notna()].copy()
    out["n_dh"] = int(len(dsub))
    out["mae_dh_days"] = mae(dsub["err_dh_days"])
    out["bias_dh_days"] = float(np.nanmean(pd.to_numeric(dsub["err_dh_days"], errors="coerce")) if len(dsub)
ub) else np.nan
    out["wape_abs_dh_days_wt_tallos"] = wape(dsub["abs_err_dh_days"], ww.loc[dsub.index]) if len(dsub) els
e np.nan

    # Hidr
    hsub = sub[sub["log_ratio_hidr"].notna()].copy()

```

```

        out["n_hidr"] = int(len(hsub))
        out["mae_log_ratio_hidr"] = mae(hsub["log_ratio_hidr"])
        out["median_ratio_hidr"] = float(np.nanmedian(pd.to_numeric(hsub["ratio_hidr"], errors="coerce"))) if
len(hsub) else np.nan
        out["p05_ratio_hidr"] = float(np.nanpercentile(pd.to_numeric(hsub["ratio_hidr"], errors="coerce"), 5))
        if len(hsub) else np.nan
        out["p95_ratio_hidr"] = float(np.nanpercentile(pd.to_numeric(hsub["ratio_hidr"], errors="coerce"), 95)
    ) if len(hsub) else np.nan

    # Desp
    dpsub = sub[sub["log_ratio_desp"].notna()].copy()
    out["n_desp"] = int(len(dpsub))
    out["mae_log_ratio_desp"] = mae(dpsub["log_ratio_desp"])
    out["median_ratio_desp"] = float(np.nanmedian(pd.to_numeric(dpsub["ratio_desp"], errors="coerce"))) if
len(dpsub) else np.nan

    # Ajuste
    asub = sub[sub["log_ratio_ajuste"].notna()].copy()
    out["n_ajuste"] = int(len(asub))
    out["mae_log_ratio_ajuste"] = mae(asub["log_ratio_ajuste"])
    out["median_ratio_ajuste"] = float(np.nanmedian(pd.to_numeric(asub["ratio_ajuste"], errors="coerce")))
if len(asub) else np.nan

    return out

global_df = pd.DataFrame([_kpi_block(df)])

# =====
# BY DESTINO / BY GRADO
# =====
by_dest = []
for dest, g in df.groupby("destino", dropna=False):
    row = _kpi_block(g)
    row["destino"] = str(dest)
    by_dest.append(row)
by_dest_df = pd.DataFrame(by_dest).sort_values("n_rows_pred", ascending=False)

by_gr = []
for gr, g in df.groupby("grado", dropna=False):
    row = _kpi_block(g)
    row["grado"] = int(gr) if pd.notna(gr) else None
    by_gr.append(row)
by_gr_df = pd.DataFrame(by_gr).sort_values("n_rows_pred", ascending=False)

# =====
# COVERAGE / SANITY
# =====
cov = pd.DataFrame([{
    "as_of_date": AOD,
    "rows_pred": int(len(pred)),
    "rows_join_real_hd": int(df["tallos"].notna().sum()),
    "coverage_real_hd": float(df["tallos"].notna().mean()) if len(df) else np.nan,
    "rows_join_real_ma": int(df["factor_desp"].notna().sum()),
    "coverage_real_ma": float(df["factor_desp"].notna().mean()) if len(df) else np.nan,
    "created_at": pd.Timestamp(datetime.now()).normalize(),
}])

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(global_df, OUT_GLOBAL)
write_parquet(by_dest_df, OUT_BY_DESTINO)
write_parquet(by_gr_df, OUT_BY_GRADO)
write_parquet(cov, OUT_COVERAGE)

print(f"[OK] Wrote global : {OUT_GLOBAL}")
print(global_df.to_string(index=False))
print(f"\n[OK] Wrote destino: {OUT_BY_DESTINO} rows={len(by_dest_df)}")
print(by_dest_df.head(10).to_string(index=False))
print(f"\n[OK] Wrote grado : {OUT_BY_GRADO} rows={len(by_gr_df)}")
print(by_gr_df.head(10).to_string(index=False))
print(f"\n[OK] Wrote coverage: {OUT_COVERAGE}")
print(cov.to_string(index=False))

```

```

if __name__ == "__main__":
    main(as_of_date=None)

-----
[42/164] FILE: \src\eval\build_kpis_share_grado_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
EVAL = DATA / "eval" / "ml2"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

IN_FACTOR = EVAL / "backtest_factor_ml2_share_grado.parquet"
IN_FINAL = EVAL / "backtest_pred_tallos_grado_dia_ml2_final.parquet"
IN_REAL = SILVER / "fact_cosecha_real_grado_dia.parquet"
IN_PRED_ML1 = GOLD / "pred_tallos_grado_dia_ml1_full.parquet"

OUT_GLOBAL = EVAL / "ml2_share_grado_eval_global.parquet"
OUT_BY_GRADO = EVAL / "ml2_share_grado_eval_by_grado.parquet"
OUT_SANITY = EVAL / "ml2_share_grado_eval_sanity.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _resolve_block_base(df: pd.DataFrame) -> pd.DataFrame:
    if "bloque_base" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_base"])
        return df
    if "bloque_padre" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_padre"])
        return df
    if "bloque" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque"])
        return df
    raise KeyError("No encuentro columna de bloque: bloque_base / bloque_padre / bloque")

def mae(x: pd.Series) -> float:
    x = pd.to_numeric(x, errors="coerce")
    return float(np.nanmean(np.abs(x))) if len(x) else np.nan

def wape_weighted(abs_err: pd.Series, w: pd.Series) -> float:
    abs_err = pd.to_numeric(abs_err, errors="coerce").fillna(0.0)
    w = pd.to_numeric(w, errors="coerce").fillna(0.0)
    denom = float(np.sum(w))
    if denom <= 0:
        return np.nan
    return float(np.sum(abs_err * w) / denom)

```

```

def main() -> None:
    fac = read_parquet(IN_FACTOR).copy()
    fin = read_parquet(IN_FINAL).copy()
    real = read_parquet(IN_REAL).copy()
    mll = read_parquet(IN_PRED_ML1).copy()

    # --- Alias handling: share_final ---
    if "share_final" not in fac.columns:
        # intentos comunes
        cands = [c for c in fac.columns if ("share" in c.lower()) and ("final" in c.lower())]
        if cands:
            fac = fac.rename(columns={cands[0]: "share_final"})
        else:
            raise KeyError(f"No encuentro share_final en factor. Columnas disponibles: {list(fac.columns)}")

    # Canon
    for df in (fac, fin, mll):
        df["fecha"] = _to_date(df["fecha"])
        df["bloque_base"] = _canon_str(df["bloque_base"])
        df["grado"] = _canon_str(df["grado"])
        if "variedad_canon" in df.columns:
            df["variedad_canon"] = _canon_str(df["variedad_canon"])

    real["fecha"] = _to_date(real["fecha"])
    real = _resolve_block_base(real)
    real["grado"] = _canon_str(real["grado"])
    real["tallos_real"] = pd.to_numeric(real["tallos_real"], errors="coerce").fillna(0.0)

    # Real share por día/bloque
    rg = real.groupby(["fecha", "bloque_base", "grado"], as_index=False).agg(tallos_real=("tallos_real", "sum"))

    tot = rg.groupby(["fecha", "bloque_base"], as_index=False).agg(tallos_real_dia=("tallos_real", "sum"))
    rg = rg.merge(tot, on=["fecha", "bloque_base"], how="left")
    rg["share_real"] = np.where(rg["tallos_real_dia"] > 0, rg["tallos_real"] / rg["tallos_real_dia"], np.nan)

    # Mll share
    mll_need = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_mll", "tallos_pred_mll_grado_dia"]
    mll = mll[mll_need].copy()
    mll["share_grado_mll"] = pd.to_numeric(mll["share_grado_mll"], errors="coerce")
    mll["tallos_pred_mll_grado_dia"] = pd.to_numeric(mll["tallos_pred_mll_grado_dia"], errors="coerce").fillna(0.0)

    # Merge everything at grain (ciclo_id, fecha, bloque_base, variedad_canon, grado)
    fac_cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"]
    opt = []
    if "share_mll" in fac.columns:
        opt.append("share_mll")
    elif "share_grado_mll" in fac.columns:
        fac = fac.rename(columns={"share_grado_mll": "share_mll"})
        opt.append("share_mll")

    # fin YA trae share_final (porque apply lo guarda ahí).
    # del factor solo necesitamos share_mll (baseline) para comparar.

    fac_cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"]

    # normalizamos share_mll desde factor
    if "share_mll" in fac.columns:
        fac_use = fac[fac_cols + ["share_mll"]].copy()
    elif "share_grado_mll" in fac.columns:
        fac_use = fac.rename(columns={"share_grado_mll": "share_mll"})[fac_cols + ["share_mll"]].copy()
    else:
        raise KeyError(f"No encuentro share_mll / share_grado_mll en factor. Columnas: {list(fac.columns)}")

    df = fin.merge(
        fac_use,
        on=["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"],
        how="left",
    )

```

```

df = df.merge(m11, on=["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"], how="left")
df = df.merge(rg[["fecha", "bloque_base", "grado", "share_real", "tallos_real", "tallos_real_dia"]],
              on=["fecha", "bloque_base", "grado"], how="left")

# Cast
df["share_final"] = pd.to_numeric(df["share_final"], errors="coerce")
df["share_grado_m11"] = pd.to_numeric(df["share_grado_m11"], errors="coerce")
df["share_real"] = pd.to_numeric(df["share_real"], errors="coerce")

df["tallos_final_grado_dia"] = pd.to_numeric(df["tallos_final_grado_dia"], errors="coerce")
df["tallos_real"] = pd.to_numeric(df["tallos_real"], errors="coerce")
df["tallos_total_m12"] = pd.to_numeric(df["tallos_total_m12"], errors="coerce")

# --- KPI share error ---
m = df["share_real"].notna() & df["share_grado_m11"].notna() & df["share_final"].notna()
d = df.loc[m].copy()

d["err_share_m11"] = d["share_real"] - d["share_grado_m11"]
d["err_share_m12"] = d["share_real"] - d["share_final"]
d["abs_err_share_m11"] = d["err_share_m11"].abs()
d["abs_err_share_m12"] = d["err_share_m12"].abs()

# Weighted by tallos_real_dia (impacto por día)
w = pd.to_numeric(d["tallos_real_dia"], errors="coerce").fillna(0.0)

# --- KPI tallos por grado error (impacto directo) ---
# Error absoluto en tallos (si hay real)
d["abs_err_tallos_m11"] = (d["tallos_real"] - d["tallos_pred_m11_grado_dia"]).abs()
d["abs_err_tallos_m12"] = (d["tallos_real"] - d["tallos_final_grado_dia"]).abs()

# Global
out_g = pd.DataFrame([
    {
        "n_rows": int(len(d)),
        "n_cycles": int(d["ciclo_id"].nunique()),
        "mae_share_m11": mae(d["err_share_m11"]),
        "mae_share_m12": mae(d["err_share_m12"]),
        "wape_share_m11_wt_day": wape_weighted(d["abs_err_share_m11"], w),
        "wape_share_m12_wt_day": wape_weighted(d["abs_err_share_m12"], w),
        "mae_abs_tallos_m11": float(np.nanmean(d["abs_err_tallos_m11"])),
        "mae_abs_tallos_m12": float(np.nanmean(d["abs_err_tallos_m12"])),
        "improvement_abs_mae_share": mae(d["err_share_m11"]) - mae(d["err_share_m12"]),
        "improvement_abs_mae_abs_tallos": float(np.nanmean(d["abs_err_tallos_m11"])) - np.nanmean(d["abs_err_tallos_m12"]),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }
])

# By grado
rows = []
for grado, g in d.groupby("grado"):
    ww = pd.to_numeric(g["tallos_real_dia"], errors="coerce").fillna(0.0)
    rows.append({
        "grado": str(grado),
        "n_rows": int(len(g)),
        "mae_share_m11": mae(g["err_share_m11"]),
        "mae_share_m12": mae(g["err_share_m12"]),
        "wape_share_m11_wt_day": wape_weighted(g["abs_err_share_m11"], ww),
        "wape_share_m12_wt_day": wape_weighted(g["abs_err_share_m12"], ww),
        "mae_abs_tallos_m11": float(np.nanmean(g["abs_err_tallos_m11"])),
        "mae_abs_tallos_m12": float(np.nanmean(g["abs_err_tallos_m12"])),
        "improvement_abs_mae_share": mae(g["err_share_m11"]) - mae(g["err_share_m12"]),
    })
out_bg = pd.DataFrame(rows).sort_values("improvement_abs_mae_share", ascending=False)

# --- Sanity: sum shares ---
s = fac.groupby(["ciclo_id", "fecha"], as_index=False).agg(sum_share_final=("share_final", "sum"))
sanity = pd.DataFrame([
    {
        "n_days": int(len(s)),
        "sum_share_min": float(s["sum_share_final"].min()),
        "sum_share_p25": float(s["sum_share_final"].quantile(0.25)),
    }
])

```

```

    "sum_share_median": float(s["sum_share_final"].median()),
    "sum_share_p75": float(s["sum_share_final"].quantile(0.75)),
    "sum_share_max": float(s["sum_share_final"].max()),
    "pct_close_lpm_1e-6": float(np.mean(np.isclose(s["sum_share_final"].values, 1.0, atol=1e-6))),
    "created_at": pd.Timestamp(datetime.now()).normalize(),
  })
})

```

```

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(out_g, OUT_GLOBAL)
write_parquet(out_bg, OUT_BY_GRADO)
write_parquet(sanity, OUT_SANITY)

print(f"[OK] Wrote global: {OUT_GLOBAL}")
print(out_g.to_string(index=False))
print(f"\n[OK] Wrote by_grado: {OUT_BY_GRADO} rows={len(out_bg)}")
print(out_bg.head(10).to_string(index=False))
print(f"\n[OK] Wrote sanity: {OUT_SANITY}")
print(sanity.to_string(index=False))

```

```

if __name__ == "__main__":
    main()

```

```

-----
[43/164] FILE: \src\eval\build_kpis_tallos_curve_ml2.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

```

```

from src.common.io import read_parquet, write_parquet

```

```

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

```

```

ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

```

```

IN_FACTORS = EVAL_DIR / "backtest_factor_ml2_tallos_curve_dia.parquet"
IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"

```

```

OUT_GLOBAL = EVAL_DIR / "ml2_tallos_curve_eval_global.parquet"
OUT_GROUP = EVAL_DIR / "ml2_tallos_curve_eval_by_group.parquet"
OUT_RATIO = EVAL_DIR / "ml2_tallos_curve_eval_ratio_dist.parquet"

```

```

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

```

```

def _mae(err: pd.Series) -> float:
    err = pd.to_numeric(err, errors="coerce")
    return float(np.nanmean(np.abs(err))) if len(err) else float("nan")

```

```

def _wape(y_true: pd.Series, y_pred: pd.Series) -> float:
    y_true = pd.to_numeric(y_true, errors="coerce").fillna(0.0)
    y_pred = pd.to_numeric(y_pred, errors="coerce").fillna(0.0)
    denom = float(np.sum(np.abs(y_true)))
    if denom <= 0:
        return float("nan")
    return float(np.sum(np.abs(y_true - y_pred)) / denom)

```

```

def _smape(y_true: pd.Series, y_pred: pd.Series) -> float:
    y_true = pd.to_numeric(y_true, errors="coerce").fillna(0.0)
    y_pred = pd.to_numeric(y_pred, errors="coerce").fillna(0.0)
    denom = np.abs(y_true) + np.abs(y_pred)
    m = denom > 0
    if not m.any():
        return float("nan")
    return float(np.mean(2.0 * np.abs(y_true[m] - y_pred[m]) / denom[m]))

def main() -> None:
    df = read_parquet(IN_FACTORS).copy()

    # Attach grouping fields if missing (estado / tipo_sp / area)
    ciclo = read_parquet(IN_CICLO).copy()
    if "ciclo_id" not in ciclo.columns:
        raise KeyError("fact_ciclo_maestro debe tener ciclo_id")

    # Normalizar estado si viene
    if "estado" in ciclo.columns:
        ciclo["estado"] = ciclo["estado"].astype(str).str.upper().str.strip()

    ciclo_cols = ["ciclo_id"]
    for c in ["estado", "tipo_sp", "area"]:
        if c in ciclo.columns:
            ciclo_cols.append(c)

    df = df.merge(ciclo[ciclo_cols].drop_duplicates("ciclo_id"), on="ciclo_id", how="left")

    # Core
    df["tallos_real_dia"] = pd.to_numeric(df["tallos_real_dia"], errors="coerce").fillna(0.0)
    df["tallos_pred_ml1_dia"] = pd.to_numeric(df["tallos_pred_ml1_dia"], errors="coerce").fillna(0.0)
    df["tallos_final_ml2_dia"] = pd.to_numeric(df["tallos_final_ml2_dia"], errors="coerce").fillna(0.0)

    # Solo rows con señal (evita universo vacío)
    df = df.loc[(df["tallos_real_dia"] > 0) | (df["tallos_pred_ml1_dia"] > 0), :].copy()

    # Errors
    df["err_ml1"] = df["tallos_real_dia"] - df["tallos_pred_ml1_dia"]
    df["err_ml2"] = df["tallos_real_dia"] - df["tallos_final_ml2_dia"]

    # Global KPIs
    out_g = pd.DataFrame([
        "n_rows": int(len(df)),
        "n_cycles": int(df["ciclo_id"].nunique()),
        "mae_ml1": _mae(df["err_ml1"]),
        "mae_ml2": _mae(df["err_ml2"]),
        "wape_ml1": _wape(df["tallos_real_dia"], df["tallos_pred_ml1_dia"]),
        "wape_ml2": _wape(df["tallos_real_dia"], df["tallos_final_ml2_dia"]),
        "smape_ml1": _smape(df["tallos_real_dia"], df["tallos_pred_ml1_dia"]),
        "smape_ml2": _smape(df["tallos_real_dia"], df["tallos_final_ml2_dia"]),
        "improvement_abs_mae": _mae(df["err_ml1"]) - _mae(df["err_ml2"]),
        "improvement_abs_wape": _wape(df["tallos_real_dia"], df["tallos_pred_ml1_dia"]) - _wape(df["tallos_real_dia"], df["tallos_final_ml2_dia"]),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    ])

    # Ratio distribution sanity
    r = pd.to_numeric(df["pred_ratio"], errors="coerce")
    out_r = pd.DataFrame([
        "n": int(r.notna().sum()),
        "ratio_min": float(np.nanmin(r)) if len(r) else np.nan,
        "ratio_p05": float(np.nanpercentile(r, 5)) if len(r) else np.nan,
        "ratio_p25": float(np.nanpercentile(r, 25)) if len(r) else np.nan,
        "ratio_median": float(np.nanmedian(r)) if len(r) else np.nan,
        "ratio_p75": float(np.nanpercentile(r, 75)) if len(r) else np.nan,
        "ratio_p95": float(np.nanpercentile(r, 95)) if len(r) else np.nan,
        "ratio_max": float(np.nanmax(r)) if len(r) else np.nan,
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    ])

    # Grouping: prefer estado, else tipo_sp, else area

```

```

if "estado" in df.columns and df["estado"].notna().any():
    group_col = "estado"
elif "tipo_sp" in df.columns and df["tipo_sp"].notna().any():
    group_col = "tipo_sp"
else:
    group_col = "area" if "area" in df.columns else None

rows = []
if group_col:
    for k, g in df.groupby(group_col):
        rows.append({
            "group_col": group_col,
            "group_val": str(k),
            "n_rows": int(len(g)),
            "n_cycles": int(g["ciclo_id"].nunique()),
            "mae_ml1": _mae(g["err_ml1"]),
            "mae_ml2": _mae(g["err_ml2"]),
            "wape_ml1": _wape(g["tallos_real_dia"], g["tallos_pred_ml1_dia"]),
            "wape_ml2": _wape(g["tallos_real_dia"], g["tallos_final_ml2_dia"]),
            "improvement_abs_mae": _mae(g["err_ml1"]) - _mae(g["err_ml2"]),
            "improvement_abs_wape": _wape(g["tallos_real_dia"], g["tallos_pred_ml1_dia"]) - _wape(g["tallos_real_dia"], g["tallos_final_ml2_dia"]),
        })
    out_grp = pd.DataFrame(rows)

# Write
EVAL_DIR.mkdir(parents=True, exist_ok=True)
write_parquet(out_g, OUT_GLOBAL)
write_parquet(out_grp, OUT_GROUP)
write_parquet(out_r, OUT_RATIO)

print(f"[OK] Wrote global: {OUT_GLOBAL}")
print(out_g.to_string(index=False))
print(f"\n[OK] Wrote group : {OUT_GROUP} (group_col={group_col}) rows={len(out_grp)}")
if not out_grp.empty:
    print(out_grp.sort_values("improvement_abs_wape", ascending=False).head(15).to_string(index=False))
print(f"\n[OK] Wrote ratio : {OUT_RATIO}")
print(out_r.to_string(index=False))

if __name__ == "__main__":
    main()

```

```

[44/164] FILE: \src\features\build_cap_tallos_real_dia.py

```

```

from __future__ import annotations

```

```

from pathlib import Path
import numpy as np
import pandas as pd

```

```

from common.io import read_parquet, write_parquet

```

```

PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")
DIM_VAR_PATH = Path("data/silver/dim_variedad_canon.parquet")
FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")

```

```

OUT_PATH = Path("data/gold/dim_cap_tallos_real_dia.parquet")

```

```

PCTL_MAIN = 0.99
PCTL_ALT = 0.95
MIN_CAP_FLOOR = 500.0      # seguridad mínima
CAP_FALLBACK_GLOBAL = 4000.0 # si todo falla (debería no usarse)

```

```

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

```

```

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

```



```

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _load_var_map(dim_var: pd.DataFrame) -> dict[str, str]:
    _require(dim_var, ["variedad_raw", "variedad_canon"], "dim_variedad_canon")
    dv = dim_var.copy()
    dv["variedad_raw"] = _canon_str(dv["variedad_raw"])
    dv["variedad_canon"] = _canon_str(dv["variedad_canon"])
    dv = dv.dropna(subset=["variedad_raw", "variedad_canon"]).drop_duplicates(subset=["variedad_raw"])
    return dict(zip(dv["variedad_raw"], dv["variedad_canon"]))

def _q(s: pd.Series, q: float) -> float:
    a = pd.to_numeric(s, errors="coerce").dropna().to_numpy(dtype=float)
    if len(a) == 0:
        return float("nan")
    return float(np.quantile(a, q))

def main() -> None:
    for p in [PROG_PATH, DIM_VAR_PATH, FEATURES_PATH]:
        if not p.exists():
            raise FileNotFoundError(f"No existe: {p}")

    prog = read_parquet(PROG_PATH).copy()
    dim_var = read_parquet(DIM_VAR_PATH).copy()
    feat = read_parquet(FEATURES_PATH).copy()

    var_map = _load_var_map(dim_var)

    _require(prog, ["ciclo_id", "fecha", "tallos_real_dia"], "prog")
    if ("variedad" not in prog.columns) and ("variedad_canon" not in prog.columns):
        raise ValueError("prog: falta variedad o variedad_canon")

    _require(feat, ["ciclo_id", "fecha", "variedad_canon"], "features")
    for c in ["area", "tipo_sp"]:
        if c not in feat.columns:
            feat[c] = "UNKNOWN"

    # --- Canon prog
    prog["ciclo_id"] = prog["ciclo_id"].astype(str)
    prog["fecha"] = _to_date(prog["fecha"])
    prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce").fillna(0.0)

    if "variedad" in prog.columns:
        prog["variedad_raw"] = _canon_str(prog["variedad"])
        prog["variedad_canon"] = prog["variedad_raw"].map(var_map).fillna(prog["variedad_raw"])
    else:
        prog["variedad_canon"] = _canon_str(prog["variedad_canon"])

    prog["variedad_canon"] = _canon_str(prog["variedad_canon"])

    # --- Canon features
    feat["ciclo_id"] = feat["ciclo_id"].astype(str)
    feat["fecha"] = _to_date(feat["fecha"])
    feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
    feat["area"] = _canon_str(feat["area"].fillna("UNKNOWN"))
    feat["tipo_sp"] = _canon_str(feat["tipo_sp"].fillna("UNKNOWN"))

    # =====
    # 1) Agregar real por día (ciclo+fecha+variedad) para que el cap sea "físico"
    # =====
    real_day = (
        prog.groupby(["ciclo_id", "fecha", "variedad_canon"], dropna=False)["tallos_real_dia"]
        .sum()
        .reset_index()
    )

    # =====
    # 2) Adjuntar segmento (area/tipo_sp) sin usar bloque_base (evita el error)
    # Si hay múltiples filas en features para el mismo (ciclo, fecha, variedad) tomamos el primero.
    # =====

```

```

seg = (
    feat[["ciclo_id", "fecha", "variedad_canon", "area", "tipo_sp"]]
    .dropna(subset=["ciclo_id", "fecha", "variedad_canon"])
    .drop_duplicates(subset=["ciclo_id", "fecha", "variedad_canon"])
)

real_day = real_day.merge(seg, on=["ciclo_id", "fecha", "variedad_canon"], how="left")
real_day["area"] = _canon_str(real_day["area"].fillna("UNKNOWN"))
real_day["tipo_sp"] = _canon_str(real_day["tipo_sp"].fillna("UNKNOWN"))

# Solo días con real>0
x = real_day.loc[real_day["tallos_real_dia"] > 0, ["area", "tipo_sp", "variedad_canon", "tallos_real_dia"]]
].copy()
if len(x) == 0:
    raise ValueError("No hay tallos_real_dia>0 para calcular caps.")

# =====
# 3) Caps por (area,tipo_sp,variedad) + fallback por variedad + fallback global
# =====
caps = (
    x.groupby(["area", "tipo_sp", "variedad_canon"], dropna=False)["tallos_real_dia"]
    .agg(
        n="size",
        cap_p99=lambda s: _q(s, PCTL_MAIN),
        cap_p95=lambda s: _q(s, PCTL_ALT),
        mean=lambda s: float(pd.to_numeric(s, errors="coerce").mean()),
    )
    .reset_index()
)

var_caps = (
    x.groupby(["variedad_canon"], dropna=False)["tallos_real_dia"]
    .agg(
        cap_var_p99=lambda s: _q(s, PCTL_MAIN),
        cap_var_p95=lambda s: _q(s, PCTL_ALT),
        mean_var=lambda s: float(pd.to_numeric(s, errors="coerce").mean()),
    )
    .reset_index()
)

global_cap = float(_q(x["tallos_real_dia"], PCTL_MAIN))
if not np.isfinite(global_cap) or global_cap <= 0:
    global_cap = CAP_FALLBACK_GLOBAL

caps = caps.merge(var_caps, on="variedad_canon", how="left")

# Cap final: p99 del segmento; si no, p99 por variedad; si no, global; y piso
cap_final = caps["cap_p99"]
cap_final = cap_final.where(cap_final.notna(), caps["cap_var_p99"])
cap_final = cap_final.fillna(global_cap)
cap_final = pd.to_numeric(cap_final, errors="coerce").fillna(global_cap)
cap_final = cap_final.clip(lower=MIN_CAP_FLOOR)

caps["cap_dia"] = cap_final
caps["cap_global_p99"] = global_cap
caps["created_at"] = pd.Timestamp.utcnow()

write_parquet(caps, OUT_PATH)
print(f"OK -> {OUT_PATH} | rows={len(caps):,} | global_p99={global_cap:.1f}")

if __name__ == "__main__":
    main()

-----
[45/164] FILE: \src\features\build_features_ciclo_fecha.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd

```

```

import numpy as np
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def add_semana_ecuador(df: pd.DataFrame, col_fecha: str = "fecha") -> pd.Series:
    """
    Semana_ como tú la has manejado (ajustando +2 días y semana empezando Sunday).
    Devuelve YYWW en texto.
    """
    f = pd.to_datetime(df[col_fecha], errors="coerce")
    f2 = f + pd.to_timedelta(2, unit="D")
    yy = (f2.dt.year % 100).astype("Int64").astype(str).str.zfill(2)
    ww = f2.dt.isocalendar().week.astype("Int64").astype(str).str.zfill(2)
    return yy + ww

def main() -> None:
    cfg = load_settings()

    silver_dir = Path(cfg["paths"]["silver"])
    features_dir = Path(cfg["paths"]["features"])
    features_dir.mkdir(parents=True, exist_ok=True)

    # Inputs
    fact_path = silver_dir / "fact_ciclo_maestro.parquet"
    grid_path = silver_dir / "grid_ciclo_fecha.parquet"
    milestones_path = silver_dir / "fact_milestones_ciclo.parquet"
    windows_path = silver_dir / "milestone_window_ciclo_final.parquet"

    for p in [fact_path, grid_path, milestones_path, windows_path]:
        if not p.exists():
            raise FileNotFoundError(f"No existe: {p}")

    fact = read_parquet(fact_path)
    grid = read_parquet(grid_path)
    milestones = read_parquet(milestones_path)
    windows = read_parquet(windows_path)

    # Normalizar fechas
    grid["fecha"] = _norm_date(grid["fecha"])
    milestones["fecha"] = _norm_date(milestones["fecha"])
    windows["start_date"] = _norm_date(windows["start_date"])
    windows["end_date"] = _norm_date(windows["end_date"])

    # 1) Base: grid + atributos del ciclo
    cols_fact = [
        "ciclo_id",
        "bloque", "bloque_padre",
        "variedad", "tipo_sp",
        "area", "estado",
        "fecha_sp",
        "fecha_inicio_cosecha",
        "fecha_fin_cosecha",
        "tallos_proy",
    ]
    cols_fact = [c for c in cols_fact if c in fact.columns]

    base = grid.merge(fact[cols_fact], on="ciclo_id", how="left")

    # 2) Pivot de milestones para tener columnas (harvest_start, post_start, etc.)
    piv = (milestones.pivot_table(

```

```

        index="ciclo_id",
        columns="milestone_code",
        values="fecha",
        aggfunc="min"
    ).reset_index()

# Renombres canónicos
rename_map = {
    "VEG_START": "veg_start",
    "HARVEST_START": "harvest_start",
    "HARVEST_END": "harvest_end",
    "POST_START": "post_start",
    "POST_END": "post_end",
}
for k, v in rename_map.items():
    if k in piv.columns:
        piv = piv.rename(columns={k: v})

base = base.merge(piv, on="ciclo_id", how="left")

# 3) Etiquetar stage por ventanas (VEG/HARVEST/POST)
# Regla: si fecha entre [start_date, end_date] (end inclusive). Si end_date es NaT => ventana abierta.
# Precedencia: POST > HARVEST > VEG (para evitar solapes).
windows2 = windows.copy()
windows2["end_date_filled"] = windows2["end_date"].fillna(pd.Timestamp("2100-01-01"))

# Join expandido (puede ser pesado si ventanas están mal; con 180 días suele estar OK)
# Creamos stage por ciclo/fecha aplicando condiciones por stage.
base["stage"] = "OUT"

def apply_stage(stage_name: str):
    w = windows2[windows2["stage"] == stage_name][["ciclo_id", "start_date", "end_date_filled"]]
    tmp = base.merge(w, on="ciclo_id", how="left", suffixes=("", "_w"))
    cond = (tmp["fecha"] >= tmp["start_date"]) & (tmp["fecha"] <= tmp["end_date_filled"])
    return cond

# Precedencia: VEG primero, luego HARVEST sobreescribe, luego POST sobreescribe
veg_cond = apply_stage("VEG")
base.loc[veg_cond, "stage"] = "VEG"

harv_cond = apply_stage("HARVEST")
base.loc[harv_cond, "stage"] = "HARVEST"

post_cond = apply_stage("POST")
base.loc[post_cond, "stage"] = "POST"

# 4) Features de tiempo (días)
# d_desde_sp
if "fecha_sp" in base.columns:
    base["fecha_sp"] = _norm_date(base["fecha_sp"])
    base["d_desde_sp"] = (base["fecha"] - base["fecha_sp"]).dt.days.astype("Int64")
else:
    base["d_desde_sp"] = pd.Series([pd.NA] * len(base), dtype="Int64")

# distancias a hitos (si existen)
for col in ["harvest_start", "post_start"]:
    if col in base.columns:
        base[f"d_hasta_{col}"] = (base[col] - base["fecha"]).dt.days.astype("Int64")
    else:
        base[f"d_hasta_{col}"] = pd.Series([pd.NA] * len(base), dtype="Int64")

# 5) Calendario
base["anio"] = base["fecha"].dt.year.astype("Int64")
base["mes"] = base["fecha"].dt.month.astype("Int64")
base["semana_"] = add_semana_ecuador(base, "fecha")

# 6) Auditoría mínima
base["created_at"] = datetime.now().isoformat(timespec="seconds")

# 7) Escritura
out_path = features_dir / "features_ciclo_fecha.parquet"
write_parquet(base, out_path)

```

```

print(f"OK: features_ciclo_fecha={len(base)} filas -> {out_path}")
print("Stage counts:\n", base["stage"].value_counts(dropna=False).to_string())

if __name__ == "__main__":
    main()

-----
[46/164] FILE: \src\features\build_features_cosecha_bloque_fecha.py
-----

from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

# =====
# Helpers
# =====
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _prep_dim_var(dim_var: pd.DataFrame) -> pd.DataFrame:
    need = {"variedad_raw", "variedad_canon"}
    miss = need - set(dim_var.columns)
    if miss:
        raise ValueError(f"dim_variedad_canon.parquet sin columnas: {sorted(miss)}")

    dv = dim_var.copy()
    dv["variedad_raw_norm"] = _canon_str(dv["variedad_raw"])
    dv["variedad_canon"] = _canon_str(dv["variedad_canon"])
    return dv[["variedad_raw_norm", "variedad_canon"]].drop_duplicates()

def _attach_variedad_canon(df: pd.DataFrame, dv: pd.DataFrame, col_raw: str) -> pd.DataFrame:
    out = df.copy()
    if col_raw not in out.columns:
        raise ValueError(f"DF no tiene columna '{col_raw}' para canonizar variedad.")
    out[col_raw] = _canon_str(out[col_raw])
    out = out.merge(dv, left_on=col_raw, right_on="variedad_raw_norm", how="left")
    out["variedad_canon"] = out["variedad_canon"].fillna(out[col_raw])
    return out.drop(columns=["variedad_raw_norm"], errors="ignore")

def _renormalize_share(df: pd.DataFrame, share_col: str, group_cols: list[str]) -> pd.DataFrame:
    out = df.copy()
    out[share_col] = pd.to_numeric(out[share_col], errors="coerce").fillna(0.0).clip(lower=0.0)
    s = out.groupby(group_cols, dropna=False)[share_col].transform("sum")
    out[share_col] = np.where(s > 0, out[share_col] / s, 0.0)
    return out

def _attach_baseline_nearest_ndias(
    feat: pd.DataFrame,
    base: pd.DataFrame,
    *,
    ndias_feat_col: str,
    ndias_base_col: str = "n_dias",

```

```

    value_col: str,
    out_col: str,
) -> pd.DataFrame:
    out = feat.copy()
    out["_nd"] = pd.to_numeric(out[ndias_feat_col], errors="coerce")

    b = base.copy()
    b["_nd"] = pd.to_numeric(b[ndias_base_col], errors="coerce")

    out["grado"] = _canon_int(out["grado"])
    b["grado"] = _canon_int(b["grado"])

    b = b.dropna(subset=["variedad_canon", "grado", "_nd"]).copy()
    b = b[["variedad_canon", "grado", "_nd", value_col]].copy()
    b = b.sort_values(["variedad_canon", "grado", "_nd"], kind="mergesort").reset_index(drop=True)

    parts: list[pd.DataFrame] = []
    for (var, g), sub in out.groupby(["variedad_canon", "grado"], dropna=False):
        sub = sub.copy()
        ref = b[(b["variedad_canon"] == var) & (b["grado"] == g)]
        if ref.empty:
            sub[out_col] = np.nan
            parts.append(sub.drop(columns=["_nd"], errors="ignore"))
            continue

        sub_null = sub[sub["_nd"].isna()].copy()
        sub_ok = sub[sub["_nd"].notna()].copy()

        sub_null[out_col] = np.nan

        if not sub_ok.empty:
            sub_ok = sub_ok.sort_values(["_nd"], kind="mergesort").reset_index(drop=True)
            ref2 = ref[["_nd", value_col]].sort_values(["_nd"], kind="mergesort").reset_index(drop=True)

            m = pd.merge_asof(
                sub_ok,
                ref2.rename(columns={value_col: out_col}),
                on="_nd",
                direction="nearest",
                allow_exact_matches=True,
            )
            m = m.drop(columns=["_nd"], errors="ignore")
            parts.append(m)

        sub_null = sub_null.drop(columns=["_nd"], errors="ignore")
        parts.append(sub_null)

    out2 = pd.concat(parts, ignore_index=True) if parts else out.drop(columns=["_nd"], errors="ignore")
    return out2

# =====
# Main
# =====
def main() -> None:
    created_at = pd.Timestamp.utcnow()

    # -----
    # Inputs (BASE = UNIVERSO ML1)
    # -----
    grid = read_parquet(Path("data/gold/universe_harvest_grid_ml1.parquet")).copy()
    df_prog = read_parquet(Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")).copy()
    df_clima = read_parquet(Path("data/silver/dim_clima_bloque_dia.parquet")).copy()
    df_term = read_parquet(Path("data/silver/dim_estado_termico_cultivo_bloque_fecha.parquet")).copy()

    df_real_cosecha_grado = read_parquet(Path("data/silver/fact_cosecha_real_grado_dia.parquet")).copy()
    df_real_peso = read_parquet(Path("data/silver/fact_peso_tallo_real_grado_dia.parquet")).copy()

    df_base_dist = read_parquet(Path("data/silver/dim_dist_grado_baseline.parquet")).copy()
    df_base_peso = read_parquet(Path("data/silver/dim_peso_tallo_baseline.parquet")).copy()

    df_maestro = read_parquet(Path("data/silver/fact_ciclo_maestro.parquet")).copy()

```

```

dim_var = read_parquet(Path("data/silver/dim_variedad_canon.parquet")).copy()
dv = _prep_dim_var(dim_var)

# -----
# Canon universo
# -----
need_u = {"ciclo_id", "fecha", "bloque_base", "variedad_canon"}
miss_u = need_u - set(grid.columns)
if miss_u:
    raise ValueError(f"universe_harvest_grid_mll sin columnas: {sorted(miss_u)}")

grid["ciclo_id"] = grid["ciclo_id"].astype(str)
grid["fecha"] = _to_date(grid["fecha"])
grid["bloque_base"] = _canon_int(grid["bloque_base"])
grid["variedad_canon"] = _canon_str(grid["variedad_canon"])

if "stage" in grid.columns:
    grid = grid[_canon_str(grid["stage"]).eq("HARVEST")].copy()

# Aliases esperados por apply_dist_grado.py
if "day_in_harvest_pred" in grid.columns and "day_in_harvest" not in grid.columns:
    grid["day_in_harvest"] = pd.to_numeric(grid["day_in_harvest_pred"], errors="coerce").astype("Int64")
if "rel_pos_pred" in grid.columns and "rel_pos" not in grid.columns:
    grid["rel_pos"] = pd.to_numeric(grid["rel_pos_pred"], errors="coerce")
if "n_harvest_days_pred" in grid.columns and "n_harvest_days" not in grid.columns:
    grid["n_harvest_days"] = pd.to_numeric(grid["n_harvest_days_pred"], errors="coerce").astype("Int64")
if "harvest_start_pred" in grid.columns and "harvest_start" not in grid.columns:
    grid["harvest_start"] = _to_date(grid["harvest_start_pred"])
if "harvest_end_pred" in grid.columns and "harvest_end_eff" not in grid.columns:
    grid["harvest_end_eff"] = _to_date(grid["harvest_end_pred"])

base = grid.drop_duplicates(subset=["ciclo_id", "fecha", "bloque_base", "variedad_canon"]).copy()

# -----
# Maestro meta
# -----
df_maestro["ciclo_id"] = df_maestro["ciclo_id"].astype(str)
if "bloque_base" in df_maestro.columns:
    df_maestro["bloque_base"] = _canon_int(df_maestro["bloque_base"])
if "variedad_canon" not in df_maestro.columns and "variedad" in df_maestro.columns:
    df_maestro = _attach_variedad_canon(df_maestro, dv, "variedad")
if "variedad_canon" in df_maestro.columns:
    df_maestro["variedad_canon"] = _canon_str(df_maestro["variedad_canon"])
for c in ["tipo_sp", "area", "estado"]:
    if c in df_maestro.columns:
        df_maestro[c] = _canon_str(df_maestro[c])

m_take = [c for c in ["ciclo_id", "tipo_sp", "area", "estado"] if c in df_maestro.columns]
m2 = df_maestro[m_take].drop_duplicates("ciclo_id") if m_take else pd.DataFrame({"ciclo_id": base["ciclo_id"].unique()})
feat_day = base.merge(m2, on="ciclo_id", how="left")

# -----
# Expand a GRADOS (desde baseline dist)
# -----
need_bd = {"variedad", "grado", "n_dias", "pct_grado"}
miss_bd = need_bd - set(df_base_dist.columns)
if miss_bd:
    raise ValueError(f"dim_dist_grado_baseline sin columnas: {sorted(miss_bd)}")

base_dist = _attach_variedad_canon(df_base_dist, dv, "variedad")
base_dist["grado"] = _canon_int(base_dist["grado"])
base_dist["n_dias"] = _canon_int(base_dist["n_dias"])
base_dist = base_dist.rename(columns={"pct_grado": "share_grado_baseline"})
base_dist = base_dist.dropna(subset=["variedad_canon", "grado", "n_dias"])

# catálogo grados por variedad (si faltara una variedad, se quedará sin grados -> FATAL)
vg = base_dist[["variedad_canon", "grado"]].drop_duplicates()

feat = feat_day.merge(vg, on="variedad_canon", how="left")
if feat["grado"].isna().any():
    bad_vars = feat.loc[feat["grado"].isna(), "variedad_canon"].value_counts().head(20)

```

```
        raise ValueError(f"[FATAL] No pude expandir grados para algunas variedades. Ejemplos:\n{bad_vars.to_string()}")
```

```
feat["grado"] = _canon_int(feat["grado"])

# -----
# Canon otros inputs
# -----
for d in (df_prog, df_clima, df_term, df_real_cosecha_grado, df_real_peso):
    if "fecha" in d.columns:
        d["fecha"] = _to_date(d["fecha"])
if "fecha_sp" in df_term.columns:
    df_term["fecha_sp"] = _to_date(df_term["fecha_sp"])

# -----
# Progreso real (LEFT)
# -----
prog = df_prog.copy()
if "ciclo_id" in prog.columns:
    prog["ciclo_id"] = prog["ciclo_id"].astype(str)
if "bloque_base" in prog.columns:
    prog["bloque_base"] = _canon_int(prog["bloque_base"])
if "variedad_canon" not in prog.columns and "variedad" in prog.columns:
    prog = _attach_variedad_canon(prog, dv, "variedad")
if "variedad_canon" in prog.columns:
    prog["variedad_canon"] = _canon_str(prog["variedad_canon"])
else:
    prog["variedad_canon"] = "UNKNOWN"

prog_key = [c for c in ["ciclo_id", "fecha", "bloque_base", "variedad_canon"] if c in prog.columns]
if prog_key:
    prog2 = prog.drop_duplicates(subset=prog_key)
    feat = feat.merge(prog2, on=prog_key, how="left", suffixes=("", "_prog"))

# -----
# Clima (LEFT por fecha+bloque)
# -----
clima = df_clima.copy()
if "bloque_base" in clima.columns:
    clima["bloque_base"] = _canon_int(clima["bloque_base"])
clima2 = clima.drop_duplicates(subset=["fecha", "bloque_base"])
feat = feat.merge(clima2, on=["fecha", "bloque_base"], how="left", suffixes=("", "_cl"))

# -----
# Term (LEFT por ciclo+bloque+fecha)
# -----
term = df_term.copy()
if "bloque_base" in term.columns:
    term["bloque_base"] = _canon_int(term["bloque_base"])
if "ciclo_id" in term.columns:
    term["ciclo_id"] = term["ciclo_id"].astype(str)
term2 = term.drop_duplicates(subset=["ciclo_id", "bloque_base", "fecha"])
feat = feat.merge(term2, on=["ciclo_id", "bloque_base", "fecha"], how="left", suffixes=("", "_term"))

# -----
# Calendario
# -----
feat["dow"] = feat["fecha"].dt.dayofweek
feat["month"] = feat["fecha"].dt.month
feat["weekofyear"] = feat["fecha"].dt.isocalendar().week.astype(int)

# -----
# n_dias_cosecha (para baseline nearest)
# -----
a = pd.to_numeric(feat["dia_rel_cosecha_real"], errors="coerce") if "dia_rel_cosecha_real" in feat.columns
else pd.Series([np.nan] * len(feat))
b = pd.to_numeric(feat["day_in_harvest"], errors="coerce") if "day_in_harvest" in feat.columns else pd.Series([np.nan] * len(feat))
feat["n_dias_cosecha"] = a.combine_first(b)
feat["n_dias_cosecha"] = pd.to_numeric(feat["n_dias_cosecha"], errors="coerce").round()
feat.loc[feat["n_dias_cosecha"].isna(), "n_dias_cosecha"] = 1
feat["n_dias_cosecha"] = feat["n_dias_cosecha"].astype("Int64")
```



```

# -----
# Baseline dist por nearest n_dias + renorm por día
# -----
nd_min = int(base_dist["n_dias"].min())
nd_max = int(base_dist["n_dias"].max())
feat["n_dias_cosecha"] = feat["n_dias_cosecha"].clip(lower=nd_min, upper=nd_max)

feat = _attach_baseline_nearest_ndias(
    feat,
    base=base_dist,
    ndias_feat_col="n_dias_cosecha",
    ndias_base_col="n_dias",
    value_col="share_grado_baseline",
    out_col="share_grado_baseline",
)

grp_day = ["ciclo_id", "bloque_base", "variedad_canon", "fecha"]
feat["share_grado_baseline"] = feat["share_grado_baseline"].fillna(0.0)
feat = _renormalize_share(feat, "share_grado_baseline", grp_day)

# -----
# Baseline peso por nearest n_dias (si existe)
# -----
need_bp = {"variedad", "grado", "n_dias", "peso_tallo_mediana_g"}
miss_bp = need_bp - set(df_base_peso.columns)
if miss_bp:
    raise ValueError(f"dim_peso_tallo_baseline sin columnas: {sorted(miss_bp)}")

base_peso = _attach_variedad_canon(df_base_peso, dv, "variedad")
base_peso["grado"] = _canon_int(base_peso["grado"])
base_peso["n_dias"] = _canon_int(base_peso["n_dias"])
base_peso = base_peso.rename(columns={"peso_tallo_mediana_g": "peso_tallo_baseline_g"})
base_peso = base_peso.dropna(subset=["variedad_canon", "grado", "n_dias"])

feat = _attach_baseline_nearest_ndias(
    feat,
    base=base_peso.rename(columns={"peso_tallo_baseline_g": "val"}),
    ndias_feat_col="n_dias_cosecha",
    ndias_base_col="n_dias",
    value_col="val",
    out_col="peso_tallo_baseline_g",
)

# -----
# Targets reales (LEFT, no corta futuro)
# -----
real_c = df_real_cosecha_grado.copy()
real_c["fecha"] = _to_date(real_c["fecha"])
if "bloque_padre" in real_c.columns and "bloque_base" not in real_c.columns:
    real_c = real_c.rename(columns={"bloque_padre": "bloque_base"})
if "bloque_base" in real_c.columns:
    real_c["bloque_base"] = _canon_int(real_c["bloque_base"])
if "variedad_canon" not in real_c.columns and "variedad" in real_c.columns:
    real_c = _attach_variedad_canon(real_c, dv, "variedad")
if "grado" in real_c.columns:
    real_c["grado"] = _canon_int(real_c["grado"])

if "tallos_real" in real_c.columns:
    rc = (
        real_c.groupby(["fecha", "bloque_base", "variedad_canon", "grado"], as_index=False)
        .agg(tallos_real_grado=("tallos_real", "sum"))
    )
    rc_tot = (
        rc.groupby(["fecha", "bloque_base", "variedad_canon"], as_index=False)
        .agg(tallos_real_total=("tallos_real_grado", "sum"))
    )
    rc = rc.merge(rc_tot, on=["fecha", "bloque_base", "variedad_canon"], how="left")
    rc["share_grado_real"] = np.where(
        rc["tallos_real_total"].fillna(0) > 0,
        rc["tallos_real_grado"] / rc["tallos_real_total"],
        np.nan,
    )

```

```

    )
    feat = feat.merge(
        rc[["fecha", "bloque_base", "variedad_canon", "grado", "tallos_real_grado", "tallos_real_total", "
share_grado_real"]],
        on=["fecha", "bloque_base", "variedad_canon", "grado"],
        how="left",
    )

# -----
# Residuales (listos ML2)
# -----
if "share_grado_real" in feat.columns:
    feat["resid_share_grado"] = feat["share_grado_real"] - feat["share_grado_baseline"]

feat["created_at"] = created_at

# -----
# Output + checks
# -----
out_path = Path("data/features/features_cosecha_bloque_fecha.parquet")
key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"]
feat = feat.sort_values(key).reset_index(drop=True)

if feat.duplicated(subset=key).any():
    raise ValueError("[FATAL] Duplicados en features_cosecha_bloque_fecha por key completa (día+grado).")

write_parquet(feat, out_path)

fmin = pd.to_datetime(feat["fecha"].min()).date() if len(feat) else None
fmax = pd.to_datetime(feat["fecha"].max()).date() if len(feat) else None
print(f"OK -> {out_path} | rows={len(feat):,} | fecha_min={fmin} fecha_max={fmax}")

sums = feat.groupby(["ciclo_id", "bloque_base", "variedad_canon", "fecha"])["share_grado_baseline"].sum()
if len(sums):
    print(f"[CHECK] baseline share sum min/max: {float(sums.min()):.6f} / {float(sums.max()):.6f}")

cov_nd = float(feat["n_dias_cosecha"].notna().mean()) if len(feat) else float("nan")
print(f"[CHECK] n_dias_cosecha notna coverage: {cov_nd:.4f}")

if __name__ == "__main__":
    main()

-----
[47/164] FILE: \src\features\build_features_curva_cosecha_bloque_dia.py
-----
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

# -----
# Paths (BASE = UNIVERSO ML1)
# -----
IN_GRID = Path("data/gold/universe_harvest_grid_ml1.parquet")
IN_MAESTRO = Path("data/silver/fact_ciclo_maestro.parquet")

IN_PROG = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")
IN_CLIMA = Path("data/silver/dim_clima_bloque_dia.parquet")
IN_TERM = Path("data/silver/dim_estado_termico_cultivo_bloque_fecha.parquet")
IN_DIM_VAR = Path("data/silver/dim_variedad_canon.parquet")

OUT = Path("data/features/features_curva_cosecha_bloque_dia.parquet")

# -----
# Helpers

```

```

# -----
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _prep_dim_var(dim_var: pd.DataFrame) -> pd.DataFrame:
    need = {"variedad_raw", "variedad_canon"}
    miss = need - set(dim_var.columns)
    if miss:
        raise ValueError(f"dim_variedad_canon.parquet sin columnas: {sorted(miss)}")
    dv = dim_var.copy()
    dv["variedad_raw_norm"] = _canon_str(dv["variedad_raw"])
    dv["variedad_canon"] = _canon_str(dv["variedad_canon"])
    return dv[["variedad_raw_norm", "variedad_canon"]].drop_duplicates()

def _attach_variedad_canon(df: pd.DataFrame, dv: pd.DataFrame, col_raw: str) -> pd.DataFrame:
    out = df.copy()
    out[col_raw] = _canon_str(out[col_raw])
    out = out.merge(dv, left_on=col_raw, right_on="variedad_raw_norm", how="left")
    out["variedad_canon"] = out["variedad_canon"].fillna(out[col_raw])
    return out.drop(columns=["variedad_raw_norm"], errors="ignore")

# -----
# Main
# -----
def main() -> None:
    created_at = pd.Timestamp.utcnow()

    grid = read_parquet(IN_GRID).copy()
    maestro = read_parquet(IN_MAESTRO).copy()
    prog = read_parquet(IN_PROG).copy() if IN_PROG.exists() else pd.DataFrame()
    clima = read_parquet(IN_CLIMA).copy() if IN_CLIMA.exists() else pd.DataFrame()
    term = read_parquet(IN_TERM).copy() if IN_TERM.exists() else pd.DataFrame()
    dim_var = read_parquet(IN_DIM_VAR).copy()

    dv = _prep_dim_var(dim_var)

    # -----
    # Base: universo ML1
    # -----
    _require(grid, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_ml1")
    grid["ciclo_id"] = grid["ciclo_id"].astype(str)
    grid["fecha"] = _to_date(grid["fecha"])
    grid["bloque_base"] = _canon_int(grid["bloque_base"])
    grid["variedad_canon"] = _canon_str(grid["variedad_canon"])

    # quedarnos solo HARVEST por si acaso
    if "stage" in grid.columns:
        st = _canon_str(grid["stage"])
        grid = grid[st.eq("HARVEST")].copy()

    base_cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    # metadatos útiles si están en grid
    for c in ["area", "tipo_sp", "estado", "day_in_harvest_pred", "rel_pos_pred", "n_harvest_days_pred", "harv
est_start_pred", "harvest_end_pred"]:
        if c in grid.columns:

```

```

        base_cols.append(c)

base = grid[base_cols].drop_duplicates(subset=["ciclo_id", "fecha", "bloque_base", "variedad_canon"]).copy()
()

# -----
# Maestro: tallos_proy (driver) + meta
# -----
_require(maestro, ["ciclo_id", "tallos_proy"], "fact_ciclo_maestro")
maestro["ciclo_id"] = maestro["ciclo_id"].astype(str)
maestro["tallos_proy"] = pd.to_numeric(maestro["tallos_proy"], errors="coerce").fillna(0.0)

m_take = ["ciclo_id", "tallos_proy"]
for c in ["area", "tipo_sp", "estado", "variedad", "variedad_canon", "bloque_base"]:
    if c in maestro.columns:
        m_take.append(c)

m2 = maestro[m_take].drop_duplicates("ciclo_id").copy()
if "variedad_canon" not in m2.columns and "variedad" in m2.columns:
    m2 = _attach_variedad_canon(m2, dv, "variedad")
if "variedad_canon" in m2.columns:
    m2["variedad_canon"] = _canon_str(m2["variedad_canon"])
if "bloque_base" in m2.columns:
    m2["bloque_base"] = _canon_int(m2["bloque_base"])
for c in ["area", "tipo_sp", "estado"]:
    if c in m2.columns:
        m2[c] = _canon_str(m2[c])

df = base.merge(m2, on="ciclo_id", how="left", suffixes=("", "_m"))

# coalesce meta desde maestro si grid no lo trae
for c in ["area", "tipo_sp", "estado"]:
    if c not in df.columns and f"{c}_m" in df.columns:
        df[c] = df[f"{c}_m"]
    if f"{c}_m" in df.columns:
        df = df.drop(columns=[f"{c}_m"])

# -----
# Baseline diario: uniforme por ciclo en el UNIVERSO
# -----
cnt = df.groupby("ciclo_id", dropna=False)["fecha"].transform("count").astype(float)
df["tallos_pred_baseline_dia"] = np.where(cnt > 0, df["tallos_proy"].astype(float) / cnt, 0.0)

# -----
# Progreso real
# -----
if not prog.empty:
    prog["fecha"] = _to_date(prog["fecha"])
    if "ciclo_id" in prog.columns:
        prog["ciclo_id"] = prog["ciclo_id"].astype(str)
    if "bloque_base" in prog.columns:
        prog["bloque_base"] = _canon_int(prog["bloque_base"])
    elif "bloque_padre" in prog.columns:
        prog["bloque_base"] = _canon_int(prog["bloque_padre"])
    elif "bloque" in prog.columns:
        prog["bloque_base"] = _canon_int(prog["bloque"])

    if "variedad_canon" not in prog.columns and "variedad" in prog.columns:
        prog = _attach_variedad_canon(prog, dv, "variedad")
    if "variedad_canon" in prog.columns:
        prog["variedad_canon"] = _canon_str(prog["variedad_canon"])
    else:
        prog["variedad_canon"] = "UNKNOWN"

    prog_take = [c for c in [
        "ciclo_id", "fecha", "bloque_base", "variedad_canon",
        "tallos_real_dia", "pct_avance_real", "dia_rel_cosecha_real",
        "en_ventana_cosecha_real", "gdc_acum_real",
    ] if c in prog.columns]

    prog2 = prog[prog_take].drop_duplicates(subset=["ciclo_id", "fecha", "bloque_base", "variedad_canon"])
    df = df.merge(prog2, on=["ciclo_id", "fecha", "bloque_base", "variedad_canon"], how="left")

```

```

# -----
# Clima (por fecha + bloque_base)
# -----
if not clima.empty:
    clima["fecha"] = _to_date(clima["fecha"])
    if "bloque_base" in clima.columns:
        clima["bloque_base"] = _canon_int(clima["bloque_base"])
    elif "bloque_padre" in clima.columns:
        clima["bloque_base"] = _canon_int(clima["bloque_padre"])
    elif "bloque" in clima.columns:
        clima["bloque_base"] = _canon_int(clima["bloque"])

    clima_take = [c for c in [
        "fecha", "bloque_base",
        "rainfall_mm_dia", "horas_lluvia", "en_lluvia_dia",
        "temp_avg_dia", "solar_energy_j_m2_dia",
        "wind_speed_avg_dia", "wind_run_dia", "gdc_dia",
    ] if c in clima.columns]

    clima2 = clima[clima_take].drop_duplicates(subset=["fecha", "bloque_base"])
    df = df.merge(clima2, on=["fecha", "bloque_base"], how="left")

# -----
# Termal (por ciclo + fecha + bloque_base)
# -----
if not term.empty:
    term["fecha"] = _to_date(term["fecha"])
    if "fecha_sp" in term.columns:
        term["fecha_sp"] = _to_date(term["fecha_sp"])
    if "ciclo_id" in term.columns:
        term["ciclo_id"] = term["ciclo_id"].astype(str)

    if "bloque_base" in term.columns:
        term["bloque_base"] = _canon_int(term["bloque_base"])
    elif "bloque_padre" in term.columns:
        term["bloque_base"] = _canon_int(term["bloque_padre"])
    elif "bloque" in term.columns:
        term["bloque_base"] = _canon_int(term["bloque"])

    term_take = [c for c in [
        "ciclo_id", "bloque_base", "fecha",
        "fecha_sp", "dias_desde_sp", "gdc_acum_desde_sp",
    ] if c in term.columns]

    term2 = term[term_take].drop_duplicates(subset=["ciclo_id", "bloque_base", "fecha"])
    if set(["ciclo_id", "bloque_base", "fecha"]).issubset(term2.columns):
        df = df.merge(term2, on=["ciclo_id", "bloque_base", "fecha"], how="left")

# -----
# Calendario
# -----
df["dow"] = df["fecha"].dt.dayofweek
df["month"] = df["fecha"].dt.month
df["weekofyear"] = df["fecha"].dt.isocalendar().week.astype(int)

# -----
# Targets (si hay real)
# -----
if "tallos_real_dia" in df.columns:
    df["tallos_real_dia"] = pd.to_numeric(df["tallos_real_dia"], errors="coerce")

df["factor_tallos_dia"] = np.where(
    df["tallos_pred_baseline_dia"].fillna(0) > 0,
    df.get("tallos_real_dia", np.nan) / df["tallos_pred_baseline_dia"],
    np.nan,
)

df["factor_tallos_dia"] = pd.to_numeric(df["factor_tallos_dia"], errors="coerce")
df["factor_tallos_dia_clipped"] = df["factor_tallos_dia"].clip(lower=0.2, upper=5.0)

df["resid_tallos_dia"] = df.get("tallos_real_dia", np.nan) - df["tallos_pred_baseline_dia"]

```

```

# -----
# Final + checks
# -----
df["created_at"] = created_at

k = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
dup_rate = float(df.duplicated(subset=k).mean()) if len(df) else 0.0
if dup_rate > 0:
    raise ValueError(f"[FATAL] Duplicados en features_curva por {k}. dup_rate={dup_rate:.6f}")

write_parquet(df.sort_values(["bloque_base", "variedad_canon", "fecha"]).reset_index(drop=True), OUT)
print(f"OK -> {OUT} | rows={len(df):,} | fecha_min={df['fecha'].min().date()} fecha_max={df['fecha'].max().date()}")
print(f"[CHECK] dup_rate features_curva por {k}: {dup_rate:.6f}")
print(f"[COVERAGE] baseline notna: {float(df['tallos_pred_baseline_dia'].notna().mean()):.4f}")
if "tallos_real_dia" in df.columns:
    print(f"[COVERAGE] real notna: {float(df['tallos_real_dia'].notna().mean()):.4f}")

# Coverage vs universo (debe ser 1:1)
n_univ = len(base)
n_feat = len(df)
if n_feat != n_univ:
    raise ValueError(f"[FATAL] features_curva rows != universe rows ({n_feat} != {n_univ}). Debe ser 1:1."
)

```

```

if __name__ == "__main__":
    main()

```

```

-----
[48/164] FILE: \src\features\build_features_harvest_window_ml1.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import json
import numpy as np
import pandas as pd
import yaml

```

```

from common.io import read_parquet, write_parquet

```

```

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

```

```

# -----
# Helpers
# -----
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

```

```

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

```

```

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

```

```

def _pick_first(df: pd.DataFrame, candidates: list[str]) -> str | None:
    for c in candidates:
        if c in df.columns:
            return c
    return None

```

```

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def main() -> None:
    cfg = load_settings()
    silver_dir = Path(cfg.get("paths", {}).get("silver", "data/silver"))
    features_dir = Path(cfg.get("paths", {}).get("features", "data/features"))
    features_dir.mkdir(parents=True, exist_ok=True)

    maestro_path = silver_dir / "fact_ciclo_maestro.parquet"
    if not maestro_path.exists():
        raise FileNotFoundError(f"No existe: {maestro_path}")

    df = read_parquet(maestro_path).copy()
    df.columns = [str(c).strip() for c in df.columns]

    # ---- columnas base
    _require(df, ["ciclo_id", "fact_ciclo_maestro"])
    df["ciclo_id"] = df["ciclo_id"].astype(str)

    # llaves / categóricas
    if "bloque_base" not in df.columns:
        # fallback típicos
        bcol = _pick_first(df, ["bloque_base", "bloque_padre", "bloque"])
        if bcol is None:
            raise ValueError("fact_ciclo_maestro: no encontré bloque_base/bloque_padre/bloque")
        df["bloque_base"] = df[bcol]
    df["bloque_base"] = _canon_int(df["bloque_base"])

    if "tipo_sp" in df.columns:
        df["tipo_sp"] = _canon_str(df["tipo_sp"])
    else:
        df["tipo_sp"] = "UNKNOWN"

    if "area" in df.columns:
        df["area"] = _canon_str(df["area"])
    else:
        df["area"] = "UNKNOWN"

    # variedad_canon: la usaremos sí o sí
    # (asumo que ya existe en maestro; si no existe, intenta desde variedad)
    if "variedad_canon" in df.columns:
        df["variedad_canon"] = _canon_str(df["variedad_canon"])
    elif "variedad" in df.columns:
        # si aquí necesitas map con dim_variedad_canon, lo integramos luego;
        # por ahora canonizo directo para no romper.
        df["variedad_canon"] = _canon_str(df["variedad"])
    else:
        df["variedad_canon"] = "UNKNOWN"

    # tallos_proy opcional (puede ayudar)
    if "tallos_proy" in df.columns:
        df["tallos_proy"] = pd.to_numeric(df["tallos_proy"], errors="coerce")
    else:
        df["tallos_proy"] = np.nan

    # fechas: según tu schema real
    sp_col = _pick_first(df, ["fecha_sp", "sp_date", "fecha_siembra", "fecha_sp_real"])
    hs_col = _pick_first(df, ["fecha_inicio_cosecha", "harvest_start", "inicio_cosecha", "fecha_inicio_real"])
    he_col = _pick_first(df, ["fecha_fin_cosecha", "harvest_end_eff", "harvest_end", "fin_cosecha", "fecha_fin_real"])

    if sp_col is None:
        raise ValueError("fact_ciclo_maestro: no encontré columna de fecha SP (ej: fecha_sp).")

    df["fecha_sp"] = _to_date(df[sp_col])
    df["harvest_start_real"] = _to_date(df[hs_col]) if hs_col is not None else pd.NaT
    df["harvest_end_real"] = _to_date(df[he_col]) if he_col is not None else pd.NaT

```

```

# calendario de SP (para estacionalidad)
df["sp_month"] = df["fecha_sp"].dt.month
df["sp_weekofyear"] = df["fecha_sp"].dt.isocalendar().week.astype("Int64")
df["sp_doy"] = df["fecha_sp"].dt.dayofyear
df["sp_dow"] = df["fecha_sp"].dt.dayofweek

# targets reales (solo donde exista)
df["d_start_real"] = (df["harvest_start_real"] - df["fecha_sp"]).dt.days
df["n_harvest_days_real"] = (df["harvest_end_real"] - df["harvest_start_real"]).dt.days + 1

# limpieza targets
df.loc[df["d_start_real"] < 0, "d_start_real"] = np.nan
df.loc[df["n_harvest_days_real"] <= 0, "n_harvest_days_real"] = np.nan

# caps razonables (evitar basura)
df["d_start_real"] = pd.to_numeric(df["d_start_real"], errors="coerce").clip(0, 180)
df["n_harvest_days_real"] = pd.to_numeric(df["n_harvest_days_real"], errors="coerce").clip(1, 180)

# salida: 1 fila por ciclo
out = df[
    [
        "ciclo_id",
        "bloque_base",
        "variedad_canon",
        "area",
        "tipo_sp",
        "tallos_proy",
        "fecha_sp",
        "sp_month",
        "sp_weekofyear",
        "sp_doy",
        "sp_dow",
        "harvest_start_real",
        "harvest_end_real",
        "d_start_real",
        "n_harvest_days_real",
    ]
].drop_duplicates(subset=["ciclo_id"]).copy()

out["created_at"] = pd.Timestamp.utcnow()

out_path = features_dir / "features_harvest_window_m11.parquet"
write_parquet(out, out_path)

# prints útiles
n_total = len(out)
n_start = int(out["d_start_real"].notna().sum())
n_days = int(out["n_harvest_days_real"].notna().sum())
print(f"OK -> {out_path} | rows={n_total:,}")
print(f"[COVERAGE] d_start_real notna: {n_start:,} ({n_start/max(n_total,1):.2%})")
print(f"[COVERAGE] n_harvest_days_real notna: {n_days:,} ({n_days/max(n_total,1):.2%})")

# segmentos chicos (para tu warning)
seg = (
    out[out["d_start_real"].notna() & out["n_harvest_days_real"].notna()]
    .groupby(["area", "variedad_canon", "tipo_sp"], dropna=False)
    .size()
    .reset_index(name="n")
    .sort_values("n")
)
small = seg[seg["n"] < 20].head(50)
if len(small):
    print("[WARN] segmentos con n < 20 (ML1 puede generalizar peor; pooling/regularización):")
    print(small.to_string(index=False))

if __name__ == "__main__":
    main()

```



```

-----
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

IN_UNIVERSE = Path("data/gold/universe_harvest_grid_m11.parquet")
IN_DIST_GRADO = Path("data/gold/pred_dist_grado_m11.parquet")
IN_FEATS_COSECHA = Path("data/features/features_cosecha_bloque_fecha.parquet")

OUT_PATH = Path("data/features/features_peso_tallo_grado_bloque_dia.parquet")

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    missing = [c for c in cols if c not in df.columns]
    if missing:
        raise ValueError(f"{name}: faltan columnas {missing}. Disponibles={list(df.columns)}")

def _ensure_calendar(df: pd.DataFrame) -> pd.DataFrame:
    if "dow" not in df.columns:
        df["dow"] = df["fecha"].dt.dayofweek.astype("Int64")
    if "month" not in df.columns:
        df["month"] = df["fecha"].dt.month.astype("Int64")
    if "weekofyear" not in df.columns:
        df["weekofyear"] = df["fecha"].dt.isocalendar().week.astype("Int64")
    return df

def _detect_dist_cols(dist: pd.DataFrame) -> tuple[str, str]:
    grado_cands = [c for c in ["grado", "grade"] if c in dist.columns]
    if not grado_cands:
        grado_cands = [c for c in dist.columns if "grad" in c.lower()]
    if not grado_cands:
        raise ValueError(f"No pude detectar columna de grado en pred_dist_grado_m11. Cols={list(dist.columns)}")

    col_grado = grado_cands[0]

    share_cands = [c for c in ["share_grado_m11", "share_grado", "share", "pct_grado_m11", "pct_grado"] if c in dist.columns]
    if not share_cands:
        share_cands = [c for c in dist.columns if ("share" in c.lower()) or ("pct" in c.lower())]
    if not share_cands:
        raise ValueError(f"No pude detectar columna share/pct en pred_dist_grado_m11. Cols={list(dist.columns)}")

    col_share = share_cands[0]

    return col_grado, col_share

def _print_dup_sample(df: pd.DataFrame, keys: list[str], name: str, n: int = 10) -> None:
    dup_mask = df.duplicated(subset=keys, keep=False)
    dup_n = int(dup_mask.sum())
    if dup_n == 0:
        print(f"[OK] {name}: sin duplicados por {keys}")
        return
    print(f"[WARN] {name}: duplicados por {keys} -> rows_dup={dup_n:,} (sobre {len(df):,})")
    print(df.loc[dup_mask, keys].head(n).to_string(index=False))

def main() -> None:
    created_at = pd.Timestamp.utcnow()

```

```

# -----
# Universe (fecha,bloque,variedad)
# -----
uni = read_parquet(IN_UNIVERSE).copy()
uni.columns = [str(c).strip() for c in uni.columns]
_require(uni, ["fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_m11")

uni["fecha"] = _to_date(uni["fecha"])
uni["bloque_base"] = _canon_int(uni["bloque_base"])
uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

uni_keys = ["fecha", "bloque_base", "variedad_canon"]

# Diagnóstico + dedupe duro (si hay duplicados, colapsamos)
_print_dup_sample(uni, uni_keys, "universe_harvest_grid_m11")
if uni.duplicated(subset=uni_keys).any():
    # Elegimos first en columnas no numéricas y sum/mean en numéricas relevantes si existen
    agg = {}
    for c in uni.columns:
        if c in uni_keys:
            continue
        if c in ("area", "tipo_sp", "estado", "stage", "m11_version"):
            agg[c] = "first"
        elif pd.api.types.is_numeric_dtype(uni[c]):
            # para universe, usualmente tallos_proy se suma si hubiera duplicado accidental,
            # pero si el duplicado es bug, sum puede inflar. Mejor mean para estabilidad.
            agg[c] = "mean"
        else:
            agg[c] = "first"

    uni = uni.groupby(uni_keys, dropna=False, as_index=False).agg(agg)
    print(f"[FIX] universe deduplicado -> rows={len(uni):,}")

# -----
# Dist grado (bloque,variedad,grado)
# -----
dist = read_parquet(IN_DIST_GRADO).copy()
dist.columns = [str(c).strip() for c in dist.columns]
_require(dist, ["bloque_base", "variedad_canon"], "pred_dist_grado_m11")

dist["bloque_base"] = _canon_int(dist["bloque_base"])
dist["variedad_canon"] = _canon_str(dist["variedad_canon"])

col_grado, col_share = _detect_dist_cols(dist)
dist[col_grado] = _canon_int(dist[col_grado])
dist[col_share] = pd.to_numeric(dist[col_share], errors="coerce")

dist = dist.rename(columns={col_grado: "grado", col_share: "share_grado_m11"})
dist_keys = ["bloque_base", "variedad_canon", "grado"]

_print_dup_sample(dist, dist_keys, "pred_dist_grado_m11")
if dist.duplicated(subset=dist_keys).any():
    # Si hay varias filas por mismo grado, promediamos share (y luego renormalizamos)
    dist = dist.groupby(dist_keys, dropna=False, as_index=False).agg(
        share_grado_m11=("share_grado_m11", "mean")
    )
    print(f"[FIX] dist deduplicado -> rows={len(dist):,}")

# Renormaliza share por (bloque,variedad)
sum_share = dist.groupby(["bloque_base", "variedad_canon"], dropna=False)["share_grado_m11"].transform("sum")
dist["share_grado_m11"] = np.where(sum_share.fillna(0) > 0, dist["share_grado_m11"] / sum_share, dist["share_grado_m11"])

# -----
# Expand: universe x dist
# -----
grid = uni.merge(dist, on=["bloque_base", "variedad_canon"], how="left", validate="m:m")

miss_dist = float(grid["grado"].isna().mean())
if miss_dist > 0:

```

```

        raise ValueError(
            f"Falta distribución de grado para parte del universo. miss_rate={miss_dist:.4f}. "
            "Revisa coverage de pred_dist_grado_mll por bloque_base+variedad_canon."
        )

grid["grado"] = _canon_int(grid["grado"])

keys = ["fecha", "bloque_base", "variedad_canon", "grado"]

# Diagnóstico: grid debería ser único por keys
_print_dup_sample(grid, keys, "grid (universe x dist)")
# Si aquí hay duplicados, casi seguro era universe o dist; pero ya los deduplicamos.
# Aun así, colapsamos defensivo.
if grid.duplicated(subset=keys).any():
    agg = {c: "first" for c in grid.columns if c not in keys}
    # share si se repite, promediamos
    if "share_grado_mll" in grid.columns:
        agg["share_grado_mll"] = "mean"
    grid = grid.groupby(keys, dropna=False, as_index=False).agg(agg)
    print(f"[FIX] grid deduplicado -> rows={len(grid):,}")

# -----
# Enrichment desde features_cosecha_bloque_fecha
# -----
feats = read_parquet(IN_FEATS_COSECHA).copy()
feats.columns = [str(c).strip() for c in feats.columns]
_require(feats, ["fecha", "bloque_base", "grado", "variedad_canon", "peso_tallo_baseline_g"], "features_cosecha_bloque_fecha")

feats["fecha"] = _to_date(feats["fecha"])
feats["bloque_base"] = _canon_int(feats["bloque_base"])
feats["grado"] = _canon_int(feats["grado"])
feats["variedad_canon"] = _canon_str(feats["variedad_canon"])

maybe_cols = [
    "tipo_sp",
    "area",
    "peso_tallo_baseline_g",
    "peso_tallo_real_g",
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "en_lluvia_dia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
]
take = [c for c in maybe_cols if c in feats.columns]
feats_take = feats[keys + take].copy()

_print_dup_sample(feats_take, keys, "features_cosecha_bloque_fecha (subset)")
if feats_take.duplicated(subset=keys).any():
    agg = {}
    for c in take:
        if c in ("tipo_sp", "area"):
            agg[c] = "first"
        else:
            agg[c] = "mean"
    feats_take = feats_take.groupby(keys, dropna=False, as_index=False).agg(agg)
    print(f"[FIX] feats_take deduplicado -> rows={len(feats_take):,}")

df = grid.merge(feats_take, on=keys, how="left", validate="1:1")

```

```

# -----
# Ensure cols + targets
# -----
for c in maybe_cols:
    if c not in df.columns:
        if c in ("tipo_sp", "area"):
            df[c] = "UNKNOWN"
        else:
            df[c] = np.nan

df = _ensure_calendar(df)

df["peso_tallo_baseline_g"] = pd.to_numeric(df["peso_tallo_baseline_g"], errors="coerce")
df["peso_tallo_real_g"] = pd.to_numeric(df["peso_tallo_real_g"], errors="coerce")

df["factor_peso_tallo"] = np.where(
    df["peso_tallo_baseline_g"].fillna(0) > 0,
    df["peso_tallo_real_g"] / df["peso_tallo_baseline_g"],
    np.nan,
)
df["factor_peso_tallo_clipped"] = pd.to_numeric(df["factor_peso_tallo"], errors="coerce").clip(lower=0.60,
upper=1.60)
df["delta_peso_tallo_g"] = df["peso_tallo_real_g"] - df["peso_tallo_baseline_g"]

out_cols = [
    "fecha",
    "bloque_base",
    "variedad_canon",
    "grado",
    "tipo_sp",
    "area",
    "share_grado_m11",
    "peso_tallo_baseline_g",
    "peso_tallo_real_g",
    "factor_peso_tallo",
    "factor_peso_tallo_clipped",
    "delta_peso_tallo_g",
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "en_lluvia_dia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
]
out = df[out_cols].copy()
out["created_at"] = created_at

# -----
# FINAL DEDUPE (último seguro)
# -----
_print_dup_sample(out, keys, "OUT features_peso_tallo_grado_bloque_dia (pre-final)")
if out.duplicated(subset=keys).any():
    # Colapsa determinístico: numéricos mean, categóricos first
    agg = {}
    for c in out.columns:
        if c in keys:
            continue
        if c in ("tipo_sp", "area"):
            agg[c] = "first"
        elif pd.api.types.is_numeric_dtype(out[c]):
            agg[c] = "mean"
        else:

```

```

        agg[c] = "first"

    out = out.groupby(keys, dropna=False, as_index=False).agg(agg)
    out["created_at"] = created_at # re-set
    print(f"[FIX] OUT deduplicado por keys -> rows={len(out):,}")

    out = out.sort_values(["bloque_base", "variedad_canon", "fecha", "grado"]).reset_index(drop=True)

    OUT_PATH.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(out, OUT_PATH)

    cov_real = float(out["peso_tallo_real_g"].notna().mean())
    print(f"OK -> {OUT_PATH} | rows={len(out):,}")
    print(f"[COVERAGE] peso_tallo_real_g notna: {cov_real:.4f}")
    print(f"[HORIZON] min_fecha={out['fecha'].min()} max_fecha={out['fecha'].max()}")

if __name__ == "__main__":
    main()

-----
[50/164] FILE: \src\features\build_targets_curva_beta_multiplier_dia.py
-----
from __future__ import annotations

from pathlib import Path
import json
import math
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

# =====
# Paths
# =====
FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_mll.parquet")
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")
DIM_VAR_PATH = Path("data/silver/dim_variedad_canon.parquet")

# Usamos los alpha/beta fiteados
BETA_TRAINSET_PATH = Path("data/features/trainset_curva_beta_params.parquet")

OUT_PATH = Path("data/features/trainset_curva_beta_multiplier_dia.parquet")

# =====
# Columns (match your previous intent)
# =====
NUM_COLS = [
    "day_in_harvest",
    "rel_pos",
    "n_harvest_days",
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
]

CAT_COLS = ["variedad_canon", "area", "tipo_sp"]

```

```

# =====
# Config
# =====
EPS = 1e-12
REL_CLIP = 1e-4
MIN_REAL_TOTAL_CYCLE = 50.0
INCLUDE_ZERO_REAL_DAYS = True
TARGET_CLIP = (-4.0, 4.0)

# =====
# Helpers
# =====
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _dedupe_columns(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.Index(df.columns.astype(str))
    if cols.is_unique:
        return df
    out = df.copy()
    seen: dict[str, list[int]] = {}
    for i, c in enumerate(out.columns.astype(str)):
        seen.setdefault(c, []).append(i)
    keep: dict[str, pd.Series] = {}
    for c, idxs in seen.items():
        s = out.iloc[:, idxs[0]]
        for j in idxs[1:]:
            s2 = out.iloc[:, j]
            s = s.where(s.notna(), s2)
        keep[c] = s
    ordered: list[str] = []
    for c in out.columns.astype(str):
        if c not in ordered:
            ordered.append(c)
    return pd.DataFrame({c: keep[c] for c in ordered})

def _coalesce_cols(df: pd.DataFrame, out_col: str, candidates: list[str]) -> None:
    if out_col in df.columns:
        base = df[out_col]
    else:
        base = pd.Series([pd.NA] * len(df), index=df.index)
    for c in candidates:
        if c in df.columns:
            base = base.where(base.notna(), df[c])
    df[out_col] = base

def _load_var_map(dim_var: pd.DataFrame) -> dict[str, str]:
    _require(dim_var, ["variedad_raw", "variedad_canon"], "dim_variedad_canon")
    dv = dim_var.copy()
    dv["variedad_raw"] = _canon_str(dv["variedad_raw"])
    dv["variedad_canon"] = _canon_str(dv["variedad_canon"])
    dv = dv.dropna(subset=["variedad_raw", "variedad_canon"]).drop_duplicates(subset=["variedad_raw"])
    return dict(zip(dv["variedad_raw"], dv["variedad_canon"]))

def _log_beta_pdf(x: np.ndarray, a: float, b: float) -> np.ndarray:
    logB = math.lgamma(a) + math.lgamma(b) - math.lgamma(a + b)
    return (a - 1.0) * np.log(x) + (b - 1.0) * np.log(1.0 - x) - logB

def _beta_share(rel: np.ndarray, a: float, b: float) -> np.ndarray:
    rel = np.clip(rel, REL_CLIP, 1.0 - REL_CLIP)
    lp = _log_beta_pdf(rel, float(a), float(b))

```

```

lp = lp - np.max(lp)
p = np.exp(lp)
s = float(np.sum(p))
return p / s if s > 0 else np.zeros_like(p)

# =====
# Main
# =====
def main() -> None:
    created_at = pd.Timestamp.utcnow()

    for p in [FEATURES_PATH, UNIVERSE_PATH, PROG_PATH, DIM_VAR_PATH, BETA_TRAINSET_PATH]:
        if not p.exists():
            raise FileNotFoundError(f"No existe: {p}")

    feat = _dedupe_columns(read_parquet(FEATURES_PATH).copy())
    uni = read_parquet(UNIVERSE_PATH).copy()
    prog = read_parquet(PROG_PATH).copy()
    dim_var = read_parquet(DIM_VAR_PATH).copy()
    beta_ts = read_parquet(BETA_TRAINSET_PATH).copy()

    var_map = _load_var_map(dim_var)

    _require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe")
    _require(feat, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "features")
    _require(prog, ["ciclo_id", "fecha", "bloque_base"], "prog")
    _require(beta_ts, ["ciclo_id", "bloque_base", "variedad_canon", "alpha", "beta"], "beta_trainset")

    # Canon
    for df in (feat, uni, prog, beta_ts):
        df["ciclo_id"] = df["ciclo_id"].astype(str)
        if "fecha" in df.columns:
            df["fecha"] = _to_date(df["fecha"])
        df["bloque_base"] = _canon_int(df["bloque_base"])

    feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])
    beta_ts["variedad_canon"] = _canon_str(beta_ts["variedad_canon"])

    # Canon prog variedad
    if "variedad" in prog.columns:
        prog["variedad_raw"] = _canon_str(prog["variedad"])
        prog["variedad_canon"] = prog["variedad_raw"].map(var_map).fillna(prog["variedad_raw"])
    else:
        prog["variedad_canon"] = _canon_str(prog["variedad_canon"])

    prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce").fillna(0.0)

    # Coalesce harvest cols
    _coalesce_cols(feat, "day_in_harvest", ["day_in_harvest", "day_in_harvest_pred", "day_in_harvest_pred_fina
1"])
    _coalesce_cols(feat, "rel_pos", ["rel_pos", "rel_pos_pred", "rel_pos_pred_final"])
    _coalesce_cols(feat, "n_harvest_days", ["n_harvest_days", "n_harvest_days_pred", "n_harvest_days_pred_fina
1"])

    for c in ["day_in_harvest", "rel_pos", "n_harvest_days"]:
        feat[c] = pd.to_numeric(feat[c], errors="coerce")

    # Ensure feature cols exist
    for c in NUM_COLS:
        if c not in feat.columns:
            feat[c] = np.nan
    for c in CAT_COLS:
        if c not in feat.columns:
            feat[c] = "UNKNOWN"

    # Universe panel
    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    uni_k = uni[key].drop_duplicates()

    feat_take = key + NUM_COLS + CAT_COLS
    feat_take = list(dict.fromkeys(feat_take))

```

```

panel = (
    uni_k
    .merge(feats[feat_take], on=key, how="left")
    .merge(prog[key + ["tallos_real_dia"]].drop_duplicates(subset=key), on=key, how="left")
)
panel["tallos_real_dia"] = pd.to_numeric(panel["tallos_real_dia"], errors="coerce").fillna(0.0)

# Harvest mask
dih = pd.to_numeric(panel["day_in_harvest"], errors="coerce")
nh = pd.to_numeric(panel["n_harvest_days"], errors="coerce")
is_h = dih.notna() & nh.notna() & (dih >= 1) & (nh >= 1) & (dih <= nh)
panel = panel[is_h].copy()

# rel_pos fallback if needed
if panel["rel_pos"].isna().any():
    di2 = pd.to_numeric(panel["day_in_harvest"], errors="coerce").astype(float)
    nh2 = pd.to_numeric(panel["n_harvest_days"], errors="coerce").astype(float)
    panel["rel_pos"] = np.clip((di2 - 0.5) / (nh2 + EPS), REL_CLIP, 1.0 - REL_CLIP)

# Canon cats
panel["variedad_canon"] = _canon_str(panel["variedad_canon"])
panel["area"] = _canon_str(panel["area"].fillna("UNKNOWN"))
panel["tipo_sp"] = _canon_str(panel["tipo_sp"].fillna("UNKNOWN"))

# Filtrar ciclos con señal real
grp = ["ciclo_id", "bloque_base", "variedad_canon"]
real_total_grp = panel.groupby(grp, dropna=False)["tallos_real_dia"].transform("sum").astype(float)
panel = panel[real_total_grp >= MIN_REAL_TOTAL_CYCLE].copy()
if len(panel) == 0:
    raise ValueError("No hay ciclos con real_total suficiente para entrenar multiplicador. Baja MIN_REAL_T
OTAL_CYCLE o revisa PROG.")

# Share real por ciclo
tot = panel.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform("sum").astype(float)
panel["share_real"] = np.where(tot > 0, panel["tallos_real_dia"].astype(float) / tot, 0.0)

# Adjuntar alpha/beta fiteados (por grupo)
beta_take = beta_ts[["ciclo_id", "bloque_base", "variedad_canon", "alpha", "beta"]].drop_duplicates()
panel = panel.merge(beta_take, on=["ciclo_id", "bloque_base", "variedad_canon"], how="left")

panel = panel[panel["alpha"].notna() & panel["beta"].notna()].copy()
if len(panel) == 0:
    raise ValueError("No quedaron filas con alpha/beta (beta_trainset no alinea).")

# Share beta por ciclo
panel = panel.sort_values(["ciclo_id", "fecha"], kind="mergesort").reset_index(drop=True)
share_beta = np.zeros(len(panel), dtype=float)

for cid, idx in panel.groupby("ciclo_id", dropna=False).indices.items():
    ii = np.array(list(idx), dtype=int)
    rel = panel.loc[ii, "rel_pos"].to_numpy(dtype=float)
    a = float(pd.to_numeric(panel.loc[ii, "alpha"], errors="coerce").dropna().iloc[0])
    b = float(pd.to_numeric(panel.loc[ii, "beta"], errors="coerce").dropna().iloc[0])
    a = max(a, 1.05)
    b = max(b, 1.05)
    sh = _beta_share(rel, a, b)
    share_beta[ii] = sh

panel["share_beta"] = share_beta

# Target log-mult
mult = panel["share_real"].to_numpy(dtype=float) / (panel["share_beta"].to_numpy(dtype=float) + EPS)
y = np.log(np.clip(mult, EPS, 1e6))
y = np.clip(y, TARGET_CLIP[0], TARGET_CLIP[1])
panel["y_log_mult"] = y

if not INCLUDE_ZERO_REAL_DAYS:
    panel = panel[panel["tallos_real_dia"] > 0].copy()

# Keep only needed columns (DEDUP!)
keep = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"] + NUM_COLS + CAT_COLS + [
    "tallos_real_dia", "share_real", "share_beta", "alpha", "beta", "y_log_mult"
]

```



```

]
keep = [c for c in keep if c in panel.columns]
keep = list(dict.fromkeys(keep)) # <-- FIX: remove duplicates preserving order

out = panel[keep].copy()
out["created_at"] = created_at

# Hard check before write
cols = pd.Index(out.columns.astype(str))
if not cols.is_unique:
    dup = cols[cols.duplicated()].unique().tolist()
    raise ValueError(f"[FATAL] columnas duplicadas antes de escribir: {dup}")

write_parquet(out, OUT_PATH)
print(f"OK -> {OUT_PATH} | rows={len(out):,} | cycles={out['ciclo_id'].nunique():,}")

if __name__ == "__main__":
    main()

-----
[51/164] FILE: \src\features\build_targets_curva_beta_params.py
-----
from __future__ import annotations

from pathlib import Path
import json
import math
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

# =====
# Paths
# =====
FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_mll.parquet")
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet") # real tallos dia
DIM_VAR_PATH = Path("data/silver/dim_variedad_canon.parquet") # <-- NUEVO (canon prog -> canon)

OUT_PATH = Path("data/features/trainset_curva_beta_params.parquet")
DBG_PANEL_PATH = Path("data/features/_debug_panel_beta_params.parquet")

# =====
# Config
# =====
EPS = 1e-12
REL_CLIP = 1e-4
MIN_REAL_TOTAL = 10.0 # <-- BAJADO para no matar el entrenamiento; ajusta luego con evidencia
GRID_K = 7
GRID_STEP = 0.12

# =====
# Helpers
# =====
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _dedupe_columns(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.Index(df.columns.astype(str))
    if cols.is_unique:
        return df
    out = df.copy()
    seen: dict[str, list[int]] = {}
    for i, c in enumerate(out.columns.astype(str)):

```

```

        seen.setdefault(c, []).append(i)
keep: dict[str, pd.Series] = {}
for c, idxs in seen.items():
    s = out.iloc[:, idxs[0]]
    for j in idxs[1:]:
        s2 = out.iloc[:, j]
        s = s.where(s.notna(), s2)
    keep[c] = s
ordered: list[str] = []
for c in out.columns.astype(str):
    if c not in ordered:
        ordered.append(c)
return pd.DataFrame({c: keep[c] for c in ordered})

def _coalesce_cols(df: pd.DataFrame, out_col: str, candidates: list[str]) -> None:
    if out_col in df.columns:
        base = df[out_col]
    else:
        base = pd.Series([pd.NA] * len(df), index=df.index)
    for c in candidates:
        if c in df.columns:
            base = base.where(base.notna(), df[c])
    df[out_col] = base

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _load_var_map(dim_var: pd.DataFrame) -> dict[str, str]:
    _require(dim_var, ["variedad_raw", "variedad_canon"], "dim_variedad_canon")
    dv = dim_var.copy()
    dv["variedad_raw"] = _canon_str(dv["variedad_raw"])
    dv["variedad_canon"] = _canon_str(dv["variedad_canon"])
    dv = dv.dropna(subset=["variedad_raw", "variedad_canon"]).drop_duplicates(subset=["variedad_raw"])
    return dict(zip(dv["variedad_raw"], dv["variedad_canon"]))

def _log_beta_pdf(x: np.ndarray, a: float, b: float) -> np.ndarray:
    logB = math.lgamma(a) + math.lgamma(b) - math.lgamma(a + b)
    return (a - 1.0) * np.log(x) + (b - 1.0) * np.log(1.0 - x) - logB

def _beta_share_on_grid(rel: np.ndarray, a: float, b: float) -> np.ndarray:
    rel = np.clip(rel, REL_CLIP, 1.0 - REL_CLIP)
    lp = _log_beta_pdf(rel, a, b)
    lp = lp - np.max(lp)
    p = np.exp(lp)
    s = float(np.sum(p))
    if s <= 0:
        return np.zeros_like(p)
    return p / s

def _fit_beta_params_discrete(rel: np.ndarray, share_real: np.ndarray) -> tuple[float, float, dict]:
    w = np.clip(share_real.astype(float), 0.0, None)
    sw = float(np.sum(w))
    if sw <= 0:
        return (2.0, 2.0, {"fit_ok": False, "reason": "zero_share"})

    w = w / sw
    x = np.clip(rel.astype(float), REL_CLIP, 1.0 - REL_CLIP)

    mu = float(np.sum(w * x))
    var = float(np.sum(w * (x - mu) ** 2))
    var = max(var, 1e-6)

    k = mu * (1.0 - mu) / var - 1.0
    if not np.isfinite(k) or k <= 0:
        a0, b0 = 2.0, 2.0
    else:
        a0 = mu * k
        b0 = (1.0 - mu) * k

    a0 = float(max(a0, 1.05))

```

```

b0 = float(max(b0, 1.05))

def nll(a: float, b: float) -> float:
    pb = _beta_share_on_grid(x, a, b)
    pb = np.clip(pb, EPS, 1.0)
    return float(-np.sum(w * np.log(pb)))

best_a, best_b = a0, b0
best = nll(best_a, best_b)

la0, lb0 = math.log(best_a), math.log(best_b)
for ia in range(-GRID_K, GRID_K + 1):
    for ib in range(-GRID_K, GRID_K + 1):
        a = math.exp(la0 + ia * GRID_STEP)
        b = math.exp(lb0 + ib * GRID_STEP)
        if a <= 1.0 or b <= 1.0:
            continue
        val = nll(a, b)
        if val < best:
            best = val
            best_a, best_b = a, b

info = {"fit_ok": True, "mu": mu, "var": var, "a0_mom": a0, "b0_mom": b0, "a": float(best_a), "b": float(best_b), "nll": float(best)}
return float(best_a), float(best_b), info

def _agg_cycle_features(feats_h: pd.DataFrame) -> pd.Series:
    out = {}
    for c in [
        "rainfall_mm_dia",
        "horas_lluvia",
        "temp_avg_dia",
        "solar_energy_j_m2_dia",
        "wind_speed_avg_dia",
        "wind_run_dia",
        "gdc_dia",
    ]:
        if c in feats_h.columns:
            v = pd.to_numeric(feats_h[c], errors="coerce")
            out[f"{c}__mean"] = float(v.mean(skipna=True)) if len(v) else np.nan
            out[f"{c}__sum"] = float(v.sum(skipna=True)) if len(v) else np.nan

    for c in ["dias_desde_sp", "gdc_acum_desde_sp"]:
        if c in feats_h.columns:
            v = pd.to_numeric(feats_h[c], errors="coerce")
            out[f"{c}__last"] = float(v.dropna().iloc[-1]) if v.notna().any() else np.nan
            out[f"{c}__mean"] = float(v.mean(skipna=True)) if len(v) else np.nan

    if "n_harvest_days" in feats_h.columns:
        v = pd.to_numeric(feats_h["n_harvest_days"], errors="coerce")
        out["n_harvest_days"] = float(v.max(skipna=True)) if v.notna().any() else np.nan

    if "tallos_proy" in feats_h.columns:
        v = pd.to_numeric(feats_h["tallos_proy"], errors="coerce")
        out["tallos_proy"] = float(v.max(skipna=True)) if v.notna().any() else 0.0

    return pd.Series(out)

# =====
# Main
# =====
def main() -> None:
    created_at = pd.Timestamp.utcnow()

    for p in [FEATURES_PATH, UNIVERSE_PATH, PROG_PATH, DIM_VAR_PATH]:
        if not p.exists():
            raise FileNotFoundError(f"No existe: {p}")

    feat = _dedupe_columns(read_parquet(FEATURES_PATH).copy())
    uni = read_parquet(UNIVERSE_PATH).copy()
    prog = read_parquet(PROG_PATH).copy()
    dim_var = read_parquet(DIM_VAR_PATH).copy()

```

```

var_map = _load_var_map(dim_var)

# Regs mínimos
_require(feat, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "features")
_require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe")
_require(prog, ["ciclo_id", "fecha", "bloque_base"], "prog")
if "tallos_real_dia" not in prog.columns:
    raise ValueError("prog: falta tallos_real_dia")
if ("variedad" not in prog.columns) and ("variedad_canon" not in prog.columns):
    raise ValueError("prog: falta variedad o variedad_canon")

# Canon ids
for df in (feat, uni, prog):
    df["ciclo_id"] = df["ciclo_id"].astype(str)
    df["fecha"] = _to_date(df["fecha"])
    df["bloque_base"] = _canon_int(df["bloque_base"])

feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

# Canon prog variedad con dim_var (esta era la causa típica)
if "variedad" in prog.columns:
    prog["variedad_raw"] = _canon_str(prog["variedad"])
    prog["variedad_canon"] = prog["variedad_raw"].map(var_map).fillna(prog["variedad_raw"])
else:
    prog["variedad_canon"] = _canon_str(prog["variedad_canon"])

prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce").fillna(0.0)

# Harvest cols en features
_coalesce_cols(feat, "day_in_harvest", ["day_in_harvest", "day_in_harvest_pred", "day_in_harvest_pred_fina
1"])
_coalesce_cols(feat, "n_harvest_days", ["n_harvest_days", "n_harvest_days_pred", "n_harvest_days_pred_fina
1"])
feat["day_in_harvest"] = pd.to_numeric(feat["day_in_harvest"], errors="coerce")
feat["n_harvest_days"] = pd.to_numeric(feat["n_harvest_days"], errors="coerce")

# Universe keys
key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
uni_k = uni[key].drop_duplicates()

# Panel universe + features + prog
feat_cols = [
    "day_in_harvest", "n_harvest_days", "tallos_proy", "area", "tipo_sp",
    "rainfall_mm_dia", "horas_lluvia", "temp_avg_dia", "solar_energy_j_m2_dia",
    "wind_speed_avg_dia", "wind_run_dia", "gdc_dia", "dias_desde_sp", "gdc_acum_desde_sp",
]
feat_take = [c for c in (key + feat_cols) if c in feat.columns]

panel = (
    uni_k
    .merge(feat[feat_take], on=key, how="left")
    .merge(prog[key + ["tallos_real_dia"]].drop_duplicates(subset=key), on=key, how="left")
)

# Diagnóstico join prog
panel["tallos_real_dia"] = pd.to_numeric(panel["tallos_real_dia"], errors="coerce").fillna(0.0)
n_total = len(panel)
n_pos = int((panel["tallos_real_dia"] > 0).sum())
s_tot = float(panel["tallos_real_dia"].sum())
print(f"[DBG] panel rows={n_total:,} | rows con tallos_real_dia>0: {n_pos:,} ({(n_pos/max(n_total,1))*100:
.2f}%) | sum_real={s_tot:.1f}")

# Diagnóstico harvest mask
dih = pd.to_numeric(panel["day_in_harvest"], errors="coerce")
nh = pd.to_numeric(panel["n_harvest_days"], errors="coerce")
is_h = dii.notna() & nh.notna() & (dih >= 1) & (nh >= 1) & (dih <= nh)
print(f"[DBG] is_harvest rows={int(is_h.sum()):,} ({(float(is_h.mean())*100):.2f}%) | day_in_harvest notna
={(dih.notna().mean()*100):.2f}% | n_harvest_days notna={(nh.notna().mean()*100):.2f}%")

panel = panel[is_h].copy()
if len(panel) == 0:

```

```

write_parquet(panel, DBG_PANEL_PATH)
raise ValueError("No hay filas harvest (day_in_harvest/n_harvest_days). Revisa features_curva y universe.")

# Totales reales por grupo
grp = ["ciclo_id", "bloque_base", "variedad_canon"]
tot_real = panel.groupby(grp, dropna=False)["tallos_real_dia"].transform("sum").astype(float)
n_groups_all = panel[grp].drop_duplicates().shape[0]
n_groups_pos = panel.loc[tot_real > 0, grp].drop_duplicates().shape[0]
print(f"[DBG] grupos harvest={n_groups_all:,} | grupos con real_total>0: {n_groups_pos:,}")

panel = panel[tot_real >= MIN_REAL_TOTAL].copy()
n_groups_train = panel[grp].drop_duplicates().shape[0]
print(f"[DBG] MIN_REAL_TOTAL={MIN_REAL_TOTAL} => grupos para train: {n_groups_train:,}")

if len(panel) == 0:
    # guarda debug para inspección
    write_parquet(panel, DBG_PANEL_PATH)
    raise ValueError("No hay suficientes ciclos con señal real para entrenar beta params. Revisa join PROG
/universe o MIN_REAL_TOTAL.")

# Rel pos
dih = pd.to_numeric(panel["day_in_harvest"], errors="coerce").astype(float)
nh = pd.to_numeric(panel["n_harvest_days"], errors="coerce").astype(float)
panel["rel_pos"] = np.clip((dih - 0.5) / (nh + EPS), REL_CLIP, 1.0 - REL_CLIP)

panel = panel.sort_values(grp + ["day_in_harvest", "fecha"], kind="mergesort").reset_index(drop=True)

rows = []
fit_meta = {}

for (cid, bb, var), df in panel.groupby(grp, dropna=False):
    y = df["tallos_real_dia"].to_numpy(dtype=float)
    s = float(np.sum(y))
    if s <= 0:
        continue
    share_real = y / s
    rel = df["rel_pos"].to_numpy(dtype=float)

    a, b, info = _fit_beta_params_discrete(rel, share_real)
    f = _agg_cycle_features(df)

    row = {
        "ciclo_id": str(cid),
        "bloque_base": int(bb) if pd.notna(bb) else pd.NA,
        "variedad_canon": str(var),
        "area": str(df["area"].iloc[0]) if "area" in df.columns else "UNKNOWN",
        "tipo_sp": str(df["tipo_sp"].iloc[0]) if "tipo_sp" in df.columns else "UNKNOWN",
        "alpha": float(a),
        "beta": float(b),
        "real_total": float(s),
        "n_harvest_days": float(pd.to_numeric(df["n_harvest_days"], errors="coerce").max()),
    }
    row.update({k: (float(v) if pd.notna(v) else np.nan) for k, v in f.to_dict().items()})
    rows.append(row)
    fit_meta[f"{cid}|{bb}|{var}"] = info

out = pd.DataFrame(rows)
if len(out) == 0:
    write_parquet(panel, DBG_PANEL_PATH)
    raise ValueError("No se pudieron ajustar parámetros beta en ningún grupo. Revisa rel_pos/day_in_harvest/n_harvest_days.")

out["created_at"] = created_at
write_parquet(out, OUT_PATH)

meta_path = OUT_PATH.with_suffix(".fitmeta.json")
with open(meta_path, "w", encoding="utf-8") as f:
    json.dump({"created_at": str(created_at), "n_rows": int(len(out)), "MIN_REAL_TOTAL": MIN_REAL_TOTAL, "
fit_info": fit_meta}, f, indent=2, ensure_ascii=False)

print(f"OK -> {OUT_PATH} | rows={len(out):,} | grupos={out[['ciclo_id', 'bloque_base', 'variedad_canon']].dr

```

```

op_duplicates().shape[0:],})

if __name__ == "__main__":
    main()

-----
[52/164] FILE: \src\gold\apply_desp_ajuste_poscosecha_ml1_on_ml2_hidr.py
-----

from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"

IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_hidr_grado_dia_bloque_destino.parquet"

REG_DESP = ROOT / "models_registry" / "ml1" / "desp_poscosecha"
REG_AJ = ROOT / "models_registry" / "ml1" / "ajuste_poscosecha"

OUT_PATH = GOLD / "pred_poscosecha_ml2_full_grado_dia_bloque_destino.parquet"

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino"]

def _latest_version_dir(root: Path) -> Path:
    if not root.exists():
        raise FileNotFoundError(f"No existe {root}")
    dirs = [p for p in root.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {root}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def _load_meta_and_model(root: Path, version: str | None):
    ver_dir = _latest_version_dir(root) if version is None else (root / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe versión: {ver_dir}")
    with open(ver_dir / "metrics.json", "r", encoding="utf-8") as f:
        meta = json.load(f)
    model = load(ver_dir / "model.joblib")
    return ver_dir.name, meta, model

def main(version_desp: str | None = None, version_aj: str | None = None) -> None:
    v_desp, meta_desp, model_desp = _load_meta_and_model(REG_DESP, version_desp)
    v_aj, meta_aj, model_aj = _load_meta_and_model(REG_AJ, version_aj)

    df = read_parquet(IN_UNIVERSE).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino", "fecha_post_pred_ml1", "factor_hidr_ml1", "cajas_split_grado_dia"}
    miss = need - set(df.columns)
    if miss:
        raise KeyError(f"Universe Hidr ML2 sin columnas: {sorted(miss)}")

```

```

df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
df["fecha_post_pred_ml1"] = pd.to_datetime(df["fecha_post_pred_ml1"], errors="coerce").dt.normalize()
df["destino"] = df["destino"].astype(str).str.upper().str.strip()
df["grado"] = pd.to_numeric(df["grado"], errors="coerce").astype("Int64")

# features calendario por fecha_post_pred_ml1 (día real operativo de proceso)
df["dow"] = df["fecha_post_pred_ml1"].dt.dayofweek.astype("Int64")
df["month"] = df["fecha_post_pred_ml1"].dt.month.astype("Int64")
df["weekofyear"] = df["fecha_post_pred_ml1"].dt.isocalendar().week.astype("Int64")

for c in NUM_COLS:
    if c not in df.columns:
        df[c] = np.nan
for c in CAT_COLS:
    if c not in df.columns:
        df[c] = "UNKNOWN"

X = df[NUM_COLS + CAT_COLS]

# DESP
lo_d, hi_d = meta_desp.get("clip_range_apply", [0.05, 1.00])
df["factor_desp_ml1_raw"] = pd.to_numeric(pd.Series(model_desp.predict(X)), errors="coerce")
df["factor_desp_ml1"] = df["factor_desp_ml1_raw"].clip(lower=float(lo_d), upper=float(hi_d))

# AJUSTE (ojo: en tu fórmula se usa DIVISIÓN / ajuste_ml1)
lo_a, hi_a = meta_aj.get("clip_range_apply", [0.80, 1.50])
df["ajuste_ml1_raw"] = pd.to_numeric(pd.Series(model_aj.predict(X)), errors="coerce")
df["ajuste_ml1"] = df["ajuste_ml1_raw"].clip(lower=float(lo_a), upper=float(hi_a))

df["ml1_desp_version"] = v_desp
df["ml1_ajuste_version"] = v_aj
df["created_at"] = pd.Timestamp.utcnow()

# cálculo final poscosecha ?ML1? pero alimentado por ML2-campo
df["cajas_postcosecha_ml1"] = (
    pd.to_numeric(df["cajas_split_grado_dia"], errors="coerce").fillna(0.0)
    * pd.to_numeric(df["factor_hidr_ml1"], errors="coerce").fillna(1.0)
    * pd.to_numeric(df["factor_desp_ml1"], errors="coerce").fillna(1.0)
    / pd.to_numeric(df["ajuste_ml1"], errors="coerce").replace(0, np.nan).fillna(1.0)
)

write_parquet(df, OUT_PATH)
print(f"[OK] Wrote: {OUT_PATH} rows={len(df):,} desp={v_desp} ajuste={v_aj}")

```

```

if __name__ == "__main__":
    main(version_desp=None, version_aj=None)

```

```

-----
[53/164] FILE: \src\gold\apply_dh_poscosecha_ml1_on_ml2_seed.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

```

```

from src.common.io import read_parquet, write_parquet

```

```

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

```

```

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"

```

```

IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_seed_grado_dia_bloque_destino.parquet"

```

```

REGISTRY_ROOT = ROOT / "models_registry" / "m11" / "dh_poscosecha"
OUT_PATH = GOLD / "pred_poscosecha_m12_dh_grado_dia_bloque_destino.parquet"

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino", "grado"]

def _latest_version_dir(root: Path) -> Path:
    if not root.exists():
        raise FileNotFoundError(f"No existe {root}")
    dirs = [p for p in root.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {root}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def main(version: str | None = None) -> None:
    ver_dir = _latest_version_dir(REGISTRY_ROOT) if version is None else (REGISTRY_ROOT / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe versión: {ver_dir}")

    with open(ver_dir / "metrics.json", "r", encoding="utf-8") as f:
        meta = json.load(f)
    model = load(ver_dir / "model.joblib")

    df = read_parquet(IN_UNIVERSE).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino", "cajas_split_grado_dia"}
    miss = need - set(df.columns)
    if miss:
        raise KeyError(f"Seed ML2 sin columnas: {sorted(miss)}")

    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    df["grado"] = pd.to_numeric(df["grado"], errors="coerce").astype("Int64")
    df["destino"] = df["destino"].astype(str).str.upper().str.strip()

    # features calendario sobre fecha (cosecha)
    df["dow"] = df["fecha"].dt.dayofweek.astype("Int64")
    df["month"] = df["fecha"].dt.month.astype("Int64")
    df["weekofyear"] = df["fecha"].dt.isocalendar().week.astype("Int64")

    for c in NUM_COLS:
        if c not in df.columns:
            df[c] = np.nan
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"

    X = df[NUM_COLS + CAT_COLS]
    pred = model.predict(X)

    # clip + round
    clip_lo, clip_hi = meta.get("clip_range_apply", [0, 30])
    dh = pd.to_numeric(pd.Series(pred), errors="coerce").clip(lower=float(clip_lo), upper=float(clip_hi))
    df["dh_dias_m11_raw"] = dh
    df["dh_dias_m11"] = np rint(df["dh_dias_m11_raw"]).astype("Int64")

    df["fecha_post_pred_m11"] = df["fecha"] + pd.to_timedelta(df["dh_dias_m11"].fillna(0).astype(int), unit="D")

    df["m11_dh_version"] = ver_dir.name
    df["created_at"] = pd.Timestamp.utcnow()

    write_parquet(df, OUT_PATH)
    print(f"[OK] Wrote: {OUT_PATH} rows={len(df):,} version={ver_dir.name}")

if __name__ == "__main__":
    main(version=None)

```



```
-----  
[54/164] FILE: \src\gold\apply_hidr_poscosecha_ml1_on_ml2_dh.py  
-----
```

```
from __future__ import annotations
```

```
from pathlib import Path  
import json  
import numpy as np  
import pandas as pd  
from joblib import load
```

```
from src.common.io import read_parquet, write_parquet
```

```
def _project_root() -> Path:  
    return Path(__file__).resolve().parents[2]
```

```
ROOT = _project_root()  
DATA = ROOT / "data"  
GOLD = DATA / "gold"
```

```
IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_dh_grado_dia_bloque_destino.parquet"  
REGISTRY_ROOT = ROOT / "models_registry" / "ml1" / "hidr_poscosecha"  
OUT_PATH = GOLD / "pred_poscosecha_ml2_hidr_grado_dia_bloque_destino.parquet"
```

```
NUM_COLS = ["dow", "month", "weekofyear"]  
CAT_COLS = ["destino", "grado"]
```

```
def _latest_version_dir(root: Path) -> Path:  
    if not root.exists():  
        raise FileNotFoundError(f"No existe {root}")  
    dirs = [p for p in root.iterdir() if p.is_dir()]  
    if not dirs:  
        raise FileNotFoundError(f"No hay versiones dentro de {root}")  
    return sorted(dirs, key=lambda p: p.name)[-1]
```

```
def main(version: str | None = None) -> None:  
    ver_dir = _latest_version_dir(REGISTRY_ROOT) if version is None else (REGISTRY_ROOT / version)  
    if not ver_dir.exists():  
        raise FileNotFoundError(f"No existe versión: {ver_dir}")  
  
    with open(ver_dir / "metrics.json", "r", encoding="utf-8") as f:  
        meta = json.load(f)  
        model = load(ver_dir / "model.joblib")  
  
    df = read_parquet(IN_UNIVERSE).copy()  
    df.columns = [str(c).strip() for c in df.columns]  
  
    need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino", "dh_dias_ml1", "fecha_post_pred_ml1"}  
    miss = need - set(df.columns)  
    if miss:  
        raise KeyError(f"Universe DH ML2 sin columnas: {sorted(miss)}")  
  
    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()  
    df["grado"] = pd.to_numeric(df["grado"], errors="coerce").astype("Int64")  
    df["destino"] = df["destino"].astype(str).str.upper().str.strip()  
  
    # features calendario sobre fecha (cosecha)  
    df["dow"] = df["fecha"].dt.dayofweek.astype("Int64")  
    df["month"] = df["fecha"].dt.month.astype("Int64")  
    df["weekofyear"] = df["fecha"].dt.isocalendar().week.astype("Int64")  
  
    for c in NUM_COLS:  
        if c not in df.columns:  
            df[c] = np.nan  
    for c in CAT_COLS:  
        if c not in df.columns:  
            df[c] = "UNKNOWN"
```

```

X = df[NUM_COLS + CAT_COLS]
pred = model.predict(X)

lo, hi = meta.get("clip_range_apply", [0.80, 3.00])
df["factor_hidr_mll_raw"] = pd.to_numeric(pd.Series(pred), errors="coerce")
df["factor_hidr_mll"] = df["factor_hidr_mll_raw"].clip(lower=float(lo), upper=float(hi))

df["mll_hidr_version"] = ver_dir.name
df["created_at"] = pd.Timestamp.utcnow()

write_parquet(df, OUT_PATH)
print(f"[OK] Wrote: {OUT_PATH} rows={len(df):,} version={ver_dir.name} clip={lo}->{hi}")

if __name__ == "__main__":
    main(version=None)

-----
[55/164] FILE: \src\gold\build_ds_ajuste_poscosecha_ml2_v1.py
-----

from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
SILVER = DATA / "silver"

IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_desp_grado_dia_bloque_destino_final.parquet"
IN_REAL_MA = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"

OUT_DS = GOLD / "ml2_datasets" / "ds_ajuste_poscosecha_ml2_v1.parquet"

# -----
# TZ helpers (evita tz-aware vs tz-naive)
# -----
def _to_naive_utc(ts: pd.Timestamp) -> pd.Timestamp:
    """Convierte a UTC y deja tz-naive."""
    if ts.tzinfo is None:
        return ts
    return ts.tz_convert("UTC").tz_localize(None)

def _as_of_date_naive() -> pd.Timestamp:
    """
    Regla: as_of = hoy-1.
    IMPORTANTE: tz-naive para comparar con columnas datetime64[ns] (naive).
    """
    # utcnow() suele ser tz-naive, pero en algunos entornos puede salir tz-aware; blindamos.
    t = pd.Timestamp.utcnow()
    t = _to_naive_utc(t)
    return (t.normalize() - pd.Timedelta(days=1))

def _to_date_naive(s: pd.Series) -> pd.Series:
    """
    Convierte a datetime y deja tz-naive normalizado.
    """
    dt = pd.to_datetime(s, errors="coerce")

```

```

# Si viene tz-aware, pásalo a UTC y quita tz
try:
    if getattr(dt.dt, "tz", None) is not None:
        dt = dt.dt.tz_convert("UTC").dt.tz_localize(None)
except Exception:
    # si no soporta .dt.tz (por tipos raros), lo dejamos como está
    pass
return dt.dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _resolve_fecha_post_pred(df: pd.DataFrame) -> str:
    for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_m11", "fecha_post_pred"]:
        if c in df.columns:
            return c
    raise KeyError(
        "No encuentro columna de fecha_post_pred (esperaba fecha_post_pred_final/fecha_post_pred_m11/...)."
    )

def _resolve_ajuste_m11(df: pd.DataFrame) -> str:
    for c in ["factor_ajuste_m11", "ajuste_m11", "factor_ajuste_seed", "factor_ajuste"]:
        if c in df.columns:
            return c
    raise KeyError(
        "No encuentro columna de ajuste ML1 (esperaba ajuste_m11 / factor_ajuste_m11 / factor_ajuste_seed)."
    )

def _weight_series(df: pd.DataFrame) -> pd.Series:
    for c in ["tallos_w", "tallos", "tallos_total_m12", "tallos_total"]:
        if c in df.columns:
            return pd.to_numeric(df[c], errors="coerce").fillna(0.0)
    return pd.Series(1.0, index=df.index, dtype="float64")

def main() -> None:
    as_of = _as_of_date_naive()

    uni = read_parquet(IN_UNIVERSE).copy()
    uni.columns = [str(c).strip() for c in uni.columns]

    need = {"fecha", "destino"}
    miss = need - set(uni.columns)
    if miss:
        raise ValueError(f"Universe sin columnas: {sorted(miss)}")

    # ? tz-naive
    uni["fecha"] = _to_date_naive(uni["fecha"])
    uni["destino"] = _canon_str(uni["destino"])

    if "grado" in uni.columns:
        uni["grado"] = pd.to_numeric(uni["grado"], errors="coerce").astype("Int64")
    else:
        uni["grado"] = pd.Series(pd.NA, index=uni.index, dtype="Int64")

    fecha_post_col = _resolve_fecha_post_pred(uni)
    uni[fecha_post_col] = _to_date_naive(uni[fecha_post_col])

    ajuste_m11_col = _resolve_ajuste_m11(uni)
    uni[ajuste_m11_col] = pd.to_numeric(uni[ajuste_m11_col], errors="coerce")

    # ? comparación safe: ambos tz-naive
    uni = uni[uni["fecha"].notna() & (uni["fecha"] <= as_of)].copy()

    # ---- Real por fecha_post + destino (NO por grado) ----
    real = read_parquet(IN_REAL_MA).copy()
    real.columns = [str(c).strip() for c in real.columns]

```

```

if "fecha_post" not in real.columns or "destino" not in real.columns:
    raise ValueError("dim_mermas_ajuste_fecha_post_destino debe traer fecha_post y destino.")

if "factor_ajuste" in real.columns:
    real_factor_col = "factor_ajuste"
elif "ajuste" in real.columns:
    real_factor_col = "ajuste"
else:
    raise ValueError("dim_mermas_ajuste_fecha_post_destino no trae factor_ajuste ni ajuste.")

real["fecha_post"] = _to_date_naive(real["fecha_post"])
real["destino"] = _canon_str(real["destino"])
real[real_factor_col] = pd.to_numeric(real[real_factor_col], errors="coerce")

real2 = (
    real.groupby(["fecha_post", "destino"], dropna=False, as_index=False)
        .agg(factor_ajuste_real=(real_factor_col, "median"))
)

df = uni.merge(
    real2.rename(columns={"fecha_post": "fecha_post_key"}),
    left_on=[fecha_post_col, "destino"],
    right_on=["fecha_post_key", "destino"],
    how="left",
)

# target: log(real / m11) (si es >0)
m11 = pd.to_numeric(df[ajuste_m11_col], errors="coerce")
realv = pd.to_numeric(df["factor_ajuste_real"], errors="coerce")

ratio = realv / m11.replace(0, np.nan)
log_ratio = np.log(ratio.replace([np.inf, -np.inf], np.nan))

clip = 1.2
df["log_ratio_ajuste"] = log_ratio.clip(lower=-clip, upper=clip)

df["w"] = _weight_series(df)

# calendario sobre fecha_post_pred
df["dow"] = df[fecha_post_col].dt.dayofweek.astype("Int64")
df["month"] = df[fecha_post_col].dt.month.astype("Int64")
df["weekofyear"] = df[fecha_post_col].dt.isocalendar().week.astype("Int64")

keep = [
    "fecha",
    fecha_post_col,
    "bloque_base" if "bloque_base" in df.columns else None,
    "variedad_canon" if "variedad_canon" in df.columns else None,
    "grado",
    "destino",
    ajuste_m11_col,
    "factor_ajuste_real",
    "log_ratio_ajuste",
    "w",
    "dow",
    "month",
    "weekofyear",
]

keep = [c for c in keep if c is not None and c in df.columns]

out = df[keep].copy()
out = out.rename(columns={fecha_post_col: "fecha_post_pred_used", ajuste_m11_col: "ajuste_m11_used"})
out["as_of_date"] = as_of
out["created_at"] = pd.Timestamp.utcnow()

(GOLD / "ml2_datasets").mkdir(parents=True, exist_ok=True)
write_parquet(out, OUT_DS)

n_real = int(out["factor_ajuste_real"].notna().sum())
print(f"[OK] Wrote dataset: {OUT_DS}")
print(f"      rows={len(out):,} rows_with_real={n_real:,} as_of_date={as_of.date()} clip={clip}")

```

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

```
-----
[56/164] FILE: \src\gold\build_ds_desp_poscosecha_ml2_v1.py
-----
```

```
from __future__ import annotations
```

```
from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd
```

```
from src.common.io import read_parquet, write_parquet
```

```
def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]
```

```
ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
SILVER = DATA / "silver"
```

```
# Universe: salida final de HIDR (ya tiene fecha_post_pred usada y ML1 hidr aplicado)
IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_hidr_grado_dia_bloque_destino_final.parquet"
```

```
# Real desperdicio (por fecha_post, destino)
IN_REAL = SILVER / "dim_mermas_ajuste_fecha_post_destino.parquet"
```

```
OUT_DS = GOLD / "ml2_datasets" / "ds_desp_poscosecha_ml2_v1.parquet"
```

```
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()
```

```
def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()
```

```
def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")
```

```
def _as_of_date(default_tz: str = "America/Guayaquil") -> pd.Timestamp:
    # ¿hoy? no se usa (día no cerrado) => as_of_date = hoy-1
    # pd.Timestamp.now() toma tz local del OS; usamos normalize y -1 día.
    return (pd.Timestamp.now().normalize() - pd.Timedelta(days=1))
```

```
def _resolve_fecha_post_pred(df: pd.DataFrame) -> str:
    for c in [
        "fecha_post_pred_final",
        "fecha_post_pred_used",
        "fecha_post_pred_ml1",
        "fecha_post_pred",
    ]:
        if c in df.columns:
            return c
    raise KeyError(
        "No encuentro columna de fecha_post_pred. Espero una de: "
        "fecha_post_pred_final / fecha_post_pred_used / fecha_post_pred_ml1 / fecha_post_pred"
    )
```

```
def _resolve_factor_desp_ml1(df: pd.DataFrame) -> str:
    for c in ["factor_desp_ml1", "factor_desp_pred_ml1", "factor_desp"]:
        if c in df.columns:
```

```

        return c
    raise KeyError(
        "No encuentro factor_desp ML1 en universe. Espero: factor_desp_ml1 / factor_desp_pred_ml1 / factor_des
p"
    )

def _resolve_tallos(df: pd.DataFrame) -> str:
    for c in [
        "tallos",
        "tallos_total_ml2",
        "tallos_real_dia",
        "tallos_pred_ml1_grado_dia",
        "tallos_final_grado_dia",
        "tallos_pred_ml1_grado_dia",
    ]:
        if c in df.columns:
            return c
    # si no hay tallos, usamos 1.0 como peso
    return ""

def main() -> None:
    as_of = _as_of_date()

    uni = read_parquet(IN_UNIVERSE).copy()
    uni.columns = [str(c).strip() for c in uni.columns]

    # Canon mínimas
    need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino"}
    miss = need - set(uni.columns)
    if miss:
        raise ValueError(f"Universe sin columnas: {sorted(miss)}")

    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_str(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])
    uni["destino"] = _canon_str(uni["destino"])
    # grado a str/cat
    if pd.api.types.is_numeric_dtype(uni["grado"]):
        uni["grado"] = _canon_int(uni["grado"]).astype("Int64")
    else:
        uni["grado"] = _canon_str(uni["grado"])

    # Cortar al as_of_date
    uni = uni.loc[uni["fecha"].notna() & (uni["fecha"] <= as_of)].copy()

    fpp_col = _resolve_fecha_post_pred(uni)
    fd_ml1_col = _resolve_factor_desp_ml1(uni)

    uni[fpp_col] = _to_date(uni[fpp_col])
    uni[fd_ml1_col] = pd.to_numeric(uni[fd_ml1_col], errors="coerce")

    tallos_col = _resolve_tallos(uni)
    if tallos_col:
        uni["tallos_w"] = pd.to_numeric(uni[tallos_col], errors="coerce").fillna(0.0)
    else:
        uni["tallos_w"] = 1.0

    # REAL: desperdicio por fecha_post + destino
    real = read_parquet(IN_REAL).copy()
    real.columns = [str(c).strip() for c in real.columns]
    need_r = {"fecha_post", "destino", "factor_desp"}
    miss_r = need_r - set(real.columns)
    if miss_r:
        raise ValueError(f"Real desperdicio sin columnas: {sorted(miss_r)}")

    real["fecha_post"] = _to_date(real["fecha_post"])
    real["destino"] = _canon_str(real["destino"])
    real["factor_desp_real"] = pd.to_numeric(real["factor_desp"], errors="coerce")

    # Colapsar a 1 fila por (fecha_post, destino) usando mediana (robusto)

```

```

real2 = (
    real.groupby(["fecha_post", "destino"], dropna=False, as_index=False)
    .agg(factor_desp_real=("factor_desp_real", "median"))
)

# Join: fecha_post_pred + destino
ds = uni.merge(
    real2,
    left_on=[fpp_col, "destino"],
    right_on=["fecha_post", "destino"],
    how="left",
)

ds["factor_desp_ml1"] = pd.to_numeric(ds[fdd_ml1_col], errors="coerce")
ds["factor_desp_real"] = pd.to_numeric(ds["factor_desp_real"], errors="coerce")

# Target: log_ratio_desp = log(real/ml1)
eps = 1e-12
ratio = (ds["factor_desp_real"] + eps) / (ds["factor_desp_ml1"] + eps)
ds["log_ratio_desp"] = np.log(ratio)

# Clip target (para estabilidad)
CLIP = 1.2
ds["log_ratio_desp_clipped"] = ds["log_ratio_desp"].clip(lower=-CLIP, upper=CLIP)

# Features calendario sobre fecha_post_pred usada
ds["fecha_post_pred_used"] = ds[fpp_col]
ds["dow"] = ds["fecha_post_pred_used"].dt.dayofweek.astype("Int64")
ds["month"] = ds["fecha_post_pred_used"].dt.month.astype("Int64")
ds["weekofyear"] = ds["fecha_post_pred_used"].dt.isocalendar().week.astype("Int64")

# Flags de disponibilidad real
ds["has_real"] = ds["factor_desp_real"].notna()

ds["as_of_date"] = as_of
ds["created_at"] = pd.Timestamp(datetime.now()).normalize()

OUT_DS.parent.mkdir(parents=True, exist_ok=True)
write_parquet(ds, OUT_DS)

print(f"[OK] Wrote dataset: {OUT_DS}")
print(
    f"      rows={len(ds):,} rows_with_real={int(ds['has_real'].sum()):,} "
    f"as_of_date={as_of.date()} clip={CLIP}"
)

if __name__ == "__main__":
    main()

-----
[57/164] FILE: \src\gold\build_ds_dh_poscosecha_ml2_v1.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime, timedelta

import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"

```

```

SILVER = DATA / "silver"

# Universe: ya contiene el split por destino y llaves bloque/variedad/grado/destino
IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_seed_grado_dia_bloque_destino.parquet"

# DH baseline aplicado (ML1) sobre seed ML2 (esto te da dh_dias_ml1 y fecha_post_pred_ml1)
IN_DH_ML1_ON_ML2 = GOLD / "pred_poscosecha_ml2_dh_grado_dia_bloque_destino.parquet"

# Real DH + hidratación (tiene dh_dias a nivel fecha_cosecha, fecha_post, grado, destino)
IN_REAL_HD = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"

OUT_DS = GOLD / "ml2_datasets" / "ds_dh_poscosecha_ml2_v1.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _as_of_date() -> pd.Timestamp:
    # Regla: usar hoy-1 (día cerrado)
    return pd.Timestamp.now().normalize() - pd.Timedelta(days=1)

def main(clip_err_days: float = 5.0) -> None:
    as_of = _as_of_date()

    uni = read_parquet(IN_UNIVERSE).copy()
    dhm = read_parquet(IN_DH_ML1_ON_ML2).copy()
    real = read_parquet(IN_REAL_HD).copy()

    for df in (uni, dhm, real):
        df.columns = [str(c).strip() for c in df.columns]

    # Canon keys
    uni["fecha"] = _to_date(uni["fecha"])
    uni["destino"] = _canon_str(uni["destino"])
    uni["grado"] = _canon_int(uni["grado"])
    uni["bloque_base"] = _canon_int(uni.get("bloque_base", pd.Series([pd.NA] * len(uni))))
    if "variedad_canon" in uni.columns:
        uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

    # Filtrar as_of (no usar hoy)
    uni = uni[uni["fecha"].notna() & (uni["fecha"] <= as_of)].copy()

    # DH ML1 sobre ML2 seed
    dhm["fecha"] = _to_date(dhm["fecha"])
    dhm["destino"] = _canon_str(dhm["destino"])
    dhm["grado"] = _canon_int(dhm["grado"])
    dhm["bloque_base"] = _canon_int(dhm.get("bloque_base", pd.Series([pd.NA] * len(dhm))))
    if "variedad_canon" in dhm.columns:
        dhm["variedad_canon"] = _canon_str(dhm["variedad_canon"])

    # Identificar columna DH ML1
    dh_col = None
    for c in ["dh_dias_ml1", "dh_dias_pred_ml1", "dh_dias_pred", "dh_dias"]:
        if c in dhm.columns:
            dh_col = c
            break
    if dh_col is None:
        raise KeyError("No encuentro columna dh en pred_poscosecha_ml2_dh_*. Espero dh_dias_ml1 o similar.")

    dhm[dh_col] = _canon_int(dhm[dh_col])

    # Real

```



```

real["fecha_cosecha"] = _to_date(real["fecha_cosecha"])
real["fecha_post"] = _to_date(real["fecha_post"])
real["destino"] = _canon_str(real["destino"])
real["grado"] = _canon_int(real["grado"])

# tallos real (para peso del KPI / entrenamiento)
if "tallos" in real.columns:
    real["tallos"] = pd.to_numeric(real["tallos"], errors="coerce").fillna(0.0)
else:
    real["tallos"] = 0.0

# dh real (col explícita)
if "dh_dias" in real.columns:
    real["dh_dias"] = _canon_int(real["dh_dias"])
else:
    # fallback: (fecha_post - fecha_cosecha)
    real["dh_dias"] = (real["fecha_post"] - real["fecha_cosecha"]).dt.days.astype("Int64")

# Agregar real a grano (fecha_cosecha, grado, destino) con median para dh y sum tallos
real_g = (
    real.groupby(["fecha_cosecha", "grado", "destino"], dropna=False, as_index=False)
        .agg(
            dh_real=("dh_dias", "median"),
            tallos_real=("tallos", "sum"),
            fecha_post_real=("fecha_post", "min"),
        )
)

# Unimos universe + dh_m11
keys = ["fecha", "bloque_base", "variedad_canon", "grado", "destino"]
need_uni = [c for c in keys if c in uni.columns]
need_dhm = [c for c in keys if c in dhm.columns]

df = uni.merge(
    dhm[need_dhm + [dh_col]].rename(columns={dh_col: "dh_m11"}),
    on=[c for c in keys if c in need_uni and c in need_dhm],
    how="left",
)

# Join real por fecha_cosecha=fecha + grado + destino
df = df.merge(
    real_g,
    left_on=["fecha", "grado", "destino"],
    right_on=["fecha_cosecha", "grado", "destino"],
    how="left",
)

# Features calendario sobre fecha_cosecha (fecha)
df["dow"] = df["fecha"].dt.dayofweek.astype("Int64")
df["month"] = df["fecha"].dt.month.astype("Int64")
df["weekofyear"] = df["fecha"].dt.isocalendar().week.astype("Int64")

# Target: error en días
df["dh_m11"] = _canon_int(df["dh_m11"])
df["dh_real"] = _canon_int(df["dh_real"])

df["err_dh_days"] = (df["dh_real"].astype("float") - df["dh_m11"].astype("float"))
df["err_dh_days_clipped"] = df["err_dh_days"].clip(lower=-float(clip_err_days), upper=float(clip_err_days))

df["clip_err_days"] = float(clip_err_days)
df["as_of_date"] = as_of
df["created_at"] = pd.Timestamp.utcnow()

# Guardar DS completo (incluye filas sin real; el train filtrará)
OUT_DS.parent.mkdir(parents=True, exist_ok=True)
write_parquet(df, OUT_DS)

n_all = len(df)
n_real = int(df["dh_real"].notna().sum())
print(f"[OK] Wrote dataset: {OUT_DS}")
print(f"      rows={n_all:,} rows_with_real={n_real:,} as_of_date={as_of.date()} clip={clip_err_days:g}")

```

```

if __name__ == "__main__":
    main()

-----
[58/164] FILE: \src\gold\build_ds_harvest_horizon_ml2_v2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    # ../src/gold/file.py -> repo_root = parents[2]
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"
IN_GRID_ML1 = GOLD_DIR / "universe_harvest_grid_ml1.parquet"
IN_CLIMA = SILVER_DIR / "dim_clima_bloque_dia.parquet"
IN_FACTOR_SOH = GOLD_DIR / "factors" / "factor_ml2_harvest_start.parquet" # ML2 Inicio (prod)

OUT_DS = GOLD_DIR / "ml2_datasets" / "ds_harvest_horizon_ml2_v2.parquet"

# As-of sampling (V2)
ASOF_FREQ_DAYS = 7
MIN_DAYS_AFTER_SP = 14
MAX_ASOF_POINTS_PER_CICLO = 10

# Real duration sanity
MIN_REAL_DAYS = 1
MAX_REAL_DAYS = 120

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _safe_div(a: pd.Series, b: pd.Series) -> pd.Series:
    b = b.replace(0, np.nan)
    return (a / b).fillna(0.0)

def _build_cycle_header(grid: pd.DataFrame) -> pd.DataFrame:
    """
    Reduce universe_harvest_grid_ml1 (diario) a 1 fila por ciclo_id.
    """
    g = grid.copy()
    g["harvest_start_pred"] = _to_date(g["harvest_start_pred"])
    g["harvest_end_pred"] = _to_date(g["harvest_end_pred"])
    g["fecha_sp"] = _to_date(g["fecha_sp"])

    head = (
        g.groupby("ciclo_id", as_index=False)
        .agg(
            bloque_base=("bloque_base", "first"),

```

```

        variedad_canon=("variedad_canon", "first"),
        area=("area", "first"),
        tipo_sp=("tipo_sp", "first"),
        estado=("estado", "first"),
        tallos_proy=("tallos_proy", "first"),
        fecha_sp=("fecha_sp", "first"),
        harvest_start_pred=("harvest_start_pred", "min"),
        harvest_end_pred=("harvest_end_pred", "max"),
        n_harvest_days_pred=("n_harvest_days_pred", "max"),
        mll_version=("mll_version", "max"),
    )
)
head["days_sp_to_start_pred"] = (head["harvest_start_pred"] - head["fecha_sp"]).dt.days
return head

def _make_asof_dates(fecha_sp: pd.Timestamp, inicio_real: pd.Timestamp) -> list[pd.Timestamp]:
    """
    V2: as_of cada ASOF_FREQ_DAYS desde (sp + MIN_DAYS_AFTER_SP) hasta (inicio_real - 1).
    Cap MAX_ASOF_POINTS_PER_CICLO y asegura incluir el último as_of.
    """
    if pd.isna(fecha_sp) or pd.isna(inicio_real):
        return []

    last = inicio_real - pd.Timedelta(days=1)
    start = fecha_sp + pd.Timedelta(days=MIN_DAYS_AFTER_SP)

    if last < start:
        return [max(fecha_sp, last)]

    dates = list(pd.date_range(start=start, end=last, freq=f"{ASOF_FREQ_DAYS}D"))
    if not dates or dates[-1] != last:
        dates.append(last)

    if len(dates) > MAX_ASOF_POINTS_PER_CICLO:
        dates = dates[-MAX_ASOF_POINTS_PER_CICLO:]

    return [pd.Timestamp(d).normalize() for d in dates]

def _features_asof(clima: pd.DataFrame, ciclo_chunk: pd.DataFrame, as_of_date: pd.Timestamp) -> pd.DataFrame:
    """
    Features clima por ciclo_id usando rango [fecha_sp, as_of_date].
    Si un ciclo no tiene clima en el rango, queda con ceros (no se pierde).
    """
    base = ciclo_chunk[["ciclo_id", "bloque_base", "fecha_sp"]].drop_duplicates().copy()
    base["bloque_base"] = _canon_str(base["bloque_base"])
    base["fecha_sp"] = _to_date(base["fecha_sp"])

    cl = clima.copy()
    cl["fecha"] = _to_date(cl["fecha"])
    cl["bloque_base"] = _canon_str(cl["bloque_base"])
    cl = cl.loc[cl["fecha"] <= as_of_date, :].copy()

    m = base.merge(cl, on="bloque_base", how="left")
    m = m.loc[(m["fecha"] >= m["fecha_sp"]) & (m["fecha"] <= as_of_date), :].copy()

    # no hay datos -> ceros
    if m.empty:
        out = base[["ciclo_id"]].copy()
        for c in [
            "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
            "rain_cum_sp", "rain_7d", "enllovía_days_7d",
            "solar_cum_sp", "solar_7d",
            "temp_avg_7d",
        ]:
            out[c] = 0.0
        return out

    # usar gdc_dia acumulado (ignorar gdc_base)
    m["gdc_dia"] = pd.to_numeric(m["gdc_dia"], errors="coerce").fillna(0.0)
    m["rainfall_mm_dia"] = pd.to_numeric(m["rainfall_mm_dia"], errors="coerce").fillna(0.0)

```

```

m["solar_energy_j_m2_dia"] = pd.to_numeric(m["solar_energy_j_m2_dia"], errors="coerce").fillna(0.0)
m["temp_avg_dia"] = pd.to_numeric(m["temp_avg_dia"], errors="coerce")
m["en_lluvia_dia"] = pd.to_numeric(m["en_lluvia_dia"], errors="coerce").fillna(0.0)

m = m.sort_values(["ciclo_id", "fecha"])

def _roll_sum(s: pd.Series, w: int) -> pd.Series:
    return s.rolling(window=w, min_periods=1).sum()

def _roll_mean(s: pd.Series, w: int) -> pd.Series:
    return s.rolling(window=w, min_periods=1).mean()

m["gdc_7d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 7))
m["gdc_14d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 14))
m["rain_7d"] = m.groupby("ciclo_id")["rainfall_mm_dia"].transform(lambda s: _roll_sum(s, 7))
m["solar_7d"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].transform(lambda s: _roll_sum(s, 7))
m["enlluvia_days_7d"] = m.groupby("ciclo_id")["en_lluvia_dia"].transform(lambda s: _roll_sum(s, 7))
m["temp_avg_7d"] = m.groupby("ciclo_id")["temp_avg_dia"].transform(lambda s: _roll_mean(s, 7))

m["gdc_cum_sp"] = m.groupby("ciclo_id")["gdc_dia"].cumsum()
m["rain_cum_sp"] = m.groupby("ciclo_id")["rainfall_mm_dia"].cumsum()
m["solar_cum_sp"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].cumsum()

last = m.groupby("ciclo_id", as_index=False).tail(1).copy()
last["days_from_sp"] = (as_of_date - last["fecha_sp"]).dt.days
last["days_from_sp"] = pd.to_numeric(last["days_from_sp"], errors="coerce").fillna(0).clip(lower=0)
last["gdc_per_day"] = _safe_div(last["gdc_cum_sp"], last["days_from_sp"].replace(0, np.nan))

feat_cols = [
    "ciclo_id",
    "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
    "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
    "solar_cum_sp", "solar_7d",
    "temp_avg_7d",
]
last = last[feat_cols].copy()

out = base[["ciclo_id"]].merge(last, on="ciclo_id", how="left")

for c in [
    "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
    "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
    "solar_cum_sp", "solar_7d",
]:
    out[c] = pd.to_numeric(out[c], errors="coerce").fillna(0.0)

out["temp_avg_7d"] = pd.to_numeric(out["temp_avg_7d"], errors="coerce")
med = out["temp_avg_7d"].median()
if pd.isna(med):
    med = 0.0
out["temp_avg_7d"] = out["temp_avg_7d"].fillna(med)

return out

def main() -> None:
    ciclo = read_parquet(IN_CICLO).copy()
    grid = read_parquet(IN_GRID_ML1).copy()
    clima = read_parquet(IN_CLIMA).copy()

    # ML2 inicio (factor) es opcional: si existe lo usamos; si no, fallback a 0
    factor_soh = None
    if IN_FACTOR_SOH.exists():
        factor_soh = read_parquet(IN_FACTOR_SOH).copy()

    # canon y fechas
    ciclo["bloque_base"] = _canon_str(ciclo["bloque_base"])
    ciclo["fecha_sp"] = _to_date(ciclo["fecha_sp"])
    ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])
    ciclo["fecha_fin_cosecha"] = _to_date(ciclo["fecha_fin_cosecha"])
    ciclo["estado"] = _canon_str(ciclo["estado"])

```

```

grid["bloque_base"] = _canon_str(grid["bloque_base"])
grid["variedad_canon"] = _canon_str(grid["variedad_canon"])

head = _build_cycle_header(grid)

# Base ciclo
df0 = ciclo.merge(head, on="ciclo_id", how="inner", suffixes=("", "_ml1"))

# Requiere reales válidos de inicio/fin + pred n_harvest_days
df0 = df0.loc[
    df0["fecha_inicio_cosecha"].notna()
    & df0["fecha_fin_cosecha"].notna()
    & df0["n_harvest_days_pred"].notna(),
    :
].copy()

# Real duration
df0["n_harvest_days_real"] = (df0["fecha_fin_cosecha"] - df0["fecha_inicio_cosecha"]).dt.days + 1
df0["n_harvest_days_real"] = pd.to_numeric(df0["n_harvest_days_real"], errors="coerce")

df0 = df0.loc[
    df0["n_harvest_days_real"].between(MIN_REAL_DAYS, MAX_REAL_DAYS, inclusive="both"),
    :
].copy()

df0["n_harvest_days_pred"] = pd.to_numeric(df0["n_harvest_days_pred"], errors="coerce")
df0 = df0.loc[df0["n_harvest_days_pred"].notna(), :].copy()

# Target
df0["error_horizon_days"] = (df0["n_harvest_days_real"] - df0["n_harvest_days_pred"]).astype(int)

# Panel as_of por ciclo (V2)
rows = []
for r in df0.itertuples(index=False):
    for a in _make_asof_dates(r.fecha_sp, r.fecha_inicio_cosecha):
        rows.append((r.ciclo_id, a))
asof_df = pd.DataFrame(rows, columns=["ciclo_id", "as_of_date"])

df = df0.merge(asof_df, on="ciclo_id", how="inner")
df = df.loc[df["as_of_date"] < df["fecha_inicio_cosecha"], :].copy()

# Calendario
df["dow"] = df["as_of_date"].dt.dayofweek
df["month"] = df["as_of_date"].dt.month
df["weekofyear"] = df["as_of_date"].dt.isocalendar().week.astype(int)
df["days_from_sp"] = (df["as_of_date"] - df["fecha_sp"]).dt.days

# Agregar pred_error_start_days (ML2 Inicio) si existe
df["pred_error_start_days"] = 0.0
if factor_soh is not None and len(factor_soh):
    factor_soh = factor_soh.copy()
    # si factor tiene as_of_date o no: en prod normalmente es único; aquí usamos el valor por ciclo
    factor_soh["pred_error_start_days"] = pd.to_numeric(factor_soh["pred_error_start_days"], errors="coerc
e")

    soh = factor_soh[["ciclo_id", "pred_error_start_days"]].drop_duplicates("ciclo_id")
    df = df.drop(columns=["pred_error_start_days"]).merge(soh, on="ciclo_id", how="left")
    df["pred_error_start_days"] = pd.to_numeric(df["pred_error_start_days"], errors="coerce").fillna(0.0)

# Features clima por as_of_date (batch)
feats = []
for as_of_date, chunk in df.groupby("as_of_date"):
    feats.append(_features_asof(clima, chunk[["ciclo_id", "bloque_base", "fecha_sp"]], pd.Timestamp(as_of_
date)))
feat = pd.concat(feats, ignore_index=True) if feats else pd.DataFrame(columns=["ciclo_id"])

df = df.merge(feat, on="ciclo_id", how="left")

# Final imputaciones numéricas
for c in [
    "days_sp_to_start_pred",
    "n_harvest_days_pred",
    "tallos_proy",

```

```

        "days_from_sp",
        "pred_error_start_days",
        "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
        "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
        "solar_cum_sp", "solar_7d",
        "temp_avg_7d",
    ]:
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)

    df["created_at"] = pd.Timestamp(datetime.now()).normalize()

    OUT_DS.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(df, OUT_DS)

    print(f"[OK] Wrote dataset: {OUT_DS}")
    print(f"      rows={len(df):,} cycles={df['ciclo_id'].nunique():,} as_of_dates={df['as_of_date'].nunique():,}")

if __name__ == "__main__":
    main()

-----
[59/164] FILE: \src\gold\build_ds_harvest_start_ml2_v2.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    # ../src/gold/file.py -> repo_root = parents[2]
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"
IN_GRID_ML1 = GOLD_DIR / "universe_harvest_grid_ml1.parquet"
IN_CLIMA = SILVER_DIR / "dim_clima_bloque_dia.parquet"

OUT_DS = GOLD_DIR / "ml2_datasets" / "ds_harvest_start_ml2_v2.parquet"

# V2 params
ASOF_FREQ_DAYS = 7
MIN_DAYS_AFTER_SP = 14
MAX_ASOF_POINTS_PER_CICLO = 10

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _safe_div(a: pd.Series, b: pd.Series) -> pd.Series:
    b = b.replace(0, np.nan)
    return (a / b).fillna(0.0)

```

```

def _build_cycle_header(grid: pd.DataFrame) -> pd.DataFrame:
    """
    Reduce universe_harvest_grid_mll (diario) a 1 fila por ciclo_id.
    """
    g = grid.copy()
    g["harvest_start_pred"] = _to_date(g["harvest_start_pred"])
    g["harvest_end_pred"] = _to_date(g["harvest_end_pred"])
    g["fecha_sp"] = _to_date(g["fecha_sp"])

    head = (
        g.groupby("ciclo_id", as_index=False)
        .agg(
            bloque_base=("bloque_base", "first"),
            variedad_canon=("variedad_canon", "first"),
            area=("area", "first"),
            tipo_sp=("tipo_sp", "first"),
            estado=("estado", "first"),
            tallos_proy=("tallos_proy", "first"),
            fecha_sp=("fecha_sp", "first"), # ? FIX: incluir fecha_sp
            harvest_start_pred=("harvest_start_pred", "min"),
            harvest_end_pred=("harvest_end_pred", "max"),
            n_harvest_days_pred=("n_harvest_days_pred", "max"),
            mll_version=("mll_version", "max"),
        )
    )

    # Robustez: si falta alguna fecha, queda NaN
    head["days_sp_to_start_pred"] = (head["harvest_start_pred"] - head["fecha_sp"]).dt.days
    return head


def _make_asof_dates(fecha_sp: pd.Timestamp, inicio_real: pd.Timestamp) -> list[pd.Timestamp]:
    if pd.isna(fecha_sp) or pd.isna(inicio_real):
        return []

    last = inicio_real - pd.Timedelta(days=1)
    start = fecha_sp + pd.Timedelta(days=MIN_DAYS_AFTER_SP)

    if last < start:
        return [max(fecha_sp, last)]

    dates = list(pd.date_range(start=start, end=last, freq=f"{ASOF_FREQ_DAYS}D"))
    if not dates or dates[-1] != last:
        dates.append(last)

    if len(dates) > MAX_ASOF_POINTS_PER_CICLO:
        dates = dates[-MAX_ASOF_POINTS_PER_CICLO:]

    return [pd.Timestamp(d).normalize() for d in dates]


def _features_asof(clima: pd.DataFrame, ciclo_chunk: pd.DataFrame, as_of_date: pd.Timestamp) -> pd.DataFrame:
    cl = clima.copy()
    cl["fecha"] = _to_date(cl["fecha"])
    cl["bloque_base"] = _canon_str(cl["bloque_base"])
    cl = cl.loc[cl["fecha"] <= as_of_date, :].copy()

    cy = ciclo_chunk[["ciclo_id", "bloque_base", "fecha_sp"]].drop_duplicates().copy()
    cy["bloque_base"] = _canon_str(cy["bloque_base"])
    cy["fecha_sp"] = _to_date(cy["fecha_sp"])

    m = cy.merge(cl, on="bloque_base", how="left")
    m = m.loc[(m["fecha"] >= m["fecha_sp"]) & (m["fecha"] <= as_of_date), :].copy()

    # IMPORTANT: usar gdc_dia (acumular). Ignorar gdc_base.
    m["gdc_dia"] = pd.to_numeric(m["gdc_dia"], errors="coerce").fillna(0.0)
    m["rainfall_mm_dia"] = pd.to_numeric(m["rainfall_mm_dia"], errors="coerce").fillna(0.0)
    m["solar_energy_j_m2_dia"] = pd.to_numeric(m["solar_energy_j_m2_dia"], errors="coerce").fillna(0.0)
    m["temp_avg_dia"] = pd.to_numeric(m["temp_avg_dia"], errors="coerce")
    m["en_lluvia_dia"] = pd.to_numeric(m["en_lluvia_dia"], errors="coerce").fillna(0.0)

```

```

m = m.sort_values(["ciclo_id", "fecha"])

def _roll_sum(s: pd.Series, w: int) -> pd.Series:
    return s.rolling(window=w, min_periods=1).sum()

def _roll_mean(s: pd.Series, w: int) -> pd.Series:
    return s.rolling(window=w, min_periods=1).mean()

m["gdc_7d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 7))
m["gdc_14d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 14))
m["rain_7d"] = m.groupby("ciclo_id")["rainfall_mm_dia"].transform(lambda s: _roll_sum(s, 7))
m["solar_7d"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].transform(lambda s: _roll_sum(s, 7))
m["enlluvia_days_7d"] = m.groupby("ciclo_id")["en_lluvia_dia"].transform(lambda s: _roll_sum(s, 7))
m["temp_avg_7d"] = m.groupby("ciclo_id")["temp_avg_dia"].transform(lambda s: _roll_mean(s, 7))

m["gdc_cum_sp"] = m.groupby("ciclo_id")["gdc_dia"].cumsum()
m["rain_cum_sp"] = m.groupby("ciclo_id")["rainfall_mm_dia"].cumsum()
m["solar_cum_sp"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].cumsum()

last = m.groupby("ciclo_id", as_index=False).tail(1).copy()
last["days_from_sp"] = (as_of_date - last["fecha_sp"]).dt.days
last["days_from_sp"] = pd.to_numeric(last["days_from_sp"], errors="coerce").fillna(0).clip(lower=0)
last["gdc_per_day"] = _safe_div(last["gdc_cum_sp"], last["days_from_sp"].replace(0, np.nan))

feat_cols = [
    "ciclo_id",
    "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
    "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
    "solar_cum_sp", "solar_7d",
    "temp_avg_7d",
]
return last[feat_cols]

def main() -> None:
    ciclo = read_parquet(IN_CICLO).copy()
    grid = read_parquet(IN_GRID_ML1).copy()
    clima = read_parquet(IN_CLIMA).copy()

    ciclo["bloque_base"] = _canon_str(ciclo["bloque_base"])
    ciclo["fecha_sp"] = _to_date(ciclo["fecha_sp"])
    ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])
    ciclo["fecha_fin_cosecha"] = _to_date(ciclo["fecha_fin_cosecha"])

    grid["bloque_base"] = _canon_str(grid["bloque_base"])
    grid["variedad_canon"] = _canon_str(grid["variedad_canon"])

    head = _build_cycle_header(grid)
    df0 = ciclo.merge(head, on="ciclo_id", how="inner", suffixes=("", "_ml1"))

    df0 = df0.loc[df0["fecha_inicio_cosecha"].notna() & df0["harvest_start_pred"].notna(), :].copy()

    # Panel as_of
    rows = []
    for r in df0.itertuples(index=False):
        for a in _make_asof_dates(r.fecha_sp, r.fecha_inicio_cosecha):
            rows.append((r.ciclo_id, a))
    asof_df = pd.DataFrame(rows, columns=["ciclo_id", "as_of_date"])

    df = df0.merge(asof_df, on="ciclo_id", how="inner")
    df = df.loc[df["as_of_date"] < df["fecha_inicio_cosecha"], :].copy()

    df["error_start_days"] = (df["fecha_inicio_cosecha"] - df["harvest_start_pred"]).dt.days.astype(int)

    df["dow"] = df["as_of_date"].dt.dayofweek
    df["month"] = df["as_of_date"].dt.month
    df["weekofyear"] = df["as_of_date"].dt.isocalendar().week.astype(int)

    feats = []
    for as_of_date, chunk in df.groupby("as_of_date"):
        feats.append(_features_asof(clima, chunk[["ciclo_id", "bloque_base", "fecha_sp"]], as_of_date))
    feat = pd.concat(feats, ignore_index=True) if feats else pd.DataFrame(columns=["ciclo_id"])

```



```

df = df.merge(feat, on="ciclo_id", how="left")

for c in [
    "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
    "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
    "solar_cum_sp", "solar_7d",
]:
    if c in df.columns:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)

if "temp_avg_7d" in df.columns:
    df["temp_avg_7d"] = pd.to_numeric(df["temp_avg_7d"], errors="coerce")
    df["temp_avg_7d"] = df["temp_avg_7d"].fillna(df["temp_avg_7d"].median())

df["created_at"] = pd.Timestamp(datetime.now()).normalize()

OUT_DS.parent.mkdir(parents=True, exist_ok=True)
write_parquet(df, OUT_DS)
print(f"[OK] Wrote {OUT_DS} rows={len(df):}, cycles={df['ciclo_id'].nunique():}")

if __name__ == "__main__":
    main()

-----
[60/164] FILE: \src\gold\build_ds_hidr_poscosecha_ml2_v1.py
-----
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
SILVER = DATA / "silver"

# ? base con hidr ML1 y demás columnas
IN_BASE = GOLD / "pred_poscosecha_ml2_full_grado_dia_bloque_destino.parquet"
# ? dh_final con dh_dias_final + fecha_post_pred_final
IN_DH_FINAL = GOLD / "pred_poscosecha_ml2_dh_grado_dia_bloque_destino_final.parquet"

IN_REAL_HIDR = SILVER / "fact_hidratacion_real_post_grado_destino.parquet"

OUT_DS = GOLD / "ml2_datasets" / "ds_hidr_poscosecha_ml2_v1.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _as_of_date_today_minus_1() -> pd.Timestamp:
    return (pd.Timestamp.now().normalize() - pd.Timedelta(days=1)).normalize()

```

```

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise KeyError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _factor_from_hidr_pct(hidr_pct: pd.Series) -> pd.Series:
    x = pd.to_numeric(hidr_pct, errors="coerce")
    return np.where(x.isna(), np.nan, np.where(x > 3.5, 1.0 + x / 100.0, x)).astype(float)

def main() -> None:
    as_of_date = _as_of_date_today_minus_1()
    created_at = pd.Timestamp.utcnow()

    base = read_parquet(IN_BASE).copy()
    base.columns = [str(c).strip() for c in base.columns]

    _require(base, ["fecha", "bloque_base", "variedad_canon", "grado", "destino"], "pred_poscosecha_ml2_full")
    # hidr ML1 debe existir aquí
    if "factor_hidr_ml1" not in base.columns:
        raise KeyError(f"pred_poscosecha_ml2_full no tiene factor_hidr_ml1. Columnas={list(base.columns)}")

    base["fecha"] = _to_date(base["fecha"])
    base["destino"] = _canon_str(base["destino"])
    base["grado"] = _canon_int(base["grado"])
    base["bloque_base"] = _canon_str(base["bloque_base"])
    base["variedad_canon"] = _canon_str(base["variedad_canon"])

    # filtro hoy-1
    base = base.loc[base["fecha"].notna() & (base["fecha"] <= as_of_date)].copy()

    # traer DH final
    dh = read_parquet(IN_DH_FINAL).copy()
    dh.columns = [str(c).strip() for c in dh.columns]
    _require(dh, ["fecha", "bloque_base", "variedad_canon", "grado", "destino", "dh_dias_final", "fecha_post_p
red_final"], "pred_poscosecha_ml2_dh_final")

    dh["fecha"] = _to_date(dh["fecha"])
    dh["destino"] = _canon_str(dh["destino"])
    dh["grado"] = _canon_int(dh["grado"])
    dh["bloque_base"] = _canon_str(dh["bloque_base"])
    dh["variedad_canon"] = _canon_str(dh["variedad_canon"])
    dh["fecha_post_pred_final"] = _to_date(dh["fecha_post_pred_final"])

    key = ["fecha", "bloque_base", "variedad_canon", "grado", "destino"]
    dh_take = dh[key + ["dh_dias_final", "fecha_post_pred_final"]].copy()

    uni = base.merge(dh_take, on=key, how="left")

    # features calendario por fecha_post_pred_final
    uni["dow"] = uni["fecha_post_pred_final"].dt.dayofweek.astype("Int64")
    uni["month"] = uni["fecha_post_pred_final"].dt.month.astype("Int64")
    uni["weekofyear"] = uni["fecha_post_pred_final"].dt.isocalendar().week.astype("Int64")

    # pesos por tallos si existe
    tallos_col = None
    for c in ["tallos", "tallos_final_grado_dia", "tallos_pred_ml2_grado_dia", "tallos_pred_ml1_grado_dia"]:
        if c in uni.columns:
            tallos_col = c
            break
    uni["tallos_w"] = pd.to_numeric(uni[tallos_col], errors="coerce").fillna(0.0).clip(lower=0.0) if tallos_co
l else 1.0

    uni["factor_hidr_ml1"] = pd.to_numeric(uni["factor_hidr_ml1"], errors="coerce")

    # ----- REAL HIDR
    real = read_parquet(IN_REAL_HIDR).copy()
    real.columns = [str(c).strip() for c in real.columns]
    _require(real, ["fecha_post", "grado", "destino"], "fact_hidratacion_real_post_grado_destino")

```

```

if "hidr_pct" in real.columns:
    real["factor_hidr_real"] = _factor_from_hidr_pct(real["hidr_pct"])
else:
    _require(real, ["peso_base_g", "peso_post_g"], "fact_hidratacion_real_post_grado_destino")
    pb = pd.to_numeric(real["peso_base_g"], errors="coerce")
    pp = pd.to_numeric(real["peso_post_g"], errors="coerce")
    real["factor_hidr_real"] = np.where(pb > 0, pp / pb, np.nan)

real["fecha_post"] = _to_date(real["fecha_post"])
real["destino"] = _canon_str(real["destino"])
real["grado"] = _canon_int(real["grado"])

# agregación real por llave
if "tallos" in real.columns:
    real["tallos"] = pd.to_numeric(real["tallos"], errors="coerce").fillna(0.0)
    g = real.groupby(["fecha_post", "grado", "destino"], dropna=False)
    real2 = g.apply(
        lambda x: pd.Series({
            "factor_hidr_real": float(np.nansum(x["factor_hidr_real"] * x["tallos"]) / np.nansum(x["tallos"])),
            "tallos_real_sum": float(np.nansum(x["tallos"])),
        })
    ).reset_index()
else:
    real2 = (
        real.groupby(["fecha_post", "grado", "destino"], dropna=False, as_index=False)
        .agg(factor_hidr_real=("factor_hidr_real", "median"))
    )
    real2["tallos_real_sum"] = np.nan

ds = uni.merge(
    real2,
    left_on=["fecha_post_pred_final", "grado", "destino"],
    right_on=["fecha_post", "grado", "destino"],
    how="left",
).drop(columns=["fecha_post"], errors="ignore")

# target log error
eps = 1e-9
ds["ratio_hidr"] = ds["factor_hidr_real"] / ds["factor_hidr_ml1"].clip(lower=eps)
ds["log_error_hidr"] = np.log(ds["ratio_hidr"].clip(lower=eps))

clip = 1.2
ds["log_error_hidr_clipped"] = ds["log_error_hidr"].clip(lower=-clip, upper=clip)
ds["is_in_real"] = ds["factor_hidr_real"].notna()

ds["as_of_date"] = as_of_date
ds["created_at"] = created_at

OUT_DS.parent.mkdir(parents=True, exist_ok=True)
write_parquet(ds, OUT_DS)

print(f"[OK] Wrote dataset: {OUT_DS}")
print(f"    rows={len(ds):,} rows_with_real={int(ds['is_in_real'].sum()):,} as_of_date={as_of_date.date()}")
} clip=±{clip}")

if __name__ == "__main__":
    main()

```

```

-----
[61/164] FILE: \src\gold\build_ds_peso_tallo_ml2_v1.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

```

```

from src.common.io import read_parquet, write_parquet

```

```

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
SILVER = DATA / "silver"

IN_GRID = GOLD / "universe_harvest_grid_ml2.parquet"
IN_CLIMA = SILVER / "dim_clima_bloque_dia.parquet"

# ML1 peso por tallo (ajusta si tu nombre es distinto; dejo dos candidatos)
IN_ML1_A = GOLD / "pred_peso_tallo_grado_ml1.parquet"
IN_ML1_B = GOLD / "pred_peso_tallo_grado_ml1_full.parquet"

# Reales (Opción A: ya tienes peso/tallo real por grado-día)
IN_REAL_PT = SILVER / "fact_peso_tallo_real_grado_dia.parquet"

# Reales (Opción B: derivar desde kg_real y tallos_real)
IN_REAL_KG = SILVER / "fact_kg_real_grado_dia.parquet"
IN_REAL_TALLOS = SILVER / "fact_cosecha_real_grado_dia.parquet"

OUT_DS = GOLD / "ml2_datasets" / "ds_peso_tallo_ml2_v1.parquet"

EPS = 0.1 # gramos, para estabilidad (peso/tallo suele ser > 0)
CLIP_LOG_ERR = 0.8 # exp(±0.8) ? [0.45 .. 2.23]

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _resolve_block_base(df: pd.DataFrame) -> pd.DataFrame:
    if "bloque_base" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_base"])
        return df
    if "bloque_padre" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_padre"])
        return df
    if "bloque" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque"])
        return df
    raise KeyError("No encuentro columna de bloque: bloque_base / bloque_padre / bloque")

def _pick_ml1_file() -> Path:
    if IN_ML1_A.exists():
        return IN_ML1_A
    if IN_ML1_B.exists():
        return IN_ML1_B
    raise FileNotFoundError(f"No encuentro ML1 peso/tallo: {IN_ML1_A} ni {IN_ML1_B}")

def _load_real_peso_tallo() -> pd.DataFrame:
    # Opción A
    if IN_REAL_PT.exists():
        real = read_parquet(IN_REAL_PT).copy()
        # Esperamos algo como: fecha, bloque(_base), variedad, grado, peso_tallo_real (g)
        real["fecha"] = _to_date(real["fecha"])
        real = _resolve_block_base(real)
        if "grado" in real.columns:
            real["grado"] = _canon_str(real["grado"])
        if "variedad" in real.columns:
            real["variedad"] = _canon_str(real["variedad"])

```

```

# detectar col peso
cand = [c for c in ["peso_tallo_real_g", "peso_tallo_real", "peso_tallo_g_real", "peso_tallo_g"] if c
in real.columns]
if not cand and ("peso_real_g" in real.columns) and ("tallos_real" in real.columns):
    tmp = real[["fecha", "bloque_base", "grado", "peso_real_g", "tallos_real"]].copy()
    tmp["peso_real_g"] = pd.to_numeric(tmp["peso_real_g"], errors="coerce").fillna(0.0)
    tmp["tallos_real"] = pd.to_numeric(tmp["tallos_real"], errors="coerce").fillna(0.0)
    tmp["peso_tallo_real_g"] = np.where(tmp["tallos_real"] > 0, tmp["peso_real_g"] / tmp["tallos_real"]
], np.nan)
    return tmp[["fecha", "bloque_base", "grado", "peso_tallo_real_g"]]

if not cand:
    raise KeyError("fact_peso_tallo_real_grado_dia no tiene columna de peso. "
                  "Espero: peso_tallo_real / peso_tallo_g_real / peso_tallo_g")
col = cand[0]
out = real[["fecha", "bloque_base", "grado", col]].rename(columns={col: "peso_tallo_real_g"})
out["peso_tallo_real_g"] = pd.to_numeric(out["peso_tallo_real_g"], errors="coerce")
return out

# Opción B: kg_real / tallos_real
if not IN_REAL_KG.exists() or not IN_REAL_TALLOS.exists():
    raise FileNotFoundError(
        "No encuentro reales para peso/tallo. Necesito IN_REAL_PT o (IN_REAL_KG + IN_REAL_TALLOS). "
        f"Faltan: {IN_REAL_PT} o {IN_REAL_KG} / {IN_REAL_TALLOS}"
    )

kg = read_parquet(IN_REAL_KG).copy()
tl = read_parquet(IN_REAL_TALLOS).copy()

kg["fecha"] = _to_date(kg["fecha"])
tl["fecha"] = _to_date(tl["fecha"])

kg = _resolve_block_base(kg)
tl = _resolve_block_base(tl)

if "grado" in kg.columns:
    kg["grado"] = _canon_str(kg["grado"])
if "grado" in tl.columns:
    tl["grado"] = _canon_str(tl["grado"])

# detectar columna kg real
cand_kg = [c for c in ["kg_real", "kg", "kg_real_grado_dia"] if c in kg.columns]
if not cand_kg:
    raise KeyError("fact_kg_real_grado_dia no tiene kg. Espero: kg_real / kg / kg_real_grado_dia")
col_kg = cand_kg[0]

kg2 = kg.groupby(["fecha", "bloque_base", "grado"], as_index=False).agg(kg_real=(col_kg, "sum"))
tl2 = tl.groupby(["fecha", "bloque_base", "grado"], as_index=False).agg(tallos_real=("tallos_real", "sum"))
)

m = kg2.merge(tl2, on=["fecha", "bloque_base", "grado"], how="inner")
m["kg_real"] = pd.to_numeric(m["kg_real"], errors="coerce").fillna(0.0)
m["tallos_real"] = pd.to_numeric(m["tallos_real"], errors="coerce").fillna(0.0)

# g/tallo
m["peso_tallo_real_g"] = np.where(m["tallos_real"] > 0, (m["kg_real"] * 1000.0) / m["tallos_real"], np.nan)
)

return m[["fecha", "bloque_base", "grado", "peso_tallo_real_g"]]

def main() -> None:
    grid = read_parquet(IN_GRID).copy()
    clima = read_parquet(IN_CLIMA).copy()

    grid["fecha"] = _to_date(grid["fecha"])
    grid["bloque_base"] = _canon_str(grid["bloque_base"])
    if "variedad_canon" in grid.columns:
        grid["variedad_canon"] = _canon_str(grid["variedad_canon"])

    # ML1 peso/tallo
    mll_path = _pick_mll_file()
    mll = read_parquet(mll_path).copy()

```

```

mll["bloque_base"] = _canon_str(mll["bloque_base"]) if "bloque_base" in mll.columns else mll.assign(bloque_base=None)
if "grado" in mll.columns:
    mll["grado"] = _canon_str(mll["grado"])

# detectar columna peso mll
cand_mll = [c for c in ["peso_tallo_mll_g", "peso_tallo_mll", "peso_tallo_pred_mll_g", "peso_tallo_pred_mll"] if c in mll.columns]
if not cand_mll:
    # si no existe, usa peso_tallo_pred (legacy)
    cand_mll = [c for c in ["peso_tallo_pred", "peso_tallo_g_pred"] if c in mll.columns]
if not cand_mll:
    raise KeyError(
        "No encuentro columna de peso/tallo ML1 en pred_peso_tallo_grado_mll*. "
        "Espero: peso_tallo_mll_g / peso_tallo_mll / peso_tallo_pred_mll_g / peso_tallo_pred_mll (o peso_tallo_pred)."
    )
col_mll = cand_mll[0]

# Unir universo con ML1 por (bloque_base, grado). Si ML1 tiene variedad, se agrega también.
join_keys = ["bloque_base", "grado"]
if "variedad_canon" in mll.columns and "variedad_canon" in grid.columns:
    mll["variedad_canon"] = _canon_str(mll["variedad_canon"])
    join_keys = ["bloque_base", "variedad_canon", "grado"]

base_cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "rel_pos_final", "day_in_harvest_final", "n_harvest_days_final", "dow", "month", "weekofyear", "area", "tipo_sp", "estado"]
base_cols = [c for c in base_cols if c in grid.columns]
# Para peso/tallo, necesitamos grade. Lo tomamos desde ML1 y replicamos por grado (merge many-to-many controlado)
df = grid[base_cols].drop_duplicates(["ciclo_id", "fecha"]).copy()

# expand grades via ML1 universe (distinct grados por join_keys sin fecha)
mll_small = mll[join_keys + [col_mll]].drop_duplicates(join_keys).copy()
mll_small = mll_small.rename(columns={col_mll: "peso_tallo_mll_g"})
mll_small["peso_tallo_mll_g"] = pd.to_numeric(mll_small["peso_tallo_mll_g"], errors="coerce")

df = df.merge(mll_small, on=[k for k in join_keys if k in df.columns], how="left")

# si join_keys incluye grado pero df no lo tiene, lo agrega desde mll_small en el merge
if "grado" in df.columns:
    df["grado"] = _canon_str(df["grado"])

# reales peso/tallo
real = _load_real_peso_tallo()
real["fecha"] = _to_date(real["fecha"])
real["bloque_base"] = _canon_str(real["bloque_base"])
if "grado" in real.columns:
    real["grado"] = _canon_str(real["grado"])

df = df.merge(real, on=["fecha", "bloque_base", "grado"], how="left")

# clima
clima["fecha"] = _to_date(clima["fecha"])
clima["bloque_base"] = _canon_str(clima["bloque_base"])
clima_cols = [
    "fecha", "bloque_base",
    "gdc_dia",
    "rainfall_mm_dia", "en_lluvia_dia",
    "temp_avg_dia", "solar_energy_j_m2_dia",
    "wind_speed_avg_dia", "wind_run_dia",
]
clima_cols = [c for c in clima_cols if c in clima.columns]
df = df.merge(clima[clima_cols].copy(), on=["fecha", "bloque_base"], how="left")

# Limpieza + target
df["peso_tallo_mll_g"] = pd.to_numeric(df["peso_tallo_mll_g"], errors="coerce")
df["peso_tallo_real_g"] = pd.to_numeric(df["peso_tallo_real_g"], errors="coerce")

# Filtrar filas con señal suficiente

```

```

df = df[df["peso_tallo_m11_g"].notna()].copy()

ratio = (df["peso_tallo_real_g"] + EPS) / (df["peso_tallo_m11_g"] + EPS)
df["log_error_peso"] = np.log(ratio).clip(-CLIP_LOG_ERR, CLIP_LOG_ERR)
df["peso_tallo_ratio"] = np.exp(df["log_error_peso"])

# evitar columnas duplicadas
df = df.loc[:, ~df.columns.duplicated()].copy()

df["created_at"] = pd.Timestamp(datetime.now()).normalize()

OUT_DS.parent.mkdir(parents=True, exist_ok=True)
write_parquet(df, OUT_DS)

print(f"[OK] Wrote dataset: {OUT_DS}")
print(f"      rows={len(df):,} cycles={df['ciclo_id'].nunique():,} "
      f"fecha_range=[{df['fecha'].min().date()}..{df['fecha'].max().date()}]")
print(f"      m11_file_used={m11_path.name} m11_col_used={col_m11}")
print(f"      real_source={'fact_peso_tallo_real_grado_dia' if IN_REAL_PT.exists() else 'kg_real/tallos_rea
l derived'}")

```

```

if __name__ == "__main__":
    main()

```

```

-----
[62/164] FILE: \src\gold\build_ds_share_grado_ml2_v1.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

```

```

from src.common.io import read_parquet, write_parquet

```

```

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

```

```

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
SILVER = DATA / "silver"

```

```

IN_GRID = GOLD / "universe_harvest_grid_ml2.parquet"
IN_PRED = GOLD / "pred_tallos_grado_dia_m11_full.parquet"
IN_REAL = SILVER / "fact_cosecha_real_grado_dia.parquet"
IN_CLIMA = SILVER / "dim_clima_bloque_dia.parquet"

```

```

OUT_DS = GOLD / "ml2_datasets" / "ds_share_grado_ml2_v1.parquet"

```

```

EPS = 1e-6
CLIP_LOG_ERR = 1.2 # ratio ? [0.301..3.320]

```

```

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

```

```

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

```

```

def _resolve_block_base(df: pd.DataFrame) -> pd.DataFrame:
    if "bloque_base" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_base"])
        return df
    if "bloque_padre" in df.columns:

```

```

        df["bloque_base"] = _canon_str(df["bloque_padre"])
        return df
    if "bloque" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque"])
        return df
    raise KeyError("No encuentro columna de bloque: bloque_base / bloque_padre / bloque")

def main() -> None:
    grid = read_parquet(IN_GRID).copy()
    pred = read_parquet(IN_PRED).copy()
    real = read_parquet(IN_REAL).copy()
    clima = read_parquet(IN_CLIMA).copy()

    # ---- Canon grid ----
    grid["fecha"] = _to_date(grid["fecha"])
    grid["bloque_base"] = _canon_str(grid["bloque_base"])
    if "variedad_canon" in grid.columns:
        grid["variedad_canon"] = _canon_str(grid["variedad_canon"])

    # base features from grid (sin duplicar por grado)
    base_cols = [
        "ciclo_id", "fecha", "bloque_base", "variedad_canon",
        "area", "tipo_sp", "estado",
        "rel_pos_final", "day_in_harvest_final", "n_harvest_days_final",
        "harvest_start_final", "harvest_end_final",
        "dow", "month", "weekofyear",
    ]
    base_cols = [c for c in base_cols if c in grid.columns]
    base = grid[base_cols].drop_duplicates(["ciclo_id", "fecha"]).copy()

    # ---- Canon pred ML1 ----
    pred["fecha"] = _to_date(pred["fecha"])
    pred["bloque_base"] = _canon_str(pred["bloque_base"])
    pred["grado"] = _canon_str(pred["grado"])
    if "variedad_canon" in pred.columns:
        pred["variedad_canon"] = _canon_str(pred["variedad_canon"])

    # columnas necesarias
    need_pred = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado",
        "tallos_pred_ml1_grado_dia", "share_grado_ml1"]
    for c in need_pred:
        if c not in pred.columns:
            raise KeyError(f"Falta columna en pred_tallos_grado_dia_ml1_full: {c}")

    pred_small = pred[need_pred].copy()
    pred_small["tallos_pred_ml1_grado_dia"] = pd.to_numeric(pred_small["tallos_pred_ml1_grado_dia"], errors="coerce").fillna(0.0)
    pred_small["share_grado_ml1"] = pd.to_numeric(pred_small["share_grado_ml1"], errors="coerce")

    # ---- Canon real ----
    real["fecha"] = _to_date(real["fecha"])
    real = _resolve_block_base(real)
    real["grado"] = _canon_str(real["grado"])
    if "variedad" in real.columns:
        real["variedad"] = _canon_str(real["variedad"])

    real["tallos_real"] = pd.to_numeric(real["tallos_real"], errors="coerce").fillna(0.0)
    real_g = real.groupby(["fecha", "bloque_base", "grado"], as_index=False).agg(tallos_real=("tallos_real", "sum"))
    tot = real_g.groupby(["fecha", "bloque_base"], as_index=False).agg(tallos_real_dia=("tallos_real", "sum"))
    real_g = real_g.merge(tot, on=["fecha", "bloque_base"], how="left")
    real_g["share_real"] = np.where(real_g["tallos_real_dia"] > 0, real_g["tallos_real"] / real_g["tallos_real_dia"], np.nan)

    # ---- Merge base (grid) con pred (por grado) ----
    df = base.merge(
        pred_small,
        on=["ciclo_id", "fecha", "bloque_base", "variedad_canon"],
        how="inner"
    )

```



```

# ---- Merge share real por (fecha,bloque_base,grado) ----
df = df.merge(real_g[["fecha", "bloque_base", "grado", "tallos_real", "tallos_real_dia", "share_real"]],
              on=["fecha", "bloque_base", "grado"], how="left")

# ---- Merge clima por (fecha,bloque_base) ----
clima["fecha"] = _to_date(clima["fecha"])
clima["bloque_base"] = _canon_str(clima["bloque_base"])
clima_cols = [
    "fecha", "bloque_base",
    "gdc_dia",
    "rainfall_mm_dia", "en_lluvia_dia",
    "temp_avg_dia", "solar_energy_j_m2_dia",
    "wind_speed_avg_dia", "wind_run_dia",
]
clima_cols = [c for c in clima_cols if c in clima.columns]
df = df.merge(clima[clima_cols].copy(), on=["fecha", "bloque_base"], how="left")

# ---- Target residual ----
df["share_grado_ml1"] = pd.to_numeric(df["share_grado_ml1"], errors="coerce")
df["share_real"] = pd.to_numeric(df["share_real"], errors="coerce")

ratio = (df["share_real"] + EPS) / (df["share_grado_ml1"] + EPS)
df["log_error_share"] = np.log(ratio).clip(-CLIP_LOG_ERR, CLIP_LOG_ERR)
df["share_ratio"] = np.exp(df["log_error_share"])

# ---- Limpieza ----
df = df.loc[:, ~df.columns.duplicated()].copy()
df["created_at"] = pd.Timestamp(datetime.now()).normalize()

OUT_DS.parent.mkdir(parents=True, exist_ok=True)
write_parquet(df, OUT_DS)

print(f"[OK] Wrote dataset: {OUT_DS}")
print(f"      rows={len(df):,} cycles={df['ciclo_id'].nunique():,} fecha_range=[{df['fecha'].min().date():,}
.{df['fecha'].max().date():,}]")
print("      pred_col_used=tallos_pred_ml1_grado_dia share_col_used=share_grado_ml1")
print("      target=log_error_share clip=±1.2 ratio_range=[0.301..3.320]")

if __name__ == "__main__":
    main()

-----
[63/164] FILE: \src\gold\build_ds_tallos_curve_ml2_v2.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"

IN_GRID_ML2 = GOLD_DIR / "universe_harvest_grid_ml2.parquet"
IN_REAL = SILVER_DIR / "fact_cosecha_real_grado_dia.parquet"
IN_PRED_ML1 = GOLD_DIR / "pred_tallos_grado_dia_ml1_full.parquet"
IN_CLIMA = SILVER_DIR / "dim_clima_bloque_dia.parquet"

OUT_DS = GOLD_DIR / "ml2_datasets" / "ds_tallos_curve_ml2_v2.parquet"

```

```

EPS = 1.0
CLIP_LOG_ERR = 1.5 # exp(±1.5) ? factor [0.22 .. 4.48]

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def main() -> None:
    grid = read_parquet(IN_GRID_ML2).copy()
    real = read_parquet(IN_REAL).copy()
    pred = read_parquet(IN_PRED_ML1).copy()
    clima = read_parquet(IN_CLIMA).copy()

    # --- GRID ML2 ---
    grid["fecha"] = _to_date(grid["fecha"])
    grid["bloque_base"] = _canon_str(grid["bloque_base"])
    grid["variedad_canon"] = _canon_str(grid["variedad_canon"])
    if "estado" in grid.columns:
        grid["estado"] = _canon_str(grid["estado"])

    base_cols = [
        "ciclo_id", "fecha", "bloque_base", "variedad_canon", "area", "tipo_sp", "estado",
        "rel_pos_final", "day_in_harvest_final", "n_harvest_days_final",
        "harvest_start_final", "harvest_end_final",
        "m11_version",
    ]
    base_cols = [c for c in base_cols if c in grid.columns]
    grid0 = grid[base_cols].drop_duplicates(["ciclo_id", "fecha"]).copy()

    # --- REAL tallos (sum grades) ---
    real["fecha"] = _to_date(real["fecha"])

    # Resolver bloque_base desde columnas posibles
    if "bloque_base" in real.columns:
        real["bloque_base"] = _canon_str(real["bloque_base"])
    elif "bloque_padre" in real.columns:
        real["bloque_base"] = _canon_str(real["bloque_padre"])
    elif "bloque" in real.columns:
        real["bloque_base"] = _canon_str(real["bloque"])
    else:
        raise KeyError(
            "No encuentro columna de bloque en fact_cosecha_real_grado_dia. "
            "Espero una de: bloque_base / bloque_padre / bloque"
        )

    if "variedad" in real.columns:
        real["variedad"] = _canon_str(real["variedad"])

    # real: (fecha, bloque_base, variedad, grado, tallos_real)
    real_agg = (
        real.groupby(["fecha", "bloque_base"], as_index=False)
        .agg(tallos_real_dia=("tallos_real", "sum"))
    )

    # --- PRED ML1 tallos (sum grades) ---
    pred["fecha"] = _to_date(pred["fecha"])
    pred["bloque_base"] = _canon_str(pred["bloque_base"])
    if "variedad_canon" in pred.columns:
        pred["variedad_canon"] = _canon_str(pred["variedad_canon"])

    # pred expected: (fecha, bloque_base, variedad_canon, grado, tallos_pred_*)
    # Detect the predicted column (fallback order)
    # En este proyecto, la columna canónica ML1 a nivel grado-día es:
    # tallos_pred_m11_grado_dia (sumar por grado -> total día)
    if "tallos_pred_m11_grado_dia" in pred.columns:
        pred_col = "tallos_pred_m11_grado_dia"
    elif "tallos_pred_m11_dia" in pred.columns:

```

```

        # fallback (menos preferido)
        pred_col = "tallos_pred_mll_dia"
    else:
        raise KeyError(
            "No encuentro columna de tallos predicha en pred_tallos_grado_dia_mll_full.parquet. "
            "Espero: tallos_pred_mll_grado_dia (preferida) o tallos_pred_mll_dia (fallback)."
        )

    pred_agg = (
        pred.groupby(["fecha", "bloque_base", "variedad_canon"], as_index=False)
        .agg(tallos_pred_mll_dia=(pred_col, "sum"))
    )

    # --- CLIMA diario (ya por bloque_base + fecha) ---
    clima["fecha"] = _to_date(clima["fecha"])
    clima["bloque_base"] = _canon_str(clima["bloque_base"])

    clima_cols = [
        "fecha", "bloque_base",
        "gdc_dia", "gdc_base",
        "rainfall_mm_dia", "horas_lluvia", "en_lluvia_dia",
        "temp_avg_dia", "solar_energy_j_m2_dia",
        "wind_speed_avg_dia", "wind_run_dia",
    ]
    clima_cols = [c for c in clima_cols if c in clima.columns]
    clima0 = clima[clima_cols].copy()

    # --- JOIN ---
    df = (
        grid0
        .merge(pred_agg, on=["fecha", "bloque_base", "variedad_canon"], how="left")
        .merge(real_agg, on=["fecha", "bloque_base"], how="left")
        .merge(clima0, on=["fecha", "bloque_base"], how="left")
    )

    # pyarrow no permite columnas duplicadas (ej: gdc_base repetida)
    df = df.loc[:, ~df.columns.duplicated()].copy()

    df["tallos_pred_mll_dia"] = pd.to_numeric(df["tallos_pred_mll_dia"], errors="coerce").fillna(0.0)
    df["tallos_real_dia"] = pd.to_numeric(df["tallos_real_dia"], errors="coerce").fillna(0.0)

    # Target: log error ratio
    ratio = (df["tallos_real_dia"] + EPS) / (df["tallos_pred_mll_dia"] + EPS)
    df["log_error"] = np.log(ratio).clip(-CLIP_LOG_ERR, CLIP_LOG_ERR)

    # convenience columns
    df["error_ratio"] = np.exp(df["log_error"])

    # Filter: para entrenamiento, necesitamos pred > 0 o real > 0 (para no meter basura total)
    df = df.loc[(df["tallos_pred_mll_dia"] > 0) | (df["tallos_real_dia"] > 0), :].copy()

    df["created_at"] = pd.Timestamp(datetime.now()).normalize()

    OUT_DS.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(df, OUT_DS)

    print(f"[OK] Wrote dataset: {OUT_DS}")
    print(f"      rows={len(df):,} cycles={df['ciclo_id'].nunique():,} "
          f"fecha_range=[{df['fecha'].min().date()}..{df['fecha'].max().date()}]")
    print(f"      pred_col_used={pred_col}")

if __name__ == "__main__":
    main()

```

```

from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

# =====
# INPUTS
# =====
IN_CAJAS_GRADO = Path("data/gold/pred_cajas_grado_dia_mll_full.parquet")
IN_MIX = Path("data/silver/dim_mix_proceso_semana.parquet")

# Hidratación: por fecha_cosecha (que en seed = fecha de cosecha proyectada)
IN_HIDR_FC = Path("data/silver/dim_hidratacion_fecha_cosecha_grado_destino.parquet")

# DH baseline por grado-destino
IN_DH = Path("data/silver/dim_dh_baseline_grado_destino.parquet")

# Merma + ajuste por fecha_post (día de proceso) y destino
IN_MERMA_AJUSTE = Path("data/silver/dim_mermas_ajuste_fecha_post_destino.parquet")

# =====
# OUTPUTS
# =====
OUT_GD_BD = Path("data/gold/pred_poscosecha_seed_grado_dia_bloque_destino.parquet")
OUT_DD = Path("data/gold/pred_poscosecha_seed_dia_destino.parquet")
OUT_DT = Path("data/gold/pred_poscosecha_seed_dia_total.parquet")

# =====
# HELPERS
# =====
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _safe_float(x, default: float) -> float:
    try:
        v = float(x)
        if np.isfinite(v):
            return v
    except Exception:
        pass
    return float(default)

def _semana_ventas_from_fecha_cosecha(fecha: pd.Series) -> pd.Series:
    """
    Debe ser CONSISTENTE con ventas:
    ventas:
        Fecha_Clasificacion = Fecha - 2
        Semana_Ventas = semana_ventas(Fecha_Clasificacion)
        semana_ventas(x): (x + 2) -> %U
    Entonces para fecha de cosecha (equivale a Fecha_Clasificacion):
        Semana_Ventas = semana_ventas(fecha)
    """

```

```

"""
d = pd.to_datetime(fecha, errors="coerce") + pd.Timedelta(days=2)
yy = (d.dt.year.astype("Int64") % 100).astype("Int64")
ww = d.dt.strftime("%U").astype(int)
ww = np.where(ww == 0, 1, ww)
return yy.astype(str).str.zfill(2) + pd.Series(ww).astype(str).str.zfill(2)

def _collapse_sum(df: pd.DataFrame, keys: list[str], val_cols: list[str], name: str) -> pd.DataFrame:
    dup = int(df.duplicated(subset=keys).sum())
    if dup > 0:
        dup_rate = float(dup / max(len(df), 1))
        print(f"[WARN] {name} tiene duplicados por {keys} -> dup_count={dup:,} dup_rate={dup_rate:.4%}")
        print("[WARN] ejemplos dup (top 10):")
        print(df.loc[df.duplicated(subset=keys, keep=False), keys].head(10).to_string(index=False))

        agg = {c: "sum" for c in val_cols if c in df.columns}
        # Mantener cualquier columna extra (first) si existe
        extra = [c for c in df.columns if c not in keys and c not in agg]
        for c in extra:
            agg[c] = "first"

        out = df.groupby(keys, dropna=False, as_index=False).agg(agg)
        print(f"[FIX] {name} colapsado por {keys} (sum) -> rows={len(out):,}")
        return out
    return df

def _collapse_median(df: pd.DataFrame, keys: list[str], cols: list[str], name: str) -> pd.DataFrame:
    agg = {}
    for c in cols:
        if c in df.columns:
            agg[c] = "median"
    # Mantener created_at si existe
    if "created_at" in df.columns:
        agg["created_at"] = "first"
    out = df.groupby(keys, dropna=False, as_index=False).agg(agg)
    return out

# =====
# MAIN
# =====
def main() -> None:
    created_at = pd.Timestamp.utcnow()

    # -----
    # 1) Load cajas ML1 (grado/día)
    # -----
    cajas = read_parquet(IN_CAJAS_GRADO).copy()
    cajas.columns = [str(c).strip() for c in cajas.columns]

    need_cajas = ["fecha", "bloque_base", "variedad_canon", "grado", "cajas_mll_grado_dia"]
    _require(cajas, need_cajas, "pred_cajas_grado_dia_mll_full")

    cajas["fecha"] = _to_date(cajas["fecha"])
    cajas["bloque_base"] = _canon_int(cajas["bloque_base"])
    cajas["grado"] = _canon_int(cajas["grado"])
    cajas["variedad_canon"] = _canon_str(cajas["variedad_canon"])
    cajas["cajas_mll_grado_dia"] = pd.to_numeric(cajas["cajas_mll_grado_dia"], errors="coerce").fillna(0.0)

    key_supply = ["fecha", "bloque_base", "variedad_canon", "grado"]
    cajas = _collapse_sum(cajas, key_supply, ["cajas_mll_grado_dia"], "cajas")

    # -----
    # 2) Load mix por Semana_Ventas
    # -----
    mix = read_parquet(IN_MIX).copy()
    mix.columns = [str(c).strip() for c in mix.columns]

    _require(mix, ["Semana_Ventas", "W_Blanco", "W_Arcoiris", "W_Tinturado"], "dim_mix_proceso_semana")

```

```

mix["Semana_Ventas"] = mix["Semana_Ventas"].astype(str).str.strip()
for c in ["W_Blanco", "W_Arcoiris", "W_Tinturado"]:
    mix[c] = pd.to_numeric(mix[c], errors="coerce")

# DEFAULT row
mix_def = mix[mix["Semana_Ventas"].eq("DEFAULT")].copy()
if mix_def.empty:
    raise ValueError("dim_mix_proceso_semana no trae Semana_Ventas='DEFAULT' (fallback obligatorio).")
mix_def_row = mix_def.iloc[0]
def_w = {
    "W_Blanco": _safe_float(mix_def_row["W_Blanco"], 0.80),
    "W_Arcoiris": _safe_float(mix_def_row["W_Arcoiris"], 0.10),
    "W_Tinturado": _safe_float(mix_def_row["W_Tinturado"], 0.10),
}
sdef = def_w["W_Blanco"] + def_w["W_Arcoiris"] + def_w["W_Tinturado"]
if sdef > 0:
    def_w = {k: v / sdef for k, v in def_w.items()}
else:
    def_w = {"W_Blanco": 0.80, "W_Arcoiris": 0.10, "W_Tinturado": 0.10}

# mapeo destino (proceso)
DESTS = [
    ("BLANCO", "W_Blanco"),
    ("ARCOIRIS", "W_Arcoiris"),
    ("TINTURADO", "W_Tinturado"),
]

# -----
# 3) Attach Semana_Ventas to supply + join mix (fallback DEFAULT)
# -----
supply = cajas.copy()
supply["Semana_Ventas"] = _semana_ventas_from_fecha_cosecha(supply["fecha"])

mix_take = mix[mix["Semana_Ventas"].ne("DEFAULT")].copy()
mix_take = mix_take.drop_duplicates(subset=["Semana_Ventas"], keep="last")

supply = supply.merge(mix_take, on="Semana_Ventas", how="left")

# fill missing with DEFAULT medians
for _, wcol in DESTS:
    supply[wcol] = supply[wcol].fillna(def_w[wcol])

# renormalize weights per row (defensive)
ws = supply[[w for _, w in DESTS]].sum(axis=1)
ws = ws.replace(0, np.nan)
for _, wcol in DESTS:
    supply[wcol] = np.where(ws.notna(), supply[wcol] / ws, def_w[wcol])

cov_mix = float(supply["W_Blanco"].notna().mean())
print(f"[MIX] coverage semana->mix (W_Blanco notna): {cov_mix:.4f}")

# -----
# 4) Split supply by destino using weights (mass-balance exact)
# -----
chunks = []
for dest, wcol in DESTS:
    sub = supply[key_supply + ["Semana_Ventas", "cajas_mll_grado_dia", wcol]].copy()
    sub = sub.rename(columns={wcol: "w_dest"})
    sub["destino"] = dest
    sub["cajas_split_grado_dia"] = sub["cajas_mll_grado_dia"].astype(float) * sub["w_dest"].astype(float)
    chunks.append(sub)

split = pd.concat(chunks, ignore_index=True)
split["destino"] = _canon_str(split["destino"])

# Mass-balance check: sum_split == cajas original
chk = (
    split.groupby(key_supply, dropna=False, as_index=False)
        .agg(sum_split=("cajas_split_grado_dia", "sum"))
    .merge(
        supply[key_supply + ["cajas_mll_grado_dia"]],
        on=key_supply,
    )

```

```

        how="left",
        validate="1:1",
    )
)
chk["abs_diff"] = (chk["sum_split"] - chk["cajas_mll_grado_dia"]).abs()
max_abs = float(chk["abs_diff"].max()) if len(chk) else 0.0
print(f"[CHECK] split mass-balance max_abs_diff={max_abs:.10f}")
if max_abs > 1e-9:
    print("[DEBUG] ejemplos mismatch (top 20):")
    print(
        chk.sort_values("abs_diff", ascending=False)
        .head(20)
        .to_string(index=False)
    )
    raise ValueError("[FATAL] split por destino no conserva cajas.")

# -----
# 5) Attach DH baseline (grado,destino) -> fecha_post_pred
# -----
dh = read_parquet(IN_DH).copy()
dh.columns = [str(c).strip() for c in dh.columns]
_require(dh, ["grado", "destino", "dh_dias_med"], "dim_dh_baseline_grado_destino")

dh["grado"] = _canon_int(dh["grado"])
dh["destino"] = _canon_str(dh["destino"])
dh["dh_dias_med"] = pd.to_numeric(dh["dh_dias_med"], errors="coerce")

dh2 = dh.groupby(["grado", "destino"], dropna=False, as_index=False).agg(
    dh_dias_med=("dh_dias_med", "median")
)

split = split.merge(dh2, on=["grado", "destino"], how="left")

dh_global = _safe_float(np.nanmedian(dh2["dh_dias_med"].values), default=7.0)
split["dh_dias"] = (
    pd.to_numeric(split["dh_dias_med"], errors="coerce")
    .fillna(dh_global)
    .round()
    .astype("Int64")
    .clip(lower=0, upper=30)
)
split["fecha_post_pred"] = _to_date(split["fecha"]) + pd.to_timedelta(split["dh_dias"].astype(float), unit="D")

# -----
# 6) Hidratación seed: prefer match por fecha_cosecha=fecha,grado,destino; fallback medianas
# -----
hidr = read_parquet(IN_HIDR_FC).copy()
hidr.columns = [str(c).strip() for c in hidr.columns]
_require(hidr, ["fecha_cosecha", "grado", "destino", "factor_hidr"], "dim_hidratacion_fecha_cosecha_grado_destino")

hidr["fecha_cosecha"] = _to_date(hidr["fecha_cosecha"])
hidr["grado"] = _canon_int(hidr["grado"])
hidr["destino"] = _canon_str(hidr["destino"])
hidr["factor_hidr"] = pd.to_numeric(hidr["factor_hidr"], errors="coerce")

# collapse to 1 row per (fecha_cosecha, grado, destino)
hidr_fc = _collapse_median(hidr, ["fecha_cosecha", "grado", "destino"], ["factor_hidr"], "hidr_fc")

# baseline by (grado,destino)
hidr_gd = _collapse_median(hidr, ["grado", "destino"], ["factor_hidr"], "hidr_gd")

# join fc first
split = split.merge(
    hidr_fc.rename(columns={"factor_hidr": "factor_hidr_fc"}),
    left_on=["fecha", "grado", "destino"],
    right_on=["fecha_cosecha", "grado", "destino"],
    how="left",
)

# join gd baseline

```

```

split = split.merge(
    hydr_gd.rename(columns={"factor_hidr": "factor_hidr_gd"}),
    on=["grado", "destino"],
    how="left",
)

hydr_global = _safe_float(np.nanmedian(hidr["factor_hidr"].values), default=1.45)

split["factor_hidr_seed"] = (
    pd.to_numeric(split["factor_hidr_fc"], errors="coerce")
    .fillna(pd.to_numeric(split["factor_hidr_gd"], errors="coerce"))
    .fillna(hidr_global)
    .clip(lower=0.60, upper=3.00)
)

miss_hidr_fc = int(pd.to_numeric(split["factor_hidr_fc"], errors="coerce").isna().sum())
print(f"[CHECK] miss_hidr_fc={miss_hidr_fc:,} (sin match por fecha_cosecha; se usó baseline grado-destino
o global)")

# -----
# 7) Merma + ajuste seed por fecha_post_pred + destino (fallback med destino -> med global)
# *** CAMBIO CLAVE: usar factor_ajuste y MULTIPLICAR (NO dividir por ajuste) ***
# -----
mer = read_parquet(IN_MERMA_AJUSTE).copy()
mer.columns = [str(c).strip() for c in mer.columns]
_require(mer, ["fecha_post", "destino", "factor_desp", "factor_ajuste"], "dim_mermas_ajuste_fecha_post_des
tino")

mer["fecha_post"] = _to_date(mer["fecha_post"])
mer["destino"] = _canon_str(mer["destino"])
mer["factor_desp"] = pd.to_numeric(mer["factor_desp"], errors="coerce")
mer["factor_ajuste"] = pd.to_numeric(mer["factor_ajuste"], errors="coerce")

mer2 = _collapse_median(mer, ["fecha_post", "destino"], ["factor_desp", "factor_ajuste"], "mer2")

split = split.merge(
    mer2.rename(columns={"fecha_post": "fecha_post_key"}),
    left_on=["fecha_post_pred", "destino"],
    right_on=["fecha_post_key", "destino"],
    how="left",
)

med_dest = (
    mer2.groupby("destino", dropna=False, as_index=False)
    .agg(
        factor_desp_med=("factor_desp", "median"),
        factor_ajuste_med=("factor_ajuste", "median"),
    )
)

split = split.merge(med_dest, on="destino", how="left")

fd_global = _safe_float(np.nanmedian(mer2["factor_desp"].values), default=0.70)
fa_global = _safe_float(np.nanmedian(mer2["factor_ajuste"].values), default=1.00)

split["factor_desp_seed"] = (
    pd.to_numeric(split["factor_desp"], errors="coerce")
    .fillna(pd.to_numeric(split["factor_desp_med"], errors="coerce"))
    .fillna(fd_global)
    .clip(lower=0.05, upper=1.00)
)

# baseline por medianas (destino -> global). NO usamos 1.0 salvo que sea imposible calcular mediana.
split["factor_ajuste_seed"] = (
    pd.to_numeric(split["factor_ajuste"], errors="coerce")
    .fillna(pd.to_numeric(split["factor_ajuste_med"], errors="coerce"))
    .fillna(fa_global)
    .fillna(1.0) # debería ser ~0% si mer2 tiene data; queda como último guardarraíl
    .clip(lower=0.50, upper=2.00)
)

# -----
# 8) Cajas poscosecha seed (macro)

```



```

# -----
split["cajas_post_seed"] = (
    split["cajas_split_grado_dia"].astype(float)
    * split["factor_hidr_seed"].astype(float)
    * split["factor_desp_seed"].astype(float)
    * split["factor_ajuste_seed"].astype(float) # <-- CAMBIO: MULTIPLICA
)

split["created_at"] = created_at

# -----
# 9) Output 1: grado-dia-bloque-destino
# -----
out_gd_bd = split[
    [
        "fecha",
        "fecha_post_pred",
        "bloque_base",
        "variedad_canon",
        "grado",
        "Semana_Ventas",
        "destino",
        "cajas_mll_grado_dia",
        "cajas_split_grado_dia",
        "dh_dias",
        "factor_hidr_seed",
        "factor_desp_seed",
        "factor_ajuste_seed",
        "cajas_post_seed",
        "created_at",
    ]
].copy()

out_gd_bd = out_gd_bd.sort_values(
    ["fecha", "bloque_base", "variedad_canon", "grado", "destino"]
).reset_index(drop=True)

write_parquet(out_gd_bd, OUT_GD_BD)
print(f"OK -> {OUT_GD_BD} | rows={len(out_gd_bd):,}")

# -----
# 10) Output 2: día-destino (agregado)
# -----
out_dd = (
    out_gd_bd.groupby(["fecha", "fecha_post_pred", "destino"], dropna=False, as_index=False)
    .agg(
        cajas_mll_dia=("cajas_mll_grado_dia", "sum"),
        cajas_split_dia=("cajas_split_grado_dia", "sum"),
        cajas_post_seed=("cajas_post_seed", "sum"),
    )
)

out_dd["created_at"] = created_at
out_dd = out_dd.sort_values(["fecha", "destino"]).reset_index(drop=True)

write_parquet(out_dd, OUT_DD)
print(f"OK -> {OUT_DD} | rows={len(out_dd):,}")

# -----
# 11) Output 3: día-total (agregado)
# -----
out_dt = (
    out_dd.groupby(["fecha", "fecha_post_pred"], dropna=False, as_index=False)
    .agg(
        cajas_mll_dia=("cajas_mll_dia", "sum"),
        cajas_split_dia=("cajas_split_dia", "sum"),
        cajas_post_seed=("cajas_post_seed", "sum"),
    )
)

out_dt["created_at"] = created_at
out_dt = out_dt.sort_values(["fecha"]).reset_index(drop=True)

write_parquet(out_dt, OUT_DT)

```

```

print(f"OK -> {OUT_DT} | rows={len(out_dt):,}")

if __name__ == "__main__":
    main()

-----
[65/164] FILE: \src\gold\build_pred_kg_cajas_grado_dia_m11_full.py
-----

from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

# =====
# INPUTS / OUTPUTS
# =====
IN_TALLOS = Path("data/gold/pred_tallos_grado_dia_m11_full.parquet")
IN_PESO = Path("data/gold/pred_peso_tallo_grado_m11.parquet")

OUT_KG_GRADO = Path("data/gold/pred_kg_grado_dia_m11_full.parquet")
OUT_CAJAS_GRADO = Path("data/gold/pred_cajas_grado_dia_m11_full.parquet")
OUT_CAJAS_DIA = Path("data/gold/pred_cajas_dia_m11_full.parquet")

# NUEVO: agregados operacionales (sin ciclo_id)
OUT_KG_GRADO_AGG = Path("data/gold/pred_kg_grado_dia_m11_agg.parquet")
OUT_CAJAS_GRADO_AGG = Path("data/gold/pred_cajas_grado_dia_m11_agg.parquet")

# =====
# HELPERS
# =====
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    missing = [c for c in cols if c not in df.columns]
    if missing:
        raise ValueError(f"{name}: faltan columnas {missing}. Disponibles={list(df.columns)}")

def _detect_tallos_cols(tallos: pd.DataFrame) -> tuple[str, str]:
    cand_baseline = [c for c in ["tallos_pred_baseline_grado_dia", "tallos_baseline_grado_dia", "tallos_pred_baseline"] if c in tallos.columns]
    cand_m11 = [c for c in ["tallos_pred_m11_grado_dia", "tallos_m11_grado_dia", "tallos_pred_m11"] if c in tallos.columns]

    if not cand_baseline:
        cand_baseline = [c for c in tallos.columns if ("tallos" in c and "baseline" in c)]
    if not cand_m11:
        cand_m11 = [c for c in tallos.columns if ("tallos" in c and "m11" in c)]

    if not cand_baseline or not cand_m11:
        raise ValueError(
            "No pude detectar columnas tallos baseline/m11 en pred_tallos_grado_dia_m11_full. "
            f"Cols={list(tallos.columns)}"
        )
    return cand_baseline[0], cand_m11[0]

def main() -> None:
    created_at = pd.Timestamp.utcnow()

```

```

tallos = read_parquet(IN_TALLOS).copy()
peso = read_parquet(IN_PESO).copy()

tallos.columns = [str(c).strip() for c in tallos.columns]
peso.columns = [str(c).strip() for c in peso.columns]

# ciclo_id es opcional pero recomendado
has_ciclo = "ciclo_id" in tallos.columns

keys = ["fecha", "bloque_base", "variedad_canon", "grado"]
if has_ciclo:
    keys_ciclo = ["ciclo_id"] + keys
else:
    keys_ciclo = keys

_require(tallos, keys_ciclo, "pred_tallos_grado_dia_m11_full")
_require(peso, keys + ["peso_tallo_baseline_g", "peso_tallo_m11_g"], "pred_peso_tallo_grado_m11")

# Normalización
tallos["fecha"] = _to_date(tallos["fecha"])
tallos["bloque_base"] = _canon_int(tallos["bloque_base"])
tallos["grado"] = _canon_int(tallos["grado"])
tallos["variedad_canon"] = tallos["variedad_canon"].astype(str).str.upper().str.strip()
if has_ciclo:
    tallos["ciclo_id"] = tallos["ciclo_id"].astype(str)

peso["fecha"] = _to_date(peso["fecha"])
peso["bloque_base"] = _canon_int(peso["bloque_base"])
peso["grado"] = _canon_int(peso["grado"])
peso["variedad_canon"] = peso["variedad_canon"].astype(str).str.upper().str.strip()

col_tallos_baseline, col_tallos_m11 = _detect_tallos_cols(tallos)

# Peso: dedup por keys (sin ciclo_id, porque peso se modela a ese grano)
dup = int(peso.duplicated(subset=keys).sum())
if dup > 0:
    print(f"[WARN] peso duplicado por {keys}; colapso mean.")
    agg = {"peso_tallo_baseline_g": "mean", "peso_tallo_m11_g": "mean"}
    if "factor_peso_tallo_m11" in peso.columns:
        agg["factor_peso_tallo_m11"] = "mean"
    if "m11_version" in peso.columns:
        agg["m11_version"] = "first"
    peso = peso.groupby(keys, dropna=False, as_index=False).agg(agg)

# Merge
df = tallos.merge(peso, on=keys, how="left", validate="m:1")

# Num
for c in [col_tallos_baseline, col_tallos_m11, "peso_tallo_baseline_g", "peso_tallo_m11_g"]:
    df[c] = pd.to_numeric(df[c], errors="coerce")

# KG
df["kg_baseline_grado_dia"] = (df[col_tallos_baseline] * df["peso_tallo_baseline_g"]) / 1000.0
df["kg_m11_grado_dia"] = (df[col_tallos_m11] * df["peso_tallo_m11_g"]) / 1000.0

# Cajas (10 kg/caja)
df["cajas_baseline_grado_dia"] = df["kg_baseline_grado_dia"] / 10.0
df["cajas_m11_grado_dia"] = df["kg_m11_grado_dia"] / 10.0

df["created_at"] = created_at

# =====
# OUTPUT (con ciclo_id si existe)
# =====
out_cols = []
if has_ciclo:
    out_cols.append("ciclo_id")
out_cols += [
    "fecha", "bloque_base", "variedad_canon", "grado",
    col_tallos_baseline, col_tallos_m11,
    "peso_tallo_baseline_g", "peso_tallo_m11_g",

```

```

    "kg_baseline_grado_dia", "kg_mll_grado_dia",
    "cajas_baseline_grado_dia", "cajas_mll_grado_dia",
    "created_at",
]

out = df[out_cols].sort_values(
    (["ciclo_id"] if has_ciclo else []) + ["bloque_base", "variedad_canon", "fecha", "grado"]
).reset_index(drop=True)

# Split outputs
out_kg = out[[c for c in out.columns if not c.startswith("cajas_)]].copy()
out_cajas = out[[c for c in out.columns if not c.startswith("kg_") and c not in [col_tallos_baseline, col_tallos_mll]]].copy()

write_parquet(out_kg, OUT_KG_GRADO)
print(f"OK -> {OUT_KG_GRADO} | rows={len(out_kg):,}")

write_parquet(out_cajas, OUT_CAJAS_GRADO)
print(f"OK -> {OUT_CAJAS_GRADO} | rows={len(out_cajas):,}")

# =====
# OUTPUT agregados operacionales (sin ciclo_id)
# =====
grp = ["fecha", "bloque_base", "variedad_canon", "grado"]

out_kg_agg = (
    out.groupby(grp, dropna=False, as_index=False)
    .agg(
        kg_baseline_grado_dia=("kg_baseline_grado_dia", "sum"),
        kg_mll_grado_dia=("kg_mll_grado_dia", "sum"),
        cajas_baseline_grado_dia=("cajas_baseline_grado_dia", "sum"),
        cajas_mll_grado_dia=("cajas_mll_grado_dia", "sum"),
    )
)
out_kg_agg["created_at"] = created_at

write_parquet(out_kg_agg[["fecha", "bloque_base", "variedad_canon", "grado", "kg_baseline_grado_dia", "kg_mll_g
rado_dia", "created_at"]], OUT_KG_GRADO_AGG)
print(f"OK -> {OUT_KG_GRADO_AGG} | rows={len(out_kg_agg):,}")

write_parquet(out_kg_agg[["fecha", "bloque_base", "variedad_canon", "grado", "cajas_baseline_grado_dia", "cajas
_mll_grado_dia", "created_at"]], OUT_CAJAS_GRADO_AGG)
print(f"OK -> {OUT_CAJAS_GRADO_AGG} | rows={len(out_kg_agg):,}")

# =====
# OUTPUT día (sin ciclo_id, como antes)
# =====
grp_day = ["fecha", "bloque_base", "variedad_canon"]
out_day = (
    out_kg_agg.groupby(grp_day, dropna=False, as_index=False)
    .agg(
        cajas_baseline_dia=("cajas_baseline_grado_dia", "sum"),
        cajas_mll_dia=("cajas_mll_grado_dia", "sum"),
        kg_baseline_dia=("kg_baseline_grado_dia", "sum"),
        kg_mll_dia=("kg_mll_grado_dia", "sum"),
    )
)
out_day["created_at"] = created_at

out_day = out_day.sort_values(["bloque_base", "variedad_canon", "fecha"]).reset_index(drop=True)
write_parquet(out_day, OUT_CAJAS_DIA)
print(f"OK -> {OUT_CAJAS_DIA} | rows={len(out_day):,}")

cov_peso = float(df["peso_tallo_mll_g"].notna().mean())
cov_tallos = float(df[col_tallos_mll].notna().mean())
print(f"[CHECK] coverage tallos_mll notna: {cov_tallos:.4f}")
print(f"[CHECK] coverage peso_tallo_mll_g notna: {cov_peso:.4f}")
if has_ciclo:
    dup_oper = int(out_kg_agg.duplicated(subset=grp).sum())
    print(f"[CHECK] duplicates en agregado operacional (debe ser 0): {dup_oper:,}")

```

```

if __name__ == "__main__":
    main()

-----
[66/164] FILE: \src\gold\build_pred_kg_cajas_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"

# ---- INPUTS ML2 (backtest) ----
IN_TALLOS ML2_BT = EVAL / "backtest_pred_tallos_grado_dia_ml2_final.parquet"
IN_PESO ML2_BT = EVAL / "backtest_pred_peso_tallo_grado_dia_ml2_final.parquet"

# ---- INPUTS ML2 (prod - si luego los materializas en gold) ----
IN_TALLOS ML2_PROD = GOLD / "pred_tallos_grado_dia_ml2_full.parquet" # opcional futuro
IN_PESO ML2_PROD = GOLD / "pred_peso_tallo_grado_dia_ml2_full.parquet" # opcional futuro

# ---- INPUTS ML1 (para ?heredar? conversión a cajas) ----
IN_KG ML1 = GOLD / "pred_kg_grado_dia_ml1_full.parquet"
IN_CAJAS ML1 = GOLD / "pred_cajas_grado_dia_ml1_full.parquet"

# ---- OUTPUTS ML2 ----
OUT_KG_GRADO = GOLD / "pred_kg_grado_dia_ml2_full.parquet"
OUT_KG_DIA = GOLD / "pred_kg_dia_ml2_full.parquet"
OUT_CAJAS_GRADO = GOLD / "pred_cajas_grado_dia_ml2_full.parquet"
OUT_CAJAS_DIA = GOLD / "pred_cajas_dia_ml2_full.parquet"

KEYS = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"]

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _pick_col(df: pd.DataFrame, candidates: list[str]) -> str:
    for c in candidates:
        if c in df.columns:
            return c
    raise KeyError(f"No encuentro ninguna de estas columnas: {candidates}. Tengo: {list(df.columns)}")

def main(mode: str = "backtest") -> None:
    mode = (mode or "backtest").strip().lower()
    if mode not in ("backtest", "prod"):
        raise ValueError("--mode debe ser backtest o prod")

    # ---- Load ML2 ----
    if mode == "backtest":
        tallos = read_parquet(IN_TALLOS ML2_BT).copy()
        peso = read_parquet(IN_PESO ML2_BT).copy()

```

```

else:
    tallos = read_parquet(IN_TALLOS_ML2_PROD).copy()
    peso = read_parquet(IN_PESO_ML2_PROD).copy()

# Canon keys
for df in (tallos, peso):
    df["fecha"] = _to_date(df["fecha"])
    df["bloque_base"] = _canon_str(df["bloque_base"])
    df["variedad_canon"] = _canon_str(df["variedad_canon"])
    df["grado"] = _canon_str(df["grado"])

# ---- resolve ML2 columns ----
col_tallos_ml2 = _pick_col(
    tallos,
    ["tallos_final_grado_dia", "tallos_ml2_grado_dia", "tallos_pred_ml2_grado_dia"]
)
# peso backtest normalmente trae algo tipo peso_tallo_final_g
col_peso_ml2 = _pick_col(
    peso,
    ["peso_tallo_final_g", "peso_tallo_ml2_g", "peso_tallo_pred_ml2_g"]
)

tallos_use = tallos[KEYS + [col_tallos_ml2]].copy().rename(columns={col_tallos_ml2: "tallos_ml2_grado_dia"})
peso_use = peso[KEYS + [col_peso_ml2]].copy().rename(columns={col_peso_ml2: "peso_tallo_ml2_g"})

# ---- Merge ML2 tallos + peso ----
df = tallos_use.merge(peso_use, on=KEYS, how="left")

df["tallos_ml2_grado_dia"] = pd.to_numeric(df["tallos_ml2_grado_dia"], errors="coerce").fillna(0.0)
df["peso_tallo_ml2_g"] = pd.to_numeric(df["peso_tallo_ml2_g"], errors="coerce")

# KG ML2
df["kg_ml2_grado_dia"] = df["tallos_ml2_grado_dia"] * df["peso_tallo_ml2_g"] / 1000.0
df.loc[df["kg_ml2_grado_dia"] < 0, "kg_ml2_grado_dia"] = 0.0

# ---- Load ML1 kg/cajas for conversion ----
kg1 = read_parquet(IN_KG_ML1).copy()
cajas1 = read_parquet(IN_CAJAS_ML1).copy()

for df1 in (kg1, cajas1):
    df1["fecha"] = _to_date(df1["fecha"])
    df1["bloque_base"] = _canon_str(df1["bloque_base"])
    df1["variedad_canon"] = _canon_str(df1["variedad_canon"])
    df1["grado"] = _canon_str(df1["grado"])

col_kg_ml1 = _pick_col(
    kg1,
    ["kg_pred_ml1_grado_dia", "kg_ml1_grado_dia", "kg_pred_grado_dia", "kg_pred_ml1"]
)
col_cajas_ml1 = _pick_col(
    cajas1,
    ["cajas_pred_ml1_grado_dia", "cajas_ml1_grado_dia", "cajas_pred_grado_dia", "cajas_pred_ml1"]
)

kg1_use = kg1[KEYS + [col_kg_ml1]].copy().rename(columns={col_kg_ml1: "kg_ml1_grado_dia"})
cajas1_use = cajas1[KEYS + [col_cajas_ml1]].copy().rename(columns={col_cajas_ml1: "cajas_ml1_grado_dia"})

df = df.merge(kg1_use, on=KEYS, how="left").merge(cajas1_use, on=KEYS, how="left")

df["kg_ml1_grado_dia"] = pd.to_numeric(df["kg_ml1_grado_dia"], errors="coerce").fillna(0.0)
df["cajas_ml1_grado_dia"] = pd.to_numeric(df["cajas_ml1_grado_dia"], errors="coerce").fillna(0.0)

# Cajas ML2 heredando conversión ML1
eps = 1e-9
ratio = np.where(df["kg_ml1_grado_dia"] > 0, df["kg_ml2_grado_dia"] / (df["kg_ml1_grado_dia"] + eps), 0.0)
df["cajas_ml2_grado_dia"] = df["cajas_ml1_grado_dia"] * ratio
df.loc[df["cajas_ml2_grado_dia"] < 0, "cajas_ml2_grado_dia"] = 0.0

# ---- Build outputs (grado/día) ----
out_kg_grado = df[KEYS + ["kg_ml2_grado_dia"]].copy()
out_cajas_grado = df[KEYS + ["cajas_ml2_grado_dia"]].copy()

```

```

created_at = pd.Timestamp(datetime.now()).normalize()
out_kg_grado["created_at"] = created_at
out_cajas_grado["created_at"] = created_at

# ---- Aggregate to día (sum grados) ----
keys_day = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
out_kg_day = out_kg_grado.groupby(keys_day, as_index=False).agg(kg_ml2_dia=("kg_ml2_grado_dia", "sum"))
out_cajas_day = out_cajas_grado.groupby(keys_day, as_index=False).agg(cajas_ml2_dia=("cajas_ml2_grado_dia",
, "sum"))
out_kg_day["created_at"] = created_at
out_cajas_day["created_at"] = created_at

GOLD.mkdir(parents=True, exist_ok=True)
write_parquet(out_kg_grado, OUT_KG_GRADO)
write_parquet(out_kg_day, OUT_KG_DIA)
write_parquet(out_cajas_grado, OUT_CAJAS_GRADO)
write_parquet(out_cajas_day, OUT_CAJAS_DIA)

print(f"[OK] Wrote: {OUT_KG_GRADO} rows={len(out_kg_grado)}")
print(f"[OK] Wrote: {OUT_KG_DIA} rows={len(out_kg_day)}")
print(f"[OK] Wrote: {OUT_CAJAS_GRADO} rows={len(out_cajas_grado)}")
print(f"[OK] Wrote: {OUT_CAJAS_DIA} rows={len(out_cajas_day)}")
print(f"      mode={mode} tallos_col={col_tallos_ml2} peso_col={col_peso_ml2}")
print(f"      kg_ml1_col={col_kg_ml1} cajas_ml1_col={col_cajas_ml1}")

if __name__ == "__main__":
    import argparse

    p = argparse.ArgumentParser()
    p.add_argument("--mode", default="backtest", choices=["backtest", "prod"])
    args = p.parse_args()
    main(mode=args.mode)

-----
[67/164] FILE: \src\gold\build_pred_poscosecha_ml1_views.py
-----

from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

IN_PATH = Path("data/gold/pred_poscosecha_ml1_full_grado_dia_bloque_destino.parquet")

OUT_DIA_BLOQUE_DEST = Path("data/gold/pred_poscosecha_ml1_dia_bloque_destino.parquet")
OUT_DIA_DEST = Path("data/gold/pred_poscosecha_ml1_dia_destino.parquet")
OUT_DIA_TOTAL = Path("data/gold/pred_poscosecha_ml1_dia_total.parquet")

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

```

```

def main() -> None:
    created_at = pd.Timestamp.utcnow()

    df = read_parquet(IN_PATH).copy()
    df.columns = [str(c).strip() for c in df.columns]

    # ---- llaves mínimas esperadas
    req = ["fecha", "bloque_base", "variedad_canon", "grado", "destino"]
    _require(df, req, "pred_poscosecha_mll_full_grado_dia_bloque_destino")

    # ---- normalización
    df["fecha"] = _to_date(df["fecha"])
    df["bloque_base"] = _canon_int(df["bloque_base"])
    df["grado"] = _canon_int(df["grado"])
    df["variedad_canon"] = _canon_str(df["variedad_canon"])
    df["destino"] = _canon_str(df["destino"])

    # =====
    # cajas iniciales: DEBE SER SPLIT (porque ya está a grano destino)
    # =====
    cajas_in_col = None
    for c in [
        "cajas_split_grado_dia",      # <- correcto
        "cajas_split",                # <- alias posible
        "cajas_iniciales",            # <- si algún step ya lo renombró
        "cajas_in",                   # <- alias
        "cajas_mll_grado_dia",        # <- fallback (NO ideal)
    ]:
        if c in df.columns:
            cajas_in_col = c
            break

    if cajas_in_col is None:
        raise ValueError(
            "No encontré columna de cajas iniciales a grano destino. "
            "Esperaba cajas_split_grado_dia (o similar). "
        )

    # Si por error solo existe cajas_mll_grado_dia, avisamos (porque puede reintroducir mismatch)
    if cajas_in_col == "cajas_mll_grado_dia":
        print("[WARN] Usando cajas_mll_grado_dia como iniciales (no hay split). Puede reintroducir mismatch.")

    # =====
    # cajas_postcosecha_mll: si falta, derivarla desde factores
    # =====
    if "cajas_postcosecha_mll" not in df.columns:
        fh = pd.to_numeric(df.get("factor_hidr_mll"), errors="coerce").fillna(1.0)
        fd = pd.to_numeric(df.get("factor_desp_mll"), errors="coerce").fillna(1.0)
        aj = pd.to_numeric(df.get("ajuste_mll"), errors="coerce").replace(0, np.nan).fillna(1.0)
        ci = pd.to_numeric(df[cajas_in_col], errors="coerce").fillna(0.0)
        df["cajas_postcosecha_mll"] = ci * fh * fd / aj

    # =====
    # fecha_post_pred_mll: si no existe, construir desde dh
    # =====
    if "fecha_post_pred_mll" not in df.columns:
        dh_col = None
        for c in ["dh_dias_mll", "dh_dias_pred_mll", "dh_dias_pred", "dh_dias"]:
            if c in df.columns:
                dh_col = c
                break
        if dh_col is not None:
            df["fecha_post_pred_mll"] = df["fecha"] + pd.to_timedelta(
                pd.to_numeric(df[dh_col], errors="coerce").fillna(0).astype(int),
                unit="D",
            )
        else:
            df["fecha_post_pred_mll"] = pd.NaT

    # =====
    # 1) DIA + BLOQUE + DESTINO

```



```

# =====
g1 = ["fecha", "fecha_post_pred_ml1", "bloque_base", "destino"]

out1 = (
    df.groupby(g1, dropna=False, as_index=False)
    .agg(
        cajas_iniciales=(cajas_in_col, "sum"),
        cajas_postcosecha_ml1=("cajas_postcosecha_ml1", "sum"),
    )
)
out1["created_at"] = created_at
out1 = out1.sort_values(["bloque_base", "destino", "fecha"]).reset_index(drop=True)

write_parquet(out1, OUT_DIA_BLOQUE_DEST)
print(f"OK -> {OUT_DIA_BLOQUE_DEST} | rows={len(out1):,} | cajas_base={cajas_in_col}")

# =====
# 2) DIA + DESTINO
# =====
g2 = ["fecha", "fecha_post_pred_ml1", "destino"]
out2 = (
    out1.groupby(g2, dropna=False, as_index=False)
    .agg(
        cajas_iniciales=("cajas_iniciales", "sum"),
        cajas_postcosecha_ml1=("cajas_postcosecha_ml1", "sum"),
    )
)
out2["created_at"] = created_at
out2 = out2.sort_values(["destino", "fecha"]).reset_index(drop=True)

write_parquet(out2, OUT_DIA_DEST)
print(f"OK -> {OUT_DIA_DEST} | rows={len(out2):,}")

# =====
# 3) DIA TOTAL
# =====
g3 = ["fecha", "fecha_post_pred_ml1"]
out3 = (
    out2.groupby(g3, dropna=False, as_index=False)
    .agg(
        cajas_iniciales=("cajas_iniciales", "sum"),
        cajas_postcosecha_ml1=("cajas_postcosecha_ml1", "sum"),
    )
)
out3["created_at"] = created_at
out3 = out3.sort_values(["fecha"]).reset_index(drop=True)

write_parquet(out3, OUT_DIA_TOTAL)
print(f"OK -> {OUT_DIA_TOTAL} | rows={len(out3):,}")

# =====
# SANITY: mass-balance interno (sum destinos = total)
# =====
chk = (
    out2.groupby(["fecha", "fecha_post_pred_ml1"], dropna=False)["cajas_iniciales"]
    .sum()
    .reset_index()
    .rename(columns={"cajas_iniciales": "cajas_iniciales_sum_dest"})
)
chk = chk.merge(
    out3[["fecha", "fecha_post_pred_ml1", "cajas_iniciales"]],
    on=["fecha", "fecha_post_pred_ml1"],
    how="left",
).rename(columns={"cajas_iniciales": "cajas_iniciales_total"})

chk["abs_diff"] = (chk["cajas_iniciales_sum_dest"] - chk["cajas_iniciales_total"]).abs()
max_abs = float(chk["abs_diff"].max()) if len(chk) else float("nan")
print(f"[CHECK] mass-balance dia (sum destinos vs total) | max_abs_diff={max_abs:.12f}")
if max_abs > 1e-6:
    raise ValueError("[FATAL] Mass-balance día no cuadra. Revisa agregaciones.")

```

```

if __name__ == "__main__":
    main()

-----
[68/164] FILE: \src\gold\build_pred_poscosecha_ml2_final_views.py
-----

from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"

IN_FULL = GOLD / "pred_poscosecha_ml2_ajuste_grado_dia_bloque_destino_final.parquet"

OUT_FULL = GOLD / "pred_poscosecha_ml2_final_full_grado_dia_bloque_destino.parquet"
OUT_DBD = GOLD / "pred_poscosecha_ml2_final_dia_bloque_destino.parquet"
OUT_DD = GOLD / "pred_poscosecha_ml2_final_dia_destino.parquet"
OUT_DT = GOLD / "pred_poscosecha_ml2_final_dia_total.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def main() -> None:
    df = read_parquet(IN_FULL).copy()
    df.columns = [str(c).strip() for c in df.columns]

    _require(
        df,
        ["fecha", "destino", "cajas_split_grado_dia", "factor_hidr_final", "factor_desp_final", "factor_ajuste_
_final"],
        "pred_poscosecha_ml2_ajuste...final",
    )

    df["fecha"] = _to_date(df["fecha"])
    df["cajas_split_grado_dia"] = pd.to_numeric(df["cajas_split_grado_dia"], errors="coerce").fillna(0.0)
    for c in ["factor_hidr_final", "factor_desp_final", "factor_ajuste_final"]:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(1.0)

    df["cajas_postcosecha_ml2_final"] = (
        df["cajas_split_grado_dia"].astype(float)
        * df["factor_hidr_final"].astype(float)
        * df["factor_desp_final"].astype(float)
        * df["factor_ajuste_final"].astype(float)
    )

    df["created_at"] = pd.Timestamp.utcnow()

    write_parquet(df, OUT_FULL)
    print(f"[OK] Wrote: {OUT_FULL} rows={len(df):,}")

    # día-bloque-destino
    keys_dbd = ["fecha", "bloque_base", "destino"]

```

```

cols_av = [c for c in keys_dbd if c in df.columns]
if len(cols_av) == 3:
    out_dbd = (
        df.groupby(keys_dbd, dropna=False, as_index=False)
        .agg(
            cajas_split=("cajas_split_grado_dia", "sum"),
            cajas_post_ml2=("cajas_postcosecha_ml2_final", "sum"),
        )
    )
    out_dbd["created_at"] = pd.Timestamp.utcnow()
    write_parquet(out_dbd, OUT_DBD)
    print(f"[OK] Wrote: {OUT_DBD} rows={len(out_dbd):,}")
else:
    print("[WARN] No se pudo escribir OUT_DBD (faltan columnas de bloque_base).")

# día-destino
out_dd = (
    df.groupby(["fecha", "destino"], dropna=False, as_index=False)
    .agg(
        cajas_split=("cajas_split_grado_dia", "sum"),
        cajas_post_ml2=("cajas_postcosecha_ml2_final", "sum"),
    )
)
out_dd["created_at"] = pd.Timestamp.utcnow()
write_parquet(out_dd, OUT_DD)
print(f"[OK] Wrote: {OUT_DD} rows={len(out_dd):,}")

# día-total
out_dt = (
    out_dd.groupby(["fecha"], dropna=False, as_index=False)
    .agg(
        cajas_split=("cajas_split", "sum"),
        cajas_post_ml2=("cajas_post_ml2", "sum"),
    )
)
out_dt["created_at"] = pd.Timestamp.utcnow()
write_parquet(out_dt, OUT_DT)
print(f"[OK] Wrote: {OUT_DT} rows={len(out_dt):,}")

if __name__ == "__main__":
    main()

```

```

-----
[69/164] FILE: \src\gold\build_pred_poscosecha_ml2_seed_mix_grado_dia.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

```

```

from src.common.io import read_parquet, write_parquet

```

```

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

```

```

ROOT = _project_root()
DATA = ROOT / "data"
SILVER = DATA / "silver"
GOLD = DATA / "gold"

```

```

# INPUTS
IN_CAJAS_GRADO = GOLD / "pred_cajas_grado_dia_ml2_full.parquet"
IN_MIX = SILVER / "dim_mix_proceso_semana.parquet"

```

```

# OUTPUTS (ML2 seed)
OUT_GD_BD = GOLD / "pred_poscosecha_ml2_seed_grado_dia_bloque_destino.parquet"

```

```

OUT_DD = GOLD / "pred_poscosecha_ml2_seed_dia_destino.parquet"
OUT_DT = GOLD / "pred_poscosecha_ml2_seed_dia_total.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise KeyError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _safe_float(x, default: float) -> float:
    try:
        v = float(x)
        if np.isfinite(v):
            return v
    except Exception:
        pass
    return float(default)

def _semana_ventas_from_fecha_cosecha(fecha: pd.Series) -> pd.Series:
    """
    CONSISTENTE con tu lógica de ventas (Fecha_Clasificacion = Fecha - 2; semana_ventas(x): (x+2)->%U).
    Para fecha de cosecha (equivale a Fecha_Clasificacion):
        Semana_Ventas = semana_ventas(fecha)
    """
    d = pd.to_datetime(fecha, errors="coerce") + pd.Timedelta(days=2)
    yy = (d.dt.year.astype("Int64") % 100).astype("Int64")
    ww = d.dt.strftime("%U").astype(int)
    ww = np.where(ww == 0, 1, ww)
    return yy.astype(str).str.zfill(2) + pd.Series(ww).astype(str).str.zfill(2)

def _collapse_sum(df: pd.DataFrame, keys: list[str], val_cols: list[str], name: str) -> pd.DataFrame:
    dup = int(df.duplicated(subset=keys).sum())
    if dup > 0:
        agg = {c: "sum" for c in val_cols if c in df.columns}
        extra = [c for c in df.columns if c not in keys and c not in agg]
        for c in extra:
            agg[c] = "first"
        out = df.groupby(keys, dropna=False, as_index=False).agg(agg)
        print(f"[WARN] {name}: duplicados por {keys} -> colapsado sum -> rows={len(out):,}")
        return out
    return df

def _resolve_cajas_col(df: pd.DataFrame) -> str:
    # prioridad: columna final ML2
    candidates = [
        "cajas_ml2_grado_dia",
        "cajas_final_grado_dia",
        "cajas_pred_ml2_grado_dia",
        "cajas_grado_dia_ml2",
        # fallback: si tu builder dejó el mismo nombre ml1 (no ideal, pero posible)
        "cajas_ml1_grado_dia",
    ]
    for c in candidates:
        if c in df.columns:
            return c
    raise ValueError(

```

```

        "No encuentro columna de cajas en pred_cajas_grado_dia_ml2_full.parquet. "
        f"Probé: {candidates}. Disponibles={list(df.columns)}"
    )

def main(as_of_date: str | None = None) -> None:
    created_at = pd.Timestamp.utcnow()

    # as_of_date default = hoy-1 (operativo)
    if as_of_date is None:
        as_of = pd.Timestamp.now().normalize() - pd.Timedelta(days=1)
    else:
        as_of = pd.to_datetime(as_of_date, errors="coerce").normalize()
        if pd.isna(as_of):
            raise ValueError(f"as_of_date inválida: {as_of_date}")

    # -----
    # 1) Load cajas ML2 (grado/día) -> supply
    # -----
    cajas = read_parquet(IN_CAJAS_GRADO).copy()
    cajas.columns = [str(c).strip() for c in cajas.columns]

    need = ["fecha", "bloque_base", "variedad_canon", "grado"]
    _require(cajas, need, "pred_cajas_grado_dia_ml2_full")

    cajas["fecha"] = _to_date(cajas["fecha"])
    cajas["bloque_base"] = _canon_int(cajas["bloque_base"])
    cajas["grado"] = _canon_int(cajas["grado"])
    cajas["variedad_canon"] = _canon_str(cajas["variedad_canon"])

    cajas_col = _resolve_cajas_col(cajas)
    cajas["cajas_campo_ml2_grado_dia"] = pd.to_numeric(cajas[cajas_col], errors="coerce").fillna(0.0)

    # filtro operativo hoy-1
    cajas = cajas.loc[cajas["fecha"].notna() & (cajas["fecha"] <= as_of)].copy()

    key_supply = ["fecha", "bloque_base", "variedad_canon", "grado"]
    cajas = _collapse_sum(cajas[key_supply + ["cajas_campo_ml2_grado_dia"]], key_supply, ["cajas_campo_ml2_grado_dia"], "cajas_ml2")

    # -----
    # 2) Load mix por Semana_Ventas
    # -----
    mix = read_parquet(IN_MIX).copy()
    mix.columns = [str(c).strip() for c in mix.columns]
    _require(mix, ["Semana_Ventas", "W_Blanco", "W_Arcoiris", "W_Tinturado"], "dim_mix_proceso_semana")

    mix["Semana_Ventas"] = mix["Semana_Ventas"].astype(str).str.strip()
    for c in ["W_Blanco", "W_Arcoiris", "W_Tinturado"]:
        mix[c] = pd.to_numeric(mix[c], errors="coerce")

    # DEFAULT obligatorio
    mix_def = mix[mix["Semana_Ventas"].eq("DEFAULT")].copy()
    if mix_def.empty:
        raise ValueError("dim_mix_proceso_semana no trae Semana_Ventas='DEFAULT' (fallback obligatorio).")
    r = mix_def.iloc[0]
    def_w = {
        "W_Blanco": _safe_float(r["W_Blanco"], 0.80),
        "W_Arcoiris": _safe_float(r["W_Arcoiris"], 0.10),
        "W_Tinturado": _safe_float(r["W_Tinturado"], 0.10),
    }
    sdef = def_w["W_Blanco"] + def_w["W_Arcoiris"] + def_w["W_Tinturado"]
    def_w = {k: (v / sdef if sdef > 0 else v) for k, v in def_w.items()}

    DESTS = [
        ("BLANCO", "W_Blanco"),
        ("ARCOIRIS", "W_Arcoiris"),
        ("TINTURADO", "W_Tinturado"),
    ]

    # -----
    # 3) Semana_Ventas + join mix (fallback DEFAULT)

```

```

# -----
supply = cajas.copy()
supply["Semana_Ventas"] = _semana_ventas_from_fecha_cosecha(supply["fecha"])

mix_take = mix[mix["Semana_Ventas"].ne("DEFAULT")].drop_duplicates(subset=["Semana_Ventas"], keep="last").
copy()
supply = supply.merge(mix_take, on="Semana_Ventas", how="left")

for _, wcol in DESTS:
    supply[wcol] = supply[wcol].fillna(def_w[wcol])

# renormalize defensivo
ws = supply[[w for _, w in DESTS]].sum(axis=1).replace(0, np.nan)
for _, wcol in DESTS:
    supply[wcol] = np.where(ws.notna(), supply[wcol] / ws, def_w[wcol])

# -----
# 4) Split por destino (mass-balance exact)
# -----
chunks = []
for dest, wcol in DESTS:
    sub = supply[key_supply + ["Semana_Ventas", "cajas_campo_ml2_grado_dia", wcol]].copy()
    sub = sub.rename(columns={wcol: "w_dest"})
    sub["destino"] = dest
    sub["cajas_split_grado_dia"] = sub["cajas_campo_ml2_grado_dia"].astype(float) * sub["w_dest"].astype(f
load)
    chunks.append(sub)

split = pd.concat(chunks, ignore_index=True)
split["destino"] = _canon_str(split["destino"])

chk = (
    split.groupby(key_supply, dropna=False, as_index=False)
        .agg(sum_split=("cajas_split_grado_dia", "sum"))
        .merge(supply[key_supply + ["cajas_campo_ml2_grado_dia"]], on=key_supply, how="left", validate="1:
1")
)
chk["abs_diff"] = (chk["sum_split"] - chk["cajas_campo_ml2_grado_dia"]).abs()
max_abs = float(chk["abs_diff"].max()) if len(chk) else 0.0
print(f"[CHECK] split mass-balance max_abs_diff={max_abs:.12f}")
if max_abs > 1e-9:
    raise ValueError("[FATAL] split por destino no conserva cajas (mass-balance roto).")

split["as_of_date"] = as_of
split["cajas_base_col_used"] = cajas_col
split["created_at"] = created_at

# -----
# Outputs
# -----
out_gd_bd = split[
    [
        "fecha",
        "bloque_base",
        "variedad_canon",
        "grado",
        "Semana_Ventas",
        "destino",
        "cajas_campo_ml2_grado_dia",
        "cajas_split_grado_dia",
        "as_of_date",
        "cajas_base_col_used",
        "created_at",
    ]
].copy()

GOLD.mkdir(parents=True, exist_ok=True)
write_parquet(out_gd_bd, OUT_GD_BD)
print(f"[OK] Wrote: {OUT_GD_BD} rows={len(out_gd_bd):}, as_of_date={as_of.date()}")

out_dd = (
    out_gd_bd.groupby(["fecha", "destino"], dropna=False, as_index=False)

```

```

        .agg(
            cajas_campo_ml2_dia=("cajas_campo_ml2_grado_dia", "sum"),
            cajas_split_dia=("cajas_split_grado_dia", "sum"),
        )
    )
    out_dd["as_of_date"] = as_of
    out_dd["created_at"] = created_at
    write_parquet(out_dd, OUT_DD)
    print(f"[OK] Wrote: {OUT_DD} rows={len(out_dd):,}")

    out_dt = (
        out_dd.groupby(["fecha"], dropna=False, as_index=False)
        .agg(
            cajas_campo_ml2_dia=("cajas_campo_ml2_dia", "sum"),
            cajas_split_dia=("cajas_split_dia", "sum"),
        )
    )
    out_dt["as_of_date"] = as_of
    out_dt["created_at"] = created_at
    write_parquet(out_dt, OUT_DT)
    print(f"[OK] Wrote: {OUT_DT} rows={len(out_dt):,}")

if __name__ == "__main__":
    main()

```

```

-----
[70/164] FILE: \src\gold\build_pred_poscosecha_ml2_views_from_full.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
import pandas as pd

```

```

from src.common.io import read_parquet, write_parquet

```

```

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

```

```

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"

```

```

IN_FULL = GOLD / "pred_poscosecha_ml2_full_grado_dia_bloque_destino.parquet"

```

```

OUT_DBD = GOLD / "pred_poscosecha_ml2_ml1_dia_bloque_destino.parquet"
OUT_DD = GOLD / "pred_poscosecha_ml2_ml1_dia_destino.parquet"
OUT_DT = GOLD / "pred_poscosecha_ml2_ml1_dia_total.parquet"

```

```

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

```

```

def main() -> None:
    df = read_parquet(IN_FULL).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha", "fecha_post_pred_ml1", "bloque_base", "variedad_canon", "grado", "destino", "cajas_postco
secha_ml1"}
    miss = need - set(df.columns)
    if miss:
        raise KeyError(f"Full ML2 sin columnas: {sorted(miss)}")

    df["fecha"] = _to_date(df["fecha"])
    df["fecha_post_pred_ml1"] = _to_date(df["fecha_post_pred_ml1"])

    # 1) día-bloque-destino
    out_dbd = (

```

```

df.groupby(["fecha_post_pred_m11", "bloque_base", "destino"], dropna=False, as_index=False)
    .agg(
        cajas_postcosecha_m11=("cajas_postcosecha_m11", "sum"),
        cajas_split=("cajas_split_grado_dia", "sum"),
    )
)
out_dbd["created_at"] = pd.Timestamp.utcnow()
write_parquet(out_dbd, OUT_DBD)
print(f"[OK] Wrote: {OUT_DBD} rows={len(out_dbd):,}")

# 2) día-destino
out_dd = (
    out_dbd.groupby(["fecha_post_pred_m11", "destino"], dropna=False, as_index=False)
        .agg(
            cajas_postcosecha_m11=("cajas_postcosecha_m11", "sum"),
            cajas_split=("cajas_split", "sum"),
        )
)
out_dd["created_at"] = pd.Timestamp.utcnow()
write_parquet(out_dd, OUT_DD)
print(f"[OK] Wrote: {OUT_DD} rows={len(out_dd):,}")

# 3) día-total
out_dt = (
    out_dd.groupby(["fecha_post_pred_m11"], dropna=False, as_index=False)
        .agg(
            cajas_postcosecha_m11=("cajas_postcosecha_m11", "sum"),
            cajas_split=("cajas_split", "sum"),
        )
)
out_dt["created_at"] = pd.Timestamp.utcnow()
write_parquet(out_dt, OUT_DT)
print(f"[OK] Wrote: {OUT_DT} rows={len(out_dt):,}")

if __name__ == "__main__":
    main()

-----
[71/164] FILE: \src\gold\build_pred_tallos_grado_dia_m11.py
-----

from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

IN_CURVA = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
IN_DIST_ML1 = Path("data/gold/pred_dist_grado_m11.parquet")
OUT_PATH = Path("data/gold/pred_tallos_grado_dia_m11.parquet")

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def main() -> None:
    created_at = pd.Timestamp.utcnow()

    curva = read_parquet(IN_CURVA).copy()
    dist = read_parquet(IN_DIST_ML1).copy()

    # -----
    # Normalización llaves/fechas

```



```

# -----
curva["fecha"] = _to_date(curva["fecha"])
curva["bloque_base"] = _canon_int(curva["bloque_base"])
curva["variedad_canon"] = curva["variedad_canon"].astype(str)

need_curva = {"fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia"}
miss = need_curva - set(curva.columns)
if miss:
    raise ValueError(f"features_curva_cosecha_bloque_dia sin columnas: {sorted(miss)}")

# ?? Universo operativo: SOLO donde existe baseline diario (curva)
curva = curva[curva["tallos_pred_baseline_dia"].notna()].copy()
curva = curva.drop_duplicates(subset=["fecha", "bloque_base", "variedad_canon"])

# Dist (por grado)
dist["fecha"] = _to_date(dist["fecha"])
dist["bloque_base"] = _canon_int(dist["bloque_base"])
dist["grado"] = _canon_int(dist["grado"])
dist["variedad_canon"] = dist["variedad_canon"].astype(str)

need_dist = {"ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_m11", "share_grad
o_baseline"}
miss2 = need_dist - set(dist.columns)
if miss2:
    raise ValueError(f"pred_dist_grado_m11 sin columnas: {sorted(miss2)}")

dist = dist.drop_duplicates(subset=["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"])

# -----
# 1) Determinar ciclo_id por llave diaria de curva
# -----
# dist trae ciclo_id; curva puede traerlo o no
# Creamos un map (fecha,bloque,variedad)->ciclo_id usando dist (first)
map_ciclo = (
    dist.groupby(["fecha", "bloque_base", "variedad_canon"], as_index=False)
        .agg(ciclo_id=("ciclo_id", "first"))
)

curva2 = curva.merge(
    map_ciclo,
    on=["fecha", "bloque_base", "variedad_canon"],
    how="left",
)

# Si hay días sin ciclo_id, no abortamos, pero los marcamos (igual se puede operar sin ciclo_id)
if "ciclo_id" in curva2.columns:
    if curva2["ciclo_id"].isna().any():
        miss_c = float(curva2["ciclo_id"].isna().mean())
        print(f"[WARN] curva sin ciclo_id en {miss_c:.4f} de filas. Se mantiene, pero revisa si esperas ci
clo_id completo.")
    else:
        print(f"[WARN] curva no tiene ciclo_id tras el mapeo. Se generará salida sin ciclo_id (operable igual).
")

# -----
# 2) Expandir curva a grados usando dist como universo de grados por variedad
# -----
# Universo de grados por variedad_canon (desde dist)
vg = dist[["variedad_canon", "grado"]].drop_duplicates()

# Expandir curva (día/bloque/variedad) a (día/bloque/variedad/grado)
base = curva2.merge(vg, on="variedad_canon", how="left")

# -----
# 3) Traer shares (baseline y m11) desde dist
# -----
key_cols = ["fecha", "bloque_base", "variedad_canon", "grado"]

# Para no perder ciclo_id, lo traemos a nivel key+grado desde dist.
dist_take = dist[["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_baseline", "
share_grado_m11"]].copy()

```

```

out = base.merge(
    dist_take,
    on=["fecha", "bloque_base", "variedad_canon", "grado"],
    how="left",
    suffixes=("", "_d"),
)

# Si ciclo_id en out quedó NaN y venía en curva2, lo completamos
if "ciclo_id_x" in out.columns and "ciclo_id_y" in out.columns:
    out["ciclo_id"] = out["ciclo_id_x"].combine_first(out["ciclo_id_y"])
    out = out.drop(columns=["ciclo_id_x", "ciclo_id_y"])
elif "ciclo_id" not in out.columns and "ciclo_id_d" in out.columns:
    out = out.rename(columns={"ciclo_id_d": "ciclo_id"})

# -----
# 4) Fallback + renormalización de shares por día/bloque/variedad
# -----
out["share_grado_baseline"] = pd.to_numeric(out["share_grado_baseline"], errors="coerce")
out["share_grado_mll"] = pd.to_numeric(out["share_grado_mll"], errors="coerce")

# Si falta Mll -> usar baseline
out["share_grado_mll_eff"] = out["share_grado_mll"].where(out["share_grado_mll"].notna(), out["share_grado_
_baseline"])

# Si baseline falta (raro) -> 0
out["share_grado_baseline_eff"] = out["share_grado_baseline"].fillna(0)
out["share_grado_mll_eff"] = out["share_grado_mll_eff"].fillna(0)

# Clip >=0
out["share_grado_baseline_eff"] = out["share_grado_baseline_eff"].clip(lower=0)
out["share_grado_mll_eff"] = out["share_grado_mll_eff"].clip(lower=0)

grp = ["fecha", "bloque_base", "variedad_canon"]
s_b = out.groupby(grp, dropna=False)["share_grado_baseline_eff"].transform("sum")
s_m = out.groupby(grp, dropna=False)["share_grado_mll_eff"].transform("sum")
eps = 1e-12

out["share_grado_baseline_eff"] = np.where(s_b > eps, out["share_grado_baseline_eff"] / s_b, np.nan)
out["share_grado_mll_eff"] = np.where(s_m > eps, out["share_grado_mll_eff"] / s_m, np.nan)

# -----
# 5) Tallos por grado día
# -----
out["tallos_pred_baseline_dia"] = pd.to_numeric(out["tallos_pred_baseline_dia"], errors="coerce")

out["tallos_pred_baseline_grado_dia"] = (out["tallos_pred_baseline_dia"] * out["share_grado_baseline_eff"]
).clip(lower=0)
out["tallos_pred_mll_grado_dia"] = (out["tallos_pred_baseline_dia"] * out["share_grado_mll_eff"]).clip(low
er=0)

out["created_at"] = created_at

# -----
# Output
# -----
cols = [
    "ciclo_id" if "ciclo_id" in out.columns else None,
    "fecha",
    "bloque_base",
    "variedad_canon",
    "grado",
    "tallos_pred_baseline_dia",
    "share_grado_baseline", # original (puede venir NaN si no matcheó dist)
    "share_grado_mll", # original (puede venir NaN si no matcheó dist)
    "tallos_pred_baseline_grado_dia",
    "tallos_pred_mll_grado_dia",
    "mll_version" if "mll_version" in out.columns else None,
    "created_at",
]

cols = [c for c in cols if c is not None and c in out.columns]

```

```

out = out[cols].sort_values(["bloque_base", "variedad_canon", "fecha", "grado"]).reset_index(drop=True)

write_parquet(out, OUT_PATH)
print(f"OK -> {OUT_PATH} | rows={len(out):,}")

cov = float(out["tallos_pred_baseline_dia"].notna().mean())
print(f"coverage tallos_pred_baseline_dia notna: {cov:.4f}")

# sanity: sumas por día ? total
grp2 = ["fecha", "bloque_base", "variedad_canon"]
s_mll = out.groupby(grp2, dropna=False)["tallos_pred_mll_grado_dia"].sum()
t = out.groupby(grp2, dropna=False)["tallos_pred_baseline_dia"].first()
diff = (s_mll - t).abs()
print(f"sanity | abs(sum(grado)-total) p50/p95/max: {diff.quantile(0.50):.6f} / {diff.quantile(0.95):.6f} / {diff.max():.6f}")

if __name__ == "__main__":
    main()

-----
[72/164] FILE: \src\gold\build_pred_tallos_grado_dia_mll_full.py
-----
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

# =====
# Paths
# =====
IN_GRID = Path("data/gold/universe_harvest_grid_mll.parquet")
IN_MAESTRO = Path("data/silver/fact_ciclo_maestro.parquet")
IN_CURVA = Path("data/gold/pred_factor_curva_mll.parquet")
IN_DIST = Path("data/gold/pred_dist_grado_mll.parquet")
DIM_VAR = Path("data/silver/dim_variedad_canon.parquet")

OUT_PATH = Path("data/gold/pred_tallos_grado_dia_mll_full.parquet")

# =====
# Helpers
# =====
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _prep_dim_var(dim_var: pd.DataFrame) -> pd.DataFrame:
    need = {"variedad_raw", "variedad_canon"}
    miss = need - set(dim_var.columns)
    if miss:
        raise ValueError(f"dim_variedad_canon.parquet sin columnas: {sorted(miss)}")

```

```

dv = dim_var.copy()
dv["variedad_raw_norm"] = _canon_str(dv["variedad_raw"])
dv["variedad_canon_norm"] = _canon_str(dv["variedad_canon"])
return dv[["variedad_raw_norm", "variedad_canon_norm"]].drop_duplicates()

def _attach_variedad_canon_always(df: pd.DataFrame, dv: pd.DataFrame) -> pd.DataFrame:
    out = df.copy()

    if "variedad" in out.columns:
        raw = out["variedad"]
    elif "variedad_canon" in out.columns:
        raw = out["variedad_canon"]
    else:
        raise ValueError("DF no tiene 'variedad' ni 'variedad_canon' para canonizar.")

    out["__var_raw_norm"] = _canon_str(raw)

    out = out.merge(
        dv,
        left_on="__var_raw_norm",
        right_on="variedad_raw_norm",
        how="left",
    )

    # fallback mínimo (debería coincidir con dim)
    fallback = {
        "XLENCE": "XL",
        "XL": "XL",
        "CLOUD": "CLO",
        "CLO": "CLO",
    }

    canon = out["variedad_canon_norm"]
    canon = canon.fillna(out["__var_raw_norm"].map(fallback))
    canon = canon.fillna(out["__var_raw_norm"])

    out["variedad_canon"] = canon

    return out.drop(
        columns=["variedad_raw_norm", "variedad_canon_norm", "__var_raw_norm"],
        errors="ignore",
    )

def _renormalize_positive(df: pd.DataFrame, col: str, group_cols: list[str]) -> pd.DataFrame:
    out = df.copy()
    out[col] = pd.to_numeric(out[col], errors="coerce").fillna(0.0)
    out[col] = out[col].clip(lower=0.0)
    s = out.groupby(group_cols, dropna=False)[col].transform("sum")
    out[col] = np.where(s > 0, out[col] / s, 0.0)
    return out

def _anti_join(left: pd.DataFrame, right: pd.DataFrame, on: list[str]) -> pd.DataFrame:
    rkeys = right[on].drop_duplicates()
    tmp = left[on].merge(rkeys, on=on, how="left", indicator=True)
    return left.loc[tmp["_merge"].eq("left_only").to_numpy()].copy()

# =====
# Main
# =====
def main() -> None:
    created_at = pd.Timestamp.utcnow()

    grid = read_parquet(IN_GRID).copy()
    maestro = read_parquet(IN_MAESTRO).copy()
    curva = read_parquet(IN_CURVA).copy()
    dist = read_parquet(IN_DIST).copy()
    dim_var = read_parquet(DIM_VAR).copy()

```

```

dv = _prep_dim_var(dim_var)

# -----
# Requisitos mínimos
# -----
_require(grid, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_mll")
_require(maestro, ["ciclo_id", "tallos_proy"], "fact_ciclo_maestro")
_require(curva, ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "factor_curva_mll"], "pred_factor_c
urva_mll")
_require(dist, ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_baseline", "sh
are_grado_mll"], "pred_dist_grado_mll")

# -----
# Canon
# -----
grid["ciclo_id"] = grid["ciclo_id"].astype(str)
grid["fecha"] = _to_date(grid["fecha"])
grid["bloque_base"] = _canon_int(grid["bloque_base"])
grid = _attach_variedad_canon_always(grid, dv)

maestro["ciclo_id"] = maestro["ciclo_id"].astype(str)
maestro["tallos_proy"] = pd.to_numeric(maestro["tallos_proy"], errors="coerce").fillna(0.0)

curva["ciclo_id"] = curva["ciclo_id"].astype(str)
curva["fecha"] = _to_date(curva["fecha"])
curva["bloque_base"] = _canon_int(curva["bloque_base"])
curva = _attach_variedad_canon_always(curva, dv)

dist["ciclo_id"] = dist["ciclo_id"].astype(str)
dist["fecha"] = _to_date(dist["fecha"])
dist["bloque_base"] = _canon_int(dist["bloque_base"])
dist["grado"] = _canon_int(dist["grado"])
dist = _attach_variedad_canon_always(dist, dv)

# -----
# Definir universo base (grid) + tallos_proy (driver absoluto)
# -----
key_dia = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]

base = grid[key_dia].drop_duplicates().copy()
if base.duplicated(subset=key_dia).any():
    raise ValueError("[FATAL] El grid tiene duplicados por key_dia; eso no debe ocurrir.")

# Join tallos_proy por ciclo
m2 = maestro[["ciclo_id", "tallos_proy"]].drop_duplicates("ciclo_id")
base = base.merge(m2, on="ciclo_id", how="left")

if base["tallos_proy"].isna().any():
    n = int(base["tallos_proy"].isna().sum())
    raise ValueError(f"[FATAL] {n:,} filas del grid quedaron sin tallos_proy (maestro incompleto).")

# -----
# Curva: unir y renormalizar por ciclo -> w(d)
# -----
curva_take = curva[key_dia] + [c for c in ["factor_curva_mll", "factor_curva_mll_raw", "mll_version"] if c
in curva.columns].drop_duplicates(subset=key_dia)

dia = base.merge(curva_take, on=key_dia, how="left")

# Diagnóstico de cobertura curva
miss_curva = dia["factor_curva_mll"].isna()
if miss_curva.any():
    nmiss = int(miss_curva.sum())
    anti = _anti_join(base, curva_take, key_dia)
    print(f"[FATAL] Curva no cubre el universo: faltan {nmiss:,} filas.")
    print("Ejemplos faltantes (top 20):")
    print(anti[key_dia].head(20).to_string(index=False))
    raise ValueError("Cobertura de curva incompleta. Arregla llaves/universo, no uses fallback.")

dia["factor_curva_mll"] = pd.to_numeric(dia["factor_curva_mll"], errors="coerce").fillna(0.0).clip(lower=0
.0)

```

```

# w(d) por ciclo (usamos key por ciclo_id, porque el universo ya es por ciclo)
# Si quieres más estricto: agrupar por (ciclo_id) solamente.
s = dia.groupby(["ciclo_id"], dropna=False)["factor_curva_mll"].transform("sum")
zero = s <= 0
if zero.any():
    nbad = int(zero.sum())
    bad_cycles = dia.loc[zero, "ciclo_id"].drop_duplicates().head(20).tolist()
    raise ValueError(f"[FATAL] Curva suma 0 en {nbad:,} filas (ciclos con curva inválida). Ej ciclos: {bad_cycles}")

dia["w_d"] = dia["factor_curva_mll"] / s

# Tallos diarios ML1 (mass-balance garantizado)
dia["tallos_pred_mll_dia"] = dia["tallos_proy"].astype(float) * dia["w_d"].astype(float)

# Baseline diario ?determinístico razonable? (uniforme) solo para comparación
cnt = dia.groupby(["ciclo_id"], dropna=False)["fecha"].transform("count").astype(float)
dia["tallos_pred_baseline_dia"] = np.where(cnt > 0, dia["tallos_proy"].astype(float) / cnt, 0.0)

# Check mass-balance por ciclo
cyc = dia.groupby("ciclo_id", dropna=False).agg(
    proy=("tallos_proy", "max"),
    mll_sum=("tallos_pred_mll_dia", "sum"),
    base_sum=("tallos_pred_baseline_dia", "sum"),
).reset_index()
cyc["abs_diff_mll"] = (cyc["proy"] - cyc["mll_sum"]).abs()
max_abs = float(cyc["abs_diff_mll"].max()) if len(cyc) else float("nan")
print(f"[CHECK] ciclo mass-balance ML1 vs tallos_proy | max abs diff: {max_abs:.12f}")
if max_abs > 1e-6:
    raise ValueError("[FATAL] Mass-balance ML1 no cierra (no debería pasar con w(d)).")

# -----
# Dist grado: asegurar shares y renormalizar por día
# -----
dist["share_grado_baseline"] = pd.to_numeric(dist["share_grado_baseline"], errors="coerce")
dist["share_grado_mll"] = pd.to_numeric(dist["share_grado_mll"], errors="coerce")
dist["share_grado_mll"] = dist["share_grado_mll"].where(dist["share_grado_mll"].notna(), dist["share_grado_baseline"])

grp_dist = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
dist = _renormalize_positive(dist, "share_grado_baseline", grp_dist)
dist = _renormalize_positive(dist, "share_grado_mll", grp_dist)

# Diagnóstico de cobertura dist
dist_keys = dist[grp_dist].drop_duplicates()
dia_keys = dia[grp_dist].drop_duplicates()
miss_dist = _anti_join(dia_keys, dist_keys, grp_dist)
if len(miss_dist) > 0:
    print(f"[FATAL] Dist grado no cubre el universo: faltan {len(miss_dist):,} grupos día.")
    print("Ejemplos faltantes (top 20):")
    print(miss_dist.head(20).to_string(index=False))
    raise ValueError("Cobertura de dist_grado incompleta. Arregla llaves/universo.")

# -----
# Expand a grado
# -----
out = dist.merge(
    dia[grp_dist + ["tallos_proy", "tallos_pred_baseline_dia", "tallos_pred_mll_dia", "factor_curva_mll"]]
+ [c for c in ["factor_curva_mll_raw", "mll_version"] if c in dia.columns],
    on=grp_dist,
    how="left",
)

if out["tallos_pred_mll_dia"].isna().any():
    raise ValueError("[FATAL] merge dist x dia dejó NaNs en tallos_pred_mll_dia. No debe pasar.")

out["tallos_pred_baseline_grado_dia"] = out["tallos_pred_baseline_dia"] * out["share_grado_baseline"]
out["tallos_pred_mll_grado_dia"] = out["tallos_pred_mll_dia"] * out["share_grado_mll"]

# Check por día: sum grado = tallos_dia
g = grp_dist
sum_mll = out.groupby(g, dropna=False)["tallos_pred_mll_grado_dia"].sum().rename("sum_grado_mll").reset_in

```

```

dex()
chk = dia[g + ["tallos_pred_mll_dia"]].merge(sum_mll, on=g, how="left")
eps = 1e-6
mismatch = (chk["sum_grado_mll"] - chk["tallos_pred_mll_dia"]).abs() > eps
print(f"[CHECK] % grupos mismatch (sum_grado_mll vs dia): {float(mismatch.mean()):.4%}")

out["created_at"] = created_at

final_cols = [
    "ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado",
    "tallos_proy",
    "tallos_pred_baseline_dia",
    "tallos_pred_mll_dia",
    "factor_curva_mll",
    "factor_curva_mll_raw",
    "mll_version",
    "share_grado_baseline",
    "share_grado_mll",
    "tallos_pred_baseline_grado_dia",
    "tallos_pred_mll_grado_dia",
    "created_at",
]
final_cols = [c for c in final_cols if c in out.columns]

out = out[final_cols].sort_values(["ciclo_id", "bloque_base", "variedad_canon", "fecha", "grado"]).reset_index(drop=True)

write_parquet(out, OUT_PATH)
fmin = pd.to_datetime(out["fecha"].min()).date() if len(out) else None
fmax = pd.to_datetime(out["fecha"].max()).date() if len(out) else None
print(f"OK -> {OUT_PATH} | rows={len(out):,} | fecha_min={fmin} fecha_max={fmax}")

if __name__ == "__main__":
    main()

-----
[73/164] FILE: \src\gold\build_pred_tallos_ml2_full.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"

# Backtest source (lo que ya tienes)
IN_TALLOS_ML2_BT = EVAL / "backtest_pred_tallos_grado_dia_ml2_final.parquet"

# Output GOLD (vista final)
OUT_TALLOS_GRADO = GOLD / "pred_tallos_grado_dia_ml2_full.parquet"
OUT_TALLOS_DIA = GOLD / "pred_tallos_dia_ml2_full.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

```

```

def main() -> None:
    df = read_parquet(IN_TALLOS_ML2_BT).copy()

    # Canon
    df["fecha"] = _to_date(df["fecha"])
    df["bloque_base"] = _canon_str(df["bloque_base"])
    df["variedad_canon"] = _canon_str(df["variedad_canon"])
    df["grado"] = _canon_str(df["grado"])

    # Resolver columna de tallos final
    if "tallos_final_grado_dia" not in df.columns:
        raise KeyError(f"No encuentro tallos_final_grado_dia en {IN_TALLOS_ML2_BT}. Columnas: {list(df.columns)}")

    out = df[["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "tallos_final_grado_dia"]].copy()
    out = out.rename(columns={"tallos_final_grado_dia": "tallos_pred_ml2_grado_dia"})
    out["created_at"] = pd.Timestamp(datetime.now()).normalize()

    # Dia total
    keys_day = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    out_day = out.groupby(keys_day, as_index=False).agg(tallos_pred_ml2_dia=("tallos_pred_ml2_grado_dia", "sum"))
    out_day["created_at"] = out["created_at"].iloc[0]

    GOLD.mkdir(parents=True, exist_ok=True)
    write_parquet(out, OUT_TALLOS_GRADO)
    write_parquet(out_day, OUT_TALLOS_DIA)

    print(f"[OK] Wrote: {OUT_TALLOS_GRADO} rows={len(out)}")
    print(f"[OK] Wrote: {OUT_TALLOS_DIA} rows={len(out_day)}")

```

```

if __name__ == "__main__":
    main()

```

```

-----
[74/164] FILE: \src\gold\build_universe_harvest_grid_mll.py
-----
from __future__ import annotations

from pathlib import Path
import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

IN_HW = Path("data/gold/pred_harvest_window_mll.parquet")
IN_MAESTRO = Path("data/silver/fact_ciclo_maestro.parquet")

OUT_GRID = Path("data/gold/universe_harvest_grid_mll.parquet")

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _build_grid_from_start_n(
    df: pd.DataFrame,
    start_col: str,
    n_col: str,

```



```

) -> pd.DataFrame:
"""
Construye grid diario EXACTO de n días por fila, empezando en start.
Evita el mismatch count != n_pred por definición.

Output: explode columna 'fecha', y añade 'day_in_harvest_pred' (1..n).
"""
out = df.copy()
out[start_col] = _to_date(out[start_col])
out[n_col] = pd.to_numeric(out[n_col], errors="coerce").round().astype("Int64")

ok = out[start_col].notna() & out[n_col].notna() & (out[n_col].astype(float) > 0)
bad = ~ok
if bad.any():
    nbad = int(bad.sum())
    print(f"[WARN] Filas HW inválidas (sin start o n<=0): {nbad:,}. Se excluyen del grid.")
out = out[ok].copy()
if out.empty:
    return out.assign(fecha=pd.Series([], dtype="datetime64[ns]"))

# Construir offsets 0..n-1 por fila sin apply:
# Creamos un array de longitudes y repetimos filas.
lens = out[n_col].astype(int).to_numpy()
idx_rep = np.repeat(np.arange(len(out)), lens)

g = out.iloc[idx_rep].copy().reset_index(drop=True)
# offset dentro de fila repetida: 0..n-1
offset = np.concatenate([np.arange(n, dtype=int) for n in lens])
g["fecha"] = g[start_col] + pd.to_timedelta(offset, unit="D")
g["fecha"] = _to_date(g["fecha"])

g["day_in_harvest_pred"] = pd.Series(offset + 1, dtype="Int64")

return g

def main() -> None:
    created_at = pd.Timestamp.utcnow()

    hw = read_parquet(IN_HW).copy()
    maestro = read_parquet(IN_MAESTRO).copy()

    hw.columns = [str(c).strip() for c in hw.columns]
    maestro.columns = [str(c).strip() for c in maestro.columns]

    need_hw = {
        "ciclo_id",
        "bloque_base",
        "variedad_canon",
        "area",
        "tipo_sp",
        "fecha_sp",
        "harvest_start_pred",
        "n_harvest_days_pred_final",
    }
    miss = need_hw - set(hw.columns)
    if miss:
        raise ValueError(f"pred_harvest_window_m11: faltan columnas {sorted(miss)}")

    # Canon HW
    hw["ciclo_id"] = hw["ciclo_id"].astype(str)
    hw["bloque_base"] = _canon_int(hw["bloque_base"])
    hw["variedad_canon"] = _canon_str(hw["variedad_canon"])
    hw["area"] = _canon_str(hw["area"])
    hw["tipo_sp"] = _canon_str(hw["tipo_sp"])
    hw["fecha_sp"] = _to_date(hw["fecha_sp"])
    hw["harvest_start_pred"] = _to_date(hw["harvest_start_pred"])
    hw["n_harvest_days_pred_final"] = pd.to_numeric(hw["n_harvest_days_pred_final"], errors="coerce").round().
    astype("Int64")

    # Maestro: mínimo para enriquecer
    if "ciclo_id" not in maestro.columns:

```

```

        raise ValueError("fact_ciclo_maestro: falta ciclo_id")
maestro["ciclo_id"] = maestro["ciclo_id"].astype(str)

if "bloque_base" in maestro.columns:
    maestro["bloque_base"] = _canon_int(maestro["bloque_base"])

if "variedad_canon" in maestro.columns:
    maestro["variedad_canon"] = _canon_str(maestro["variedad_canon"])
elif "variedad" in maestro.columns:
    maestro["variedad_canon"] = _canon_str(maestro["variedad"])
else:
    maestro["variedad_canon"] = "UNKNOWN"

if "area" in maestro.columns:
    maestro["area"] = _canon_str(maestro["area"])
if "tipo_sp" in maestro.columns:
    maestro["tipo_sp"] = _canon_str(maestro["tipo_sp"])

m_take = [c for c in [
    "ciclo_id", "bloque", "bloque_padre", "bloque_base",
    "variedad", "variedad_canon", "tipo_sp", "area", "estado", "tallos_proy"
] if c in maestro.columns]
m2 = maestro[m_take].drop_duplicates(subset=["ciclo_id"])

base = hw.merge(m2, on="ciclo_id", how="left", suffixes=("", "_m"))

# Coalesce de algunos campos si vienen duplicados (por suffixes)
for c in ["bloque_base", "variedad_canon", "area", "tipo_sp"]:
    cx, cy = f"{c}_x", f"{c}_y"
    if cx in base.columns or cy in base.columns:
        x = base[cx] if cx in base.columns else pd.Series([pd.NA] * len(base))
        y = base[cy] if cy in base.columns else pd.Series([pd.NA] * len(base))
        base[c] = x.combine_first(y)
        base = base.drop(columns=[k for k in [cx, cy] if k in base.columns])

# Grid EXACTO usando start + n
grid = _build_grid_from_start_n(base, "harvest_start_pred", "n_harvest_days_pred_final")
if grid.empty:
    raise ValueError("Grid quedó vacío. Revisa harvest_start_pred / n_harvest_days_pred_final.")

# Recalcular end_pred consistente
grid["n_harvest_days_pred"] = pd.to_numeric(grid["n_harvest_days_pred_final"], errors="coerce").astype("Int64")
grid["harvest_end_pred"] = grid["harvest_start_pred"] + pd.to_timedelta(grid["n_harvest_days_pred"].astype(int) - 1, unit="D")
grid["harvest_end_pred"] = _to_date(grid["harvest_end_pred"])

grid["rel_pos_pred"] = np.where(
    grid["n_harvest_days_pred"].notna()
    & (grid["n_harvest_days_pred"].astype(float) > 0)
    & grid["day_in_harvest_pred"].notna(),
    grid["day_in_harvest_pred"].astype(float) / grid["n_harvest_days_pred"].astype(float),
    np.nan,
)

grid["stage"] = "HARVEST"

grid["dow"] = grid["fecha"].dt.dayofweek
grid["month"] = grid["fecha"].dt.month
grid["weekofyear"] = grid["fecha"].dt.isocalendar().week.astype(int)

keep = [
    "ciclo_id",
    "fecha",
    "bloque_base",
    "variedad_canon",
    "area",
    "tipo_sp",
    "estado" if "estado" in grid.columns else None,
    "tallos_proy" if "tallos_proy" in grid.columns else None,
    "fecha_sp",
    "harvest_start_pred",

```

```

        "harvest_end_pred",
        "n_harvest_days_pred",
        "day_in_harvest_pred",
        "rel_pos_pred",
        "stage",
        "dow",
        "month",
        "weekofyear",
        "mll_version" if "mll_version" in grid.columns else None,
    ]
    keep = [c for c in keep if c is not None and c in grid.columns]

    out = grid[keep].sort_values(["bloque_base", "variedad_canon", "fecha", "ciclo_id"]).reset_index(drop=True)
)
    out["created_at"] = created_at

    write_parquet(out, OUT_GRID)

    fmin = pd.to_datetime(out["fecha"].min()).date()
    fmax = pd.to_datetime(out["fecha"].max()).date()
    print(f"OK -> {OUT_GRID} | rows={len(out):,} | fecha_min={fmin} fecha_max={fmax}")

    # sanity: ahora debería ser ~0%
    chk = out.groupby("ciclo_id", dropna=False).agg(
        n=("fecha", "count"),
        n_pred=("n_harvest_days_pred", "first"),
    )
    diff = (chk["n"].astype(float) - chk["n_pred"].astype(float)).abs()
    print(f"[CHECK] % ciclos donde count!=n_harvest_days_pred: {float((diff > 0).mean()):.2%}")

if __name__ == "__main__":
    main()

```

```

-----
[75/164] FILE: \src\gold\build_universe_harvest_grid_ml2.py
-----

```

```

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    # ../src/gold/file.py -> repo_root = parents[2]
    return Path(__file__).resolve().parents[2]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
GOLD_DIR = DATA_DIR / "gold"

IN_GRID_ML1 = GOLD_DIR / "universe_harvest_grid_ml1.parquet"

# ML2 outputs
IN_SOH_PROD = GOLD_DIR / "pred_harvest_start_final_ml2.parquet"
IN_HH_PROD = GOLD_DIR / "pred_harvest_horizon_final_ml2.parquet"

OUT_GRID_ML2 = GOLD_DIR / "universe_harvest_grid_ml2.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:

```

```

return s.astype(str).str.upper().str.strip()

def main() -> None:
    grid = read_parquet(IN_GRID_ML1).copy()

    # Canon base
    grid["fecha"] = _to_date(grid["fecha"])
    grid["fecha_sp"] = _to_date(grid["fecha_sp"])
    grid["bloque_base"] = _canon_str(grid["bloque_base"])
    grid["variedad_canon"] = _canon_str(grid["variedad_canon"])
    if "estado" in grid.columns:
        grid["estado"] = _canon_str(grid["estado"])

    # Load ML2 start + horizon (PROD)
    if not IN_SOH_PROD.exists():
        raise FileNotFoundError(f"Missing ML2 SoH prod file: {IN_SOH_PROD}")
    if not IN_HH_PROD.exists():
        raise FileNotFoundError(f"Missing ML2 HH prod file: {IN_HH_PROD}")

    soh = read_parquet(IN_SOH_PROD).copy()
    hh = read_parquet(IN_HH_PROD).copy()

    # Normalize
    soh["harvest_start_final"] = _to_date(soh["harvest_start_final"])
    soh = soh[["ciclo_id", "harvest_start_final"]].drop_duplicates("ciclo_id")

    hh["harvest_end_final"] = _to_date(hh["harvest_end_final"])
    hh["n_harvest_days_final"] = pd.to_numeric(hh["n_harvest_days_final"], errors="coerce")
    hh = hh[["ciclo_id", "harvest_end_final", "n_harvest_days_final"]].drop_duplicates("ciclo_id")

    # Reduce ML1 grid to a header (cycle-level ML1 preds) to keep metadata stable
    g = grid.copy()
    g["harvest_start_pred"] = _to_date(g["harvest_start_pred"])
    g["harvest_end_pred"] = _to_date(g["harvest_end_pred"])

    head = (
        g.groupby("ciclo_id", as_index=False)
        .agg(
            bloque_base=("bloque_base", "first"),
            variedad_canon=("variedad_canon", "first"),
            area=("area", "first"),
            tipo_sp=("tipo_sp", "first"),
            estado=("estado", "first") if "estado" in g.columns else ("ciclo_id", "first"),
            tallos_proy=("tallos_proy", "first"),
            fecha_sp=("fecha_sp", "first"),
            harvest_start_pred=("harvest_start_pred", "min"),
            harvest_end_pred=("harvest_end_pred", "max"),
            n_harvest_days_pred=("n_harvest_days_pred", "max"),
            mll_version=("mll_version", "max"),
        )
    )

    # Attach ML2 start + horizon
    head = head.merge(soh, on="ciclo_id", how="left").merge(hh, on="ciclo_id", how="left")

    # Fallbacks (should be rare)
    head["harvest_start_final"] = head["harvest_start_final"].fillna(head["harvest_start_pred"])
    head["harvest_end_final"] = head["harvest_end_final"].fillna(head["harvest_end_pred"])

    # If n_harvest_days_final missing, derive from start/end
    miss_n = head["n_harvest_days_final"].isna()
    if miss_n.any():
        head.loc[miss_n, "n_harvest_days_final"] = (
            (head.loc[miss_n, "harvest_end_final"] - head.loc[miss_n, "harvest_start_final"]).dt.days + 1
        )

    head["n_harvest_days_final"] = pd.to_numeric(head["n_harvest_days_final"], errors="coerce").fillna(0.0)
    head["n_harvest_days_final"] = head["n_harvest_days_final"].clip(lower=7)

    # Build ML2 grid per cycle: fecha range [start_final, end_final]
    parts = []

```

```

for r in head.itertuples(index=False):
    if pd.isna(r.harvest_start_final) or pd.isna(r.harvest_end_final):
        continue
    if r.harvest_end_final < r.harvest_start_final:
        continue
    dates = pd.date_range(start=r.harvest_start_final, end=r.harvest_end_final, freq="D")
    tmp = pd.DataFrame({
        "ciclo_id": r.ciclo_id,
        "fecha": dates,
    })
    tmp["bloque_base"] = r.bloque_base
    tmp["variedad_canon"] = r.variedad_canon
    tmp["area"] = r.area
    tmp["tipo_sp"] = r.tipo_sp
    tmp["estado"] = r.estado if "estado" in head.columns else "UNKNOWN"
    tmp["tallos_proy"] = r.tallos_proy
    tmp["fecha_sp"] = r.fecha_sp

    tmp["harvest_start_pred"] = r.harvest_start_pred
    tmp["harvest_end_pred"] = r.harvest_end_pred
    tmp["n_harvest_days_pred"] = r.n_harvest_days_pred

    tmp["harvest_start_final"] = r.harvest_start_final
    tmp["harvest_end_final"] = r.harvest_end_final
    tmp["n_harvest_days_final"] = r.n_harvest_days_final

    parts.append(tmp)

out = pd.concat(parts, ignore_index=True) if parts else pd.DataFrame()

if out.empty:
    raise RuntimeError("ML2 universe grid is empty. Check ML2 inputs and cycle headers.")

# Re-index / features
out["day_in_harvest_final"] = (out["fecha"] - out["harvest_start_final"]).dt.days.astype(int) + 1
out["rel_pos_final"] = (out["day_in_harvest_final"] - 1) / (out["n_harvest_days_final"] - 1).replace(0, np
.nan)
out["rel_pos_final"] = out["rel_pos_final"].fillna(0.0).clip(0.0, 1.0)

# Keep ML1 rel_pos_pred if you want, but this grid is ML2-driven.
out["dow"] = out["fecha"].dt.dayofweek
out["month"] = out["fecha"].dt.month
out["weekofyear"] = out["fecha"].dt.isocalendar().week.astype(int)

# Versioning
out["ml2_version"] = "ml2_harvest_start+hh"
out["created_at"] = pd.Timestamp(datetime.now()).normalize()

OUT_GRID_ML2.parent.mkdir(parents=True, exist_ok=True)
write_parquet(out, OUT_GRID_ML2)

print(f"[OK] Wrote: {OUT_GRID_ML2}")
print(f"      rows={len(out):,} cycles={out['ciclo_id'].nunique():,} "
      f"fecha_range=[{out['fecha'].min().date():,}..{out['fecha'].max().date():,}]")

if __name__ == "__main__":
    main()

```

```

-----
[76/164] FILE: \src\gold\build_view_planificacion_campo_tallos.py
-----

```

```

from __future__ import annotations

from pathlib import Path
import pandas as pd
import numpy as np

from common.io import read_parquet, write_parquet

```

```

IN_PRED = Path("data/gold/pred_tallos_grado_dia_mll_full.parquet")
IN_MAESTRO = Path("data/silver/fact_ciclo_maestro.parquet")

OUT_DIA = Path("data/gold/view_planificacion_campo_tallos_dia.parquet")
OUT_SEM = Path("data/gold/view_planificacion_campo_tallos_semana.parquet")
OUT_SEM_BLOQUE = Path("data/gold/view_planificacion_campo_tallos_semana_bloque.parquet")
OUT_SEM_AREA = Path("data/gold/view_planificacion_campo_tallos_semana_area.parquet")

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.strip().str.upper()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _coalesce_area(df: pd.DataFrame) -> pd.DataFrame:
    if "area" in df.columns:
        df["area"] = _canon_str(df["area"]).replace({"NaN": np.nan, "NONE": np.nan})
        return df
    if "area_x" in df.columns or "area_y" in df.columns:
        ax = df["area_x"] if "area_x" in df.columns else pd.Series([pd.NA] * len(df))
        ay = df["area_y"] if "area_y" in df.columns else pd.Series([pd.NA] * len(df))
        df["area"] = ay.combine_first(ax)
        df = df.drop(columns=[c for c in ["area_x", "area_y"] if c in df.columns])
        df["area"] = _canon_str(df["area"]).replace({"NaN": np.nan, "NONE": np.nan})
        return df
    df["area"] = pd.Series([pd.NA] * len(df), dtype="object")
    return df

def _pick_cols(pred: pd.DataFrame) -> dict[str, str]:
    """
    Decide qué columnas usar (ML1 vs baseline).
    Devuelve dict con keys:
    - tallos_grado: columna tallos por grado/día
    - tallos_dia: columna tallos total/día (puede ser None)
    - share: columna share por grado (para reconstrucción si hace falta)
    """
    cols = set(pred.columns)

    # Preferimos ML1
    tallos_grado_mll = None
    for c in ["tallos_pred_grado_dia_mll", "tallos_pred_grado_dia_mll_full", "tallos_pred_grado_dia"]:
        if c in cols:
            # OJO: tallos_pred_grado_dia es ambiguo; lo validamos luego
            tallos_grado_mll = c
            break

    tallos_dia_mll = None
    for c in ["tallos_pred_dia_mll", "tallos_pred_mll_dia", "tallos_pred_dia"]:
        if c in cols:
            tallos_dia_mll = c
            break

    share_mll = None
    for c in ["share_grado_mll", "share_mll", "share_grado_pred"]:
        if c in cols:
            share_mll = c
            break

```

```

# Baseline (opcional)
share_base = None
for c in ["share_grado_baseline", "share_baseline"]:
    if c in cols:
        share_base = c
        break

return {
    "tallos_grado": tallos_grado_m11,
    "tallos_dia": tallos_dia_m11,
    "share_m11": share_m11,
    "share_base": share_base,
}

def main() -> None:
    if not IN_PRED.exists():
        raise FileNotFoundError(f"No existe: {IN_PRED}")
    if not IN_MAESTRO.exists():
        raise FileNotFoundError(f"No existe: {IN_MAESTRO}")

    pred = read_parquet(IN_PRED).copy()
    maestro = read_parquet(IN_MAESTRO).copy()

    # Requeridos mínimos
    _require(pred, ["fecha", "bloque_base", "variedad_canon", "grado"], "pred_tallos_grado_dia_m11_full")
    _require(maestro, ["bloque_base", "area"], "fact_ciclo_maestro")

    # Normalización tipos
    pred["fecha"] = _to_date(pred["fecha"])
    pred["bloque_base"] = _canon_int(pred["bloque_base"])
    pred["grado"] = _canon_int(pred["grado"])
    pred["variedad_canon"] = _canon_str(pred["variedad_canon"])

    maestro["bloque_base"] = _canon_int(maestro["bloque_base"])
    maestro["area"] = _canon_str(maestro["area"])

    # Merge área
    map_area = (
        maestro[["bloque_base", "area"]]
        .dropna(subset=["bloque_base"])
        .sort_values(["bloque_base", "area"])
        .drop_duplicates(subset=["bloque_base"], keep="first")
    )
    pred = pred.merge(map_area, on="bloque_base", how="left")
    pred = _coalesce_area(pred)
    pred["area"] = pred["area"].fillna("UNKNOWN")

    # Selección de columnas ML1/baseline
    sel = _pick_cols(pred)
    tg = sel["tallos_grado"]
    td = sel["tallos_dia"]
    sh = sel["share_m11"]

    if tg is None and (td is None or sh is None):
        raise ValueError(
            "No tengo suficientes columnas para construir tallos por grado.\n"
            "Necesito (tallos_pred_grado_dia_m11) o (tallos_pred_dia_m11 + share_grado_m11).\n"
            f"Columnas disponibles: {list(pred.columns)}"
        )

    # Si NO existe tallos por grado, lo reconstruimos
    if tg is None:
        pred[td] = pd.to_numeric(pred[td], errors="coerce")
        pred[sh] = pd.to_numeric(pred[sh], errors="coerce")
        pred["tallos_pred_grado_dia_m11"] = pred[td] * pred[sh]
        tg = "tallos_pred_grado_dia_m11"

    # Forzar numérico
    pred[tg] = pd.to_numeric(pred[tg], errors="coerce")

    # ---- DIAGNÓSTICO CLAVE: detectar ?duplicado por grado? (lo que te pasó) ----

```

```

# Si para un mismo grupo (fecha,bloque,variedad) hay muchos grados y tg es casi constante => tg NO es por
grado.
gchk = pred.groupby(["fecha", "bloque_base", "variedad_canon"], dropna=False)[tg].agg(["nunique", "count"]
)
# Consideramos sospechoso si hay >1 fila (varios grados) pero solo 1 valor único
suspect = ((gchk["count"] > 3) & (gchk["nunique"] <= 1)).mean()
if suspect > 0.01:
    print(
        f"[WARN] La columna '{tg}' parece CONSTANTE entre grados en ~{suspect:.2%} de grupos "
        f"(eso indica que NO es tallos por grado). "
        f"Revisa el gold upstream: {IN_PRED.name}"
    )

# Vista diaria (por grado)
out_dia = (
    pred.groupby(["fecha", "area", "bloque_base", "variedad_canon", "grado"], dropna=False, as_index=False
)
        .agg(tallos_pred_grado_dia=(tg, "sum"))
    )

# Total diario: SI existe columna total diaria confiable, la usamos; sino sumamos por grado
if td is not None:
    # usamos la primera por grupo (debería ser la misma para todos los grados)
    pred[td] = pd.to_numeric(pred[td], errors="coerce")
    base_total = (
        pred.groupby(["fecha", "area", "bloque_base", "variedad_canon"], dropna=False, as_index=False)
            .agg(tallos_pred_dia=(td, "first"))
    )
    out_dia = out_dia.merge(base_total, on=["fecha", "area", "bloque_base", "variedad_canon"], how="left")
else:
    tot = (
        out_dia.groupby(["fecha", "area", "bloque_base", "variedad_canon"], dropna=False, as_index=False)
            .agg(tallos_pred_dia=("tallos_pred_grado_dia", "sum"))
    )
    out_dia = out_dia.merge(tot, on=["fecha", "area", "bloque_base", "variedad_canon"], how="left")

out_dia["created_at"] = pd.Timestamp.utcnow()
out_dia = out_dia.sort_values(["fecha", "area", "bloque_base", "variedad_canon", "grado"]).reset_index(drop=True)

write_parquet(out_dia, OUT_DIA)
print(f"OK -> {OUT_DIA} | rows={len(out_dia):,}")

# Vista semanal ISO
iso = out_dia["fecha"].dt.isocalendar()
out_dia["anio"] = iso.year.astype(int)
out_dia["semana_iso"] = iso.week.astype(int)

sem_grado = (
    out_dia.groupby(["anio", "semana_iso", "area", "bloque_base", "variedad_canon", "grado"], dropna=False
, as_index=False)
        .agg(tallos_pred_grado_semana=("tallos_pred_grado_dia", "sum"))
    )
sem_total = (
    sem_grado.groupby(["anio", "semana_iso", "area", "bloque_base", "variedad_canon"], dropna=False, as_in
dex=False)
        .agg(tallos_pred_semana=("tallos_pred_grado_semana", "sum"))
    )
out_sem = sem_grado.merge(
    sem_total,
    on=["anio", "semana_iso", "area", "bloque_base", "variedad_canon"],
    how="left",
)
out_sem["created_at"] = pd.Timestamp.utcnow()
out_sem = out_sem.sort_values(["anio", "semana_iso", "area", "bloque_base", "variedad_canon", "grado"]).re
set_index(drop=True)

write_parquet(out_sem, OUT_SEM)
print(f"OK -> {OUT_SEM} | rows={len(out_sem):,}")
# -----
# EXTRA 1) Semana por bloque (SIN grado)
# -----

```



```

out_sem_bloque = (
    out_sem.groupby(["anio", "semana_iso", "area", "bloque_base", "variedad_canon"], dropna=False, as_index=False)
        .agg(tallos_pred_semana=("tallos_pred_grado_semana", "sum"))
)
out_sem_bloque["created_at"] = pd.Timestamp.utcnow()
out_sem_bloque = out_sem_bloque.sort_values(
    ["anio", "semana_iso", "area", "bloque_base", "variedad_canon"]
).reset_index(drop=True)

write_parquet(out_sem_bloque, OUT_SEM_BLOQUE)
print(f"OK -> {OUT_SEM_BLOQUE} | rows={len(out_sem_bloque):,}")

# -----
# EXTRA 2) Semana por área (TOTAL, sin bloque/variedad/grado)
# -----
out_sem_area = (
    out_sem_bloque.groupby(["anio", "semana_iso", "area"], dropna=False, as_index=False)
        .agg(tallos_pred_semana_area=("tallos_pred_semana", "sum"))
)
out_sem_area["created_at"] = pd.Timestamp.utcnow()
out_sem_area = out_sem_area.sort_values(["anio", "semana_iso", "area"]).reset_index(drop=True)

write_parquet(out_sem_area, OUT_SEM_AREA)
print(f"OK -> {OUT_SEM_AREA} | rows={len(out_sem_area):,}")

# Check: sum(grados)=total
chk = out_dia.groupby(["fecha", "area", "bloque_base", "variedad_canon"], dropna=False).agg(
    s_grados=("tallos_pred_grado_dia", "sum"),
    s_total=("tallos_pred_dia", "first"),
)
mismatch = float((np.abs(chk["s_grados"] - chk["s_total"]) > 1e-6).mean())
print(f"[CHECK] % mismatch (día): {mismatch:.4f}")

# Check semanal: suma por bloque = suma por área
chk_sem = out_sem_bloque.groupby(["anio", "semana_iso", "area"], dropna=False)["tallos_pred_semana"].sum()
chk_area = out_sem_area.set_index(["anio", "semana_iso", "area"])[["tallos_pred_semana_area"]]
chk = (chk_sem - chk_area).abs()
mismatch_sem = float((chk > 1e-6).mean()) if len(chk) else 0.0
print(f"[CHECK] % mismatch (semana bloque vs area): {mismatch_sem:.4f}")

if __name__ == "__main__":
    main()

-----
[77/164] FILE: \src\gold\postprocess_curva_share_smooth_mll.py
-----
from __future__ import annotations

from pathlib import Path

import numpy as np
import pandas as pd

from common.io import read_parquet, write_parquet

# =====
# Paths
# =====
PRED_FACTOR_PATH = Path("data/gold/pred_factor_curva_mll.parquet")
FEATURES_CURVA_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_mll.parquet")
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")
DIM_VAR_PATH = Path("data/silver/dim_variedad_canon.parquet")

OUT_PATH = Path("data/gold/pred_factor_curva_mll.parquet") # overwrite downstream-safe

# =====
# Hyperparams (estadísticos + smoothing)
# =====

```

```

# cap por segmento = quantil alto del share_real (por defecto P99.5)
CAP_Q = 0.995

# buckets de duración (días) para estabilizar caps
NDAYS_BINS = [0, 25, 35, 45, 55, 65, 80, 120, 10_000]
NDAYS_LABELS = ["<=25", "26-35", "36-45", "46-55", "56-65", "66-80", "81-120", "120+"]

# smoothing (share-space)
SMOOTH_WIN = 5          # 3 o 5 recomendado
SMOOTH_CENTER = True    # centered rolling
SMOOTH_MINP = 1

# factor safety (mantener compat)
FACTOR_MIN, FACTOR_MAX = 0.2, 5.0
EPS = 1e-9

# =====
# Helpers
# =====
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int64(s: pd.Series) -> pd.Series:
    # usamos Int64 nullable para evitar merges raros con NaNs
    x = pd.to_numeric(s, errors="coerce")
    return x.astype("Int64")

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Cols={list(df.columns)}")

def _load_var_map(dim_var: pd.DataFrame) -> dict[str, str]:
    _require(dim_var, ["variedad_raw", "variedad_canon"], "dim_variedad_canon")
    dv = dim_var.copy()
    dv["variedad_raw"] = _canon_str(dv["variedad_raw"])
    dv["variedad_canon"] = _canon_str(dv["variedad_canon"])
    dv = dv.dropna(subset=["variedad_raw", "variedad_canon"]).drop_duplicates(subset=["variedad_raw"])
    return dict(zip(dv["variedad_raw"], dv["variedad_canon"]))

def _ndays_bucket(n: pd.Series) -> pd.Series:
    n2 = pd.to_numeric(n, errors="coerce")
    return pd.cut(n2, bins=NDAYS_BINS, labels=NDAYS_LABELS, right=True, include_lowest=True).astype(str)

def _rolling_smooth_share(df: pd.DataFrame, share_col: str, out_col: str) -> pd.DataFrame:
    """
    Suaviza share por ciclo en el orden temporal de fecha.
    """
    out = df.copy()
    out = out.sort_values(["ciclo_id", "fecha"], kind="mergesort")
    out[out_col] = (
        out.groupby("ciclo_id", dropna=False)[share_col]
        .transform(lambda s: s.rolling(SMOOTH_WIN, center=SMOOTH_CENTER, min_periods=SMOOTH_MINP).mean())
    )
    return out

# =====
# Main
# =====
def main() -> None:
    created_at = pd.Timestamp.now("UTC")

```

```

# -----
# Read inputs
# -----
for p in [PRED_FACTOR_PATH, FEATURES_CURVA_PATH, UNIVERSE_PATH, PROG_PATH, DIM_VAR_PATH]:
    if not p.exists():
        raise FileNotFoundError(f"No existe: {p}")

pred = read_parquet(PRED_FACTOR_PATH).copy()
feat = read_parquet(FEATURES_CURVA_PATH).copy()
uni = read_parquet(UNIVERSE_PATH).copy()
prog = read_parquet(PROG_PATH).copy()
dim_var = read_parquet(DIM_VAR_PATH).copy()

var_map = _load_var_map(dim_var)

# -----
# Canon llaves
# -----
_require(pred, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "pred_factor_curva_m11")
pred["ciclo_id"] = pred["ciclo_id"].astype(str)
pred["fecha"] = _to_date(pred["fecha"])
pred["bloque_base"] = _canon_int64(pred["bloque_base"])
pred["variedad_canon"] = _canon_str(pred["variedad_canon"])

_require(
    feat,
    ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia", "tallos_proy"],
    "features_curva",
)
feat["ciclo_id"] = feat["ciclo_id"].astype(str)
feat["fecha"] = _to_date(feat["fecha"])
feat["bloque_base"] = _canon_int64(feat["bloque_base"])
feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
for c in ["area", "tipo_sp"]:
    if c in feat.columns:
        feat[c] = _canon_str(feat[c])

_require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_m11")
uni["ciclo_id"] = uni["ciclo_id"].astype(str)
uni["fecha"] = _to_date(uni["fecha"])
uni["bloque_base"] = _canon_int64(uni["bloque_base"])
uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

_require(prog, ["ciclo_id", "fecha", "bloque_base", "variedad", "tallos_real_dia"], "dim_cosecha_progress_
bloque_fecha")
prog["ciclo_id"] = prog["ciclo_id"].astype(str)
prog["fecha"] = _to_date(prog["fecha"])
prog["bloque_base"] = _canon_int64(prog["bloque_base"])
prog["variedad_raw"] = _canon_str(prog["variedad"])
prog["variedad_canon"] = prog["variedad_raw"].map(var_map).fillna(prog["variedad_raw"])
prog["variedad_canon"] = _canon_str(prog["variedad_canon"])
prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce").fillna(0.0)

key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]

# -----
# Universe day-count (n_harvest_days_pred) por ciclo (para buckets)
# -----
if "n_harvest_days_pred" in uni.columns:
    n_days = (
        uni.groupby("ciclo_id", dropna=False)["n_harvest_days_pred"]
        .max()
        .rename("n_days")
        .reset_index()
    )
else:
    n_days = uni.groupby("ciclo_id", dropna=False)["fecha"].count().rename("n_days").reset_index()

# -----
# Build share_real panelizado en universe (missing days = 0)
# -----

```

```

uni_k = uni[key].drop_duplicates()
prog_k = prog[key + ["tallos_real_dia"]].drop_duplicates(subset=key)

real_panel = uni_k.merge(prog_k, on=key, how="left")
real_panel["tallos_real_dia"] = pd.to_numeric(real_panel["tallos_real_dia"], errors="coerce").fillna(0.0)

cyc_sum = real_panel.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform("sum").astype(float)
real_panel["has_real"] = cyc_sum > 0
real_panel["share_real"] = np.where(
    cyc_sum > 0,
    real_panel["tallos_real_dia"].astype(float) / cyc_sum,
    np.nan,
)

# -----
# Segment info (variedad/area/tipo_sp + bucket)
# -----
seg_cols = ["ciclo_id", "bloque_base", "variedad_canon"]
for c in ["area", "tipo_sp"]:
    if c in feat.columns:
        seg_cols.append(c)

seg = feat[seg_cols].drop_duplicates(subset=["ciclo_id", "bloque_base", "variedad_canon"])
seg = seg.merge(n_days, on="ciclo_id", how="left")
seg["ndays_bucket"] = _ndays_bucket(seg["n_days"])
for c in ["area", "tipo_sp"]:
    if c in seg.columns:
        seg[c] = _canon_str(seg[c].fillna("UNKNOWN"))

# -----
# Caps estadísticos (por segmento) usando share_real histórico
# -----
real_for_caps = real_panel.merge(
    seg,
    on=["ciclo_id", "bloque_base", "variedad_canon"],
    how="left",
    suffixes=("", "_seg"),
)

seg_key = ["variedad_canon", "ndays_bucket"]
if "area" in real_for_caps.columns:
    seg_key.append("area")
if "tipo_sp" in real_for_caps.columns:
    seg_key.append("tipo_sp")

caps_base = real_for_caps[real_for_caps["has_real"] & real_for_caps["share_real"].notna()].copy()

# fallback global cap
global_cap = float(caps_base["share_real"].quantile(CAP_Q)) if len(caps_base) else 1.0

caps = (
    caps_base.groupby(seg_key, dropna=False)["share_real"]
    .quantile(CAP_Q)
    .rename("cap_share")
    .reset_index()
)
caps["cap_share"] = pd.to_numeric(caps["cap_share"], errors="coerce").fillna(global_cap)
caps["cap_share"] = caps["cap_share"].clip(lower=0.0, upper=0.30)

# -----
# Construir share_pred desde pred_factor (preferir share_curva_mll si existe)
# -----
take_feat = key + ["tallos_pred_baseline_dia", "tallos_proy"]
for c in ["area", "tipo_sp"]:
    if c in feat.columns:
        take_feat.append(c)
take_feat = list(dict.fromkeys(take_feat))

pred2 = pred.merge(feat[take_feat].drop_duplicates(subset=key), on=key, how="left", suffixes=("", "_f"))
pred2 = pred2.merge(n_days, on="ciclo_id", how="left")
pred2["ndays_bucket"] = _ndays_bucket(pred2["n_days"])

```

```

for c in ["area", "tipo_sp"]:
    if c in pred2.columns:
        pred2[c] = _canon_str(pred2[c].fillna("UNKNOWN"))

# robust: si faltan columnas numéricas, crear con 0
if "tallos_pred_baseline_dia" in pred2.columns:
    pred2["tallos_pred_baseline_dia"] = pd.to_numeric(pred2["tallos_pred_baseline_dia"], errors="coerce").
fillna(0.0)
else:
    pred2["tallos_pred_baseline_dia"] = 0.0

if "tallos_proy" in pred2.columns:
    pred2["tallos_proy"] = pd.to_numeric(pred2["tallos_proy"], errors="coerce").fillna(0.0)
else:
    pred2["tallos_proy"] = 0.0

if "share_curva_m11" in pred2.columns:
    share_pred = pd.to_numeric(pred2["share_curva_m11"], errors="coerce").fillna(0.0)
    share_pred = share_pred.clip(lower=0.0)
    pred2["share_pred_in"] = share_pred
    pred2["_share_source"] = "share_curva_m11"
else:
    if "factor_curva_m11" not in pred2.columns:
        raise ValueError("pred_factor_curva_m11 no tiene share_curva_m11 ni factor_curva_m11. No puedo rec
onstruir share.")
    factor = pd.to_numeric(pred2["factor_curva_m11"], errors="coerce").fillna(1.0)
    tallos_m11_dia = pred2["tallos_pred_baseline_dia"].astype(float) * factor.astype(float)
    denom = tallos_m11_dia.groupby(pred2["ciclo_id"]).transform("sum").astype(float)
    pred2["share_pred_in"] = np.where(denom > 0, tallos_m11_dia / denom, 0.0)
    pred2["_share_source"] = "factor_curva_m11"

# -----
# Aplicar cap estadístico + smoothing + renorm por ciclo
# -----
# IMPORTANTÍSIMO: pred ya puede traer cap_share; si no lo botamos, el merge crea cap_share_x/cap_share_y y
luego falla.
if "cap_share" in pred2.columns:
    pred2 = pred2.drop(columns=["cap_share"], errors="ignore")

pred2 = pred2.merge(caps, on="seg_key", how="left")

# si por algo no vino cap_share tras el merge, fallback a global_cap
if "cap_share" not in pred2.columns:
    pred2["cap_share"] = float(global_cap)
else:
    pred2["cap_share"] = pd.to_numeric(pred2["cap_share"], errors="coerce").fillna(global_cap)

pred2["cap_share"] = pred2["cap_share"].clip(lower=0.0, upper=0.30)

# cap pre-smooth
pred2["share_cap_pre"] = pd.to_numeric(pred2["share_pred_in"], errors="coerce").fillna(0.0).clip(lower=0.0
)

pred2["was_capped_pre"] = pred2["share_cap_pre"] > pred2["cap_share"]
pred2["share_cap_pre"] = np.minimum(pred2["share_cap_pre"], pred2["cap_share"])

# smooth
pred2 = _rolling_smooth_share(pred2, "share_cap_pre", "share_smooth")
pred2["share_smooth"] = pd.to_numeric(pred2["share_smooth"], errors="coerce").fillna(0.0).clip(lower=0.0)

# cap post-smooth
pred2["was_capped_post"] = pred2["share_smooth"] > pred2["cap_share"]
pred2["share_cap_post"] = np.minimum(pred2["share_smooth"], pred2["cap_share"])

# renormalize per cycle
s = pred2.groupby("ciclo_id", dropna=False)["share_cap_post"].transform("sum").astype(float)
pred2["share_final"] = np.where(s > 0, pred2["share_cap_post"] / s, 0.0)

# -----
# Reconstrucción tallos_m11_dia y factor compatible downstream
# -----
pred2["tallos_pred_m11_dia_smooth"] = pred2["tallos_proy"].astype(float) * pred2["share_final"].astyp(flo
at)

```

```

base = pred2["tallos_pred_baseline_dia"].astype(float)
pred2["factor_curva_mll_raw_smooth"] = np.where(base > 0, pred2["tallos_pred_mll_dia_smooth"] / (base + EPS), 1.0)
pred2["factor_curva_mll_smooth"] = pred2["factor_curva_mll_raw_smooth"].clip(lower=FACTOR_MIN, upper=FACTOR_MAX)
pred2["factor_curva_mll_smooth"] = np.where(np.isfinite(pred2["factor_curva_mll_smooth"]), pred2["factor_curva_mll_smooth"], 1.0)

# -----
# Overwrite principal factor columns (downstream)
# -----
if "mll_version" not in pred2.columns:
    pred2["mll_version"] = "UNKNOWN"

pred2["factor_curva_mll_raw"] = pred2["factor_curva_mll_raw_smooth"]
pred2["factor_curva_mll"] = pred2["factor_curva_mll_smooth"]

# mantener share para auditoría
pred2["share_curva_mll_raw"] = pred2.get("share_curva_mll_raw", pred2["share_pred_in"])
pred2["share_curva_mll"] = pred2["share_final"]

pred2["created_at"] = created_at

# -----
# Checks
# -----
cyc = pred2.groupby("ciclo_id", dropna=False).agg(
    proy=("tallos_proy", "max"),
    sum_mll=("tallos_pred_mll_dia_smooth", "sum"),
).reset_index()
cyc["abs_diff"] = (cyc["proy"] - cyc["sum_mll"]).abs()
max_abs = float(cyc["abs_diff"].max()) if len(cyc) else float("nan")

ss = pred2.groupby("ciclo_id", dropna=False)["share_final"].sum()
smin, smax = (float(ss.min()), float(ss.max())) if len(ss) else (float("nan"), float("nan"))

cap_pre = float(pred2["was_capped_pre"].mean()) if len(pred2) else float("nan")
cap_post = float(pred2["was_capped_post"].mean()) if len(pred2) else float("nan")

# -----
# Write output (compatible)
# -----
keep = [
    "ciclo_id", "fecha", "bloque_base", "variedad_canon",
    "factor_curva_mll", "factor_curva_mll_raw", "mll_version", "created_at",
    # auditoría (no rompe downstream)
    "_share_source", "cap_share",
    "share_pred_in", "share_smooth", "share_curva_mll",
    "tallos_pred_mll_dia_smooth",
    "factor_curva_mll_raw_smooth",
    "was_capped_pre", "was_capped_post",
]
keep = [c for c in keep if c in pred2.columns]

out = pred2[keep].sort_values(["bloque_base", "variedad_canon", "fecha", "ciclo_id"]).reset_index(drop=True)

OUT_PATH.parent.mkdir(parents=True, exist_ok=True)
write_parquet(out, OUT_PATH)

print(f"OK -> {OUT_PATH} | rows={len(out):,}")
print(f"[CHECK] mass balance vs tallos_proy | max abs diff: {max_abs:.12f}")
print(f"[CHECK] share sum per ciclo | min={smin:.8f} max={smax:.8f}")
print(f"[CHECK] capped rate pre={cap_pre:.4f} post={cap_post:.4f}")
print(f"[CAP] global_cap(Q={CAP_Q}) = {global_cap:.6f}")
print(f"[SMOOTH] win={SMOOTH_WIN} centered={SMOOTH_CENTER}")

if __name__ == "__main__":
    main()

```

```
-----  
[78/164] FILE: \src\models\m11\apply_curva_beta_params.py  
-----
```

```
# src/models/m11/apply_curva_beta_params.py  
from __future__ import annotations
```

```
from pathlib import Path  
import json  
import math  
import numpy as np  
import pandas as pd  
from joblib import load
```

```
from common.io import read_parquet, write_parquet
```

```
# =====  
# Paths  
# =====  
FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")  
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_m11.parquet")
```

```
REG_BETA_PARAMS = Path("models_registry/m11/curva_beta_params")  
REG_MULT_DIA = Path("models_registry/m11/curva_beta_multiplier_dia")
```

```
CAP_PATH = Path("data/gold/dim_cap_tallos_real_dia.parquet")
```

```
OUT_PATH = Path("data/gold/pred_factor_curva_m11.parquet")
```

```
# =====  
# Daily model columns (multiplier)  
# =====
```

```
NUM_COLS_DAILY = [  
    "day_in_harvest",  
    "rel_pos",  
    "n_harvest_days",  
    "pct_avance_real",  
    "dia_rel_cosecha_real",  
    "gdc_acum_real",  
    "rainfall_mm_dia",  
    "horas_lluvia",  
    "temp_avg_dia",  
    "solar_energy_j_m2_dia",  
    "wind_speed_avg_dia",  
    "wind_run_dia",  
    "gdc_dia",  
    "dias_desde_sp",  
    "gdc_acum_desde_sp",  
    "dow",  
    "month",  
    "weekofyear",  
]
```

```
CAT_COLS = ["variedad_canon", "area", "tipo_sp"]
```

```
# =====  
# Config  
# =====
```

```
EPS = 1e-12  
REL_CLIP = 1e-4
```

```
# Small mixing with baseline share (robustness)  
LAMBDA_BASE = 0.05
```

```
# Multiplier stability clip: exp(log_mult) in [0.37, 2.72]  
LOG_MULT_CLIP = (-1.0, 1.0)
```

```
# Mild smoothing after combining (keeps unimodal-ish)  
SMOOTH_WIN = 3
```

```
# Factor caps (statistical, not capacity)  
FACTOR_MIN, FACTOR_MAX = 0.01, 5.0
```

```

# Cap table fallback
CAP_FALLBACK = 4000.0

# Cap redistribution
MAX_REDIST_ITERS = 20

# =====
# Helpers
# =====
def _latest_version_dir(root: Path) -> Path:
    if not root.exists():
        raise FileNotFoundError(f"No existe {root}")
    dirs = [p for p in root.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {root}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _dedupe_columns(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.Index(df.columns.astype(str))
    if cols.is_unique:
        return df
    out = df.copy()
    seen: dict[str, list[int]] = {}
    for i, c in enumerate(out.columns.astype(str)):
        seen.setdefault(c, []).append(i)
    keep: dict[str, pd.Series] = {}
    for c, idxs in seen.items():
        s = out.iloc[:, idxs[0]]
        for j in idxs[1:]:
            s2 = out.iloc[:, j]
            s = s.where(s.notna(), s2)
        keep[c] = s
    ordered: list[str] = []
    for c in out.columns.astype(str):
        if c not in ordered:
            ordered.append(c)
    return pd.DataFrame({c: keep[c] for c in ordered})

def _coalesce_cols(df: pd.DataFrame, out_col: str, candidates: list[str]) -> None:
    if out_col in df.columns:
        base = df[out_col]
    else:
        base = pd.Series([pd.NA] * len(df), index=df.index)
    for c in candidates:
        if c in df.columns:
            base = base.where(base.notna(), df[c])
    df[out_col] = base

def _log_beta_pdf(x: np.ndarray, a: float, b: float) -> np.ndarray:
    logB = math.lgamma(a) + math.lgamma(b) - math.lgamma(a + b)
    return (a - 1.0) * np.log(x) + (b - 1.0) * np.log(1.0 - x) - logB

def _beta_share(rel: np.ndarray, a: float, b: float) -> np.ndarray:
    rel = np.clip(rel, REL_CLIP, 1.0 - REL_CLIP)
    lp = _log_beta_pdf(rel, float(a), float(b))
    lp = lp - np.max(lp)

```



```

p = np.exp(lp)
s = float(np.sum(p))
return p / s if s > 0 else np.zeros_like(p)

def _smooth_share_centered(df: pd.DataFrame, share_col: str, is_h_col: str) -> np.ndarray:
    if SMOOTH_WIN <= 1:
        return df[share_col].to_numpy(dtype=float)

    tmp = df[["ciclo_id", share_col, is_h_col]].copy()
    tmp[share_col] = pd.to_numeric(tmp[share_col], errors="coerce").fillna(0.0).clip(lower=0.0)
    tmp.loc[~tmp[is_h_col], share_col] = 0.0

    sm = (
        tmp.groupby("ciclo_id", dropna=False)[share_col]
        .rolling(window=SMOOTH_WIN, center=True, min_periods=1)
        .mean()
        .reset_index(level=0, drop=True)
    ).to_numpy(dtype=float)

    is_h = tmp[is_h_col].to_numpy(dtype=bool)
    sm = np.where(is_h, sm, 0.0)

    s = pd.Series(sm).groupby(tmp["ciclo_id"], dropna=False).transform("sum").to_numpy(dtype=float)
    sm = np.where(s > 0, sm / s, sm)
    return sm

def _first_nonnull(s: pd.Series) -> float:
    s2 = s.dropna()
    return float(s2.iloc[0]) if len(s2) else float("nan")

def _apply_cap_and_redistribute(t: np.ndarray, cap: np.ndarray, mask: np.ndarray) -> tuple[np.ndarray, float]:
    """
    Apply per-day cap within mask and redistribute excess to days with slack.
    Returns (t_capped, uncovered_excess) where uncovered_excess>0 implies not enough slack.
    """
    out = t.copy()
    out[~mask] = 0.0

    cap2 = cap.copy()
    cap2[~mask] = 0.0

    target_total = float(np.sum(t[mask]))
    out = np.minimum(out, cap2)
    cur_total = float(np.sum(out[mask]))
    excess = target_total - cur_total

    if excess <= 1e-6:
        return out, 0.0

    for _ in range(MAX_REDIST_ITERS):
        slack = np.clip(cap2 - out, 0.0, None)
        slack_sum = float(np.sum(slack[mask]))
        if slack_sum <= 1e-6:
            break
        add = slack / slack_sum * excess
        add[~mask] = 0.0
        out2 = np.minimum(out + add, cap2)
        new_total = float(np.sum(out2[mask]))
        new_excess = target_total - new_total
        out = out2
        if new_excess <= 1e-6:
            return out, 0.0
        excess = new_excess

    return out, float(max(excess, 0.0))

def _compute_beta_cycle_features(panel: pd.DataFrame, cyc_key: list[str], needed: list[str]) -> pd.DataFrame:
    """

```

```

Build cycle-level feature frame to satisfy beta model's expected feature names.
Handles patterns:
- col__mean, col__sum, col__last
- real_total (filled NaN; beta model may tolerate missing)
- any base col present (mean over harvest)
"""
cols = cyc_key + ["fecha"] + NUM_COLS_DAILY + CAT_COLS
cols = list(dict.fromkeys(cols)) # dedupe preserve order
df = panel.loc[panel["is_harvest"], cols].copy()

# Ensure numeric for daily cols
for c in NUM_COLS_DAILY:
    if c not in df.columns:
        df[c] = np.nan
    df[c] = pd.to_numeric(df[c], errors="coerce")

# Sort for __last
df = df.sort_values(cyc_key + ["fecha"], kind="mergesort")

g = df.groupby(cyc_key, dropna=False)

out = g[[]].size().rename("_n").reset_index().drop(columns=["_n"])

# Build each needed numeric feature
for f in needed:
    if f in out.columns:
        continue

    if f == "real_total":
        out[f] = np.nan
        continue

    if "__" in f:
        base, agg = f.split("__", 1)
        if base not in df.columns:
            out[f] = np.nan
            continue
        if agg == "mean":
            out[f] = g[base].mean().reset_index(drop=True).to_numpy(dtype=float)
        elif agg == "sum":
            out[f] = g[base].sum().reset_index(drop=True).to_numpy(dtype=float)
        elif agg == "last":
            out[f] = (
                g[base]
                .apply(lambda s: float(pd.to_numeric(s, errors="coerce").dropna().iloc[-1]) if pd.to_numeric(s, errors="coerce").dropna().shape[0] > 0 else np.nan)
                .reset_index(drop=True)
                .to_numpy(dtype=float)
            )

        else:
            out[f] = np.nan
    else:
        # base numeric col: use mean over harvest
        if f in df.columns:
            out[f] = g[f].mean().reset_index(drop=True).to_numpy(dtype=float)
        else:
            out[f] = np.nan

# Attach cats (take first non-null per group)
for c in CAT_COLS:
    if c not in out.columns:
        out[c] = (
            g[c].apply(lambda s: _canon_str(s.dropna()).iloc[0] if s.notna().any() else "UNKNOWN")
            .reset_index(drop=True)
        )
    out[c] = _canon_str(out[c].fillna("UNKNOWN"))

return out

```

```

# =====
# Main
# =====
def main(beta_version: str | None = None, mult_version: str | None = None) -> None:
    # ---- Load beta params models
    beta_dir = _latest_version_dir(REG_BETA_PARAMS) if beta_version is None else (REG_BETA_PARAMS / beta_version)
    if not beta_dir.exists():
        raise FileNotFoundError(f"No existe beta version: {beta_dir}")

    metrics_beta = beta_dir / "metrics.json"
    if not metrics_beta.exists():
        raise FileNotFoundError(f"No encontré metrics.json en {beta_dir}")

    with open(metrics_beta, "r", encoding="utf-8") as f:
        meta_beta = json.load(f)

    model_a = load(beta_dir / "model_alpha.joblib")
    model_b = load(beta_dir / "model_beta.joblib")

    # ---- Load multiplier model
    mult_dir = _latest_version_dir(REG_MULT_DIA) if mult_version is None else (REG_MULT_DIA / mult_version)
    if not mult_dir.exists():
        raise FileNotFoundError(f"No existe mult version: {mult_dir}")

    metrics_mult = mult_dir / "metrics.json"
    if not metrics_mult.exists():
        raise FileNotFoundError(f"No encontré metrics.json en {mult_dir}")

    with open(metrics_mult, "r", encoding="utf-8") as f:
        meta_mult = json.load(f)

    model_mult = load(mult_dir / "model_log_mult.joblib")

    # ---- Load cap table
    if not CAP_PATH.exists():
        raise FileNotFoundError(f"No existe cap table: {CAP_PATH}")
    cap = read_parquet(CAP_PATH).copy()
    for c in ["area", "tipo_sp", "variedad_canon"]:
        if c not in cap.columns:
            raise ValueError(f"cap table: falta columna {c}")
        cap[c] = _canon_str(cap[c].fillna("UNKNOWN"))
    if "cap_dia" not in cap.columns:
        raise ValueError("cap table: falta cap_dia")
    cap["cap_dia"] = pd.to_numeric(cap["cap_dia"], errors="coerce").fillna(CAP_FALLBACK).astype(float)

    # ---- Load inputs
    feat = _dedupe_columns(read_parquet(FEATURES_PATH).copy())
    uni = read_parquet(UNIVERSE_PATH).copy()

    # Canon keys
    for df in (feat, uni):
        df["ciclo_id"] = df["ciclo_id"].astype(str)
        df["fecha"] = _to_date(df["fecha"])
        df["bloque_base"] = _canon_int(df["bloque_base"])
        df["variedad_canon"] = _canon_str(df["variedad_canon"])

    for c in ["area", "tipo_sp"]:
        if c not in feat.columns:
            feat[c] = "UNKNOWN"
        feat[c] = _canon_str(feat[c].fillna("UNKNOWN"))

    # Coalesce harvest position columns
    _coalesce_cols(feat, "day_in_harvest", ["day_in_harvest", "day_in_harvest_pred", "day_in_harvest_pred_final"])
    _coalesce_cols(feat, "rel_pos", ["rel_pos", "rel_pos_pred", "rel_pos_pred_final"])
    _coalesce_cols(feat, "n_harvest_days", ["n_harvest_days", "n_harvest_days_pred", "n_harvest_days_pred_final"])

    for c in ["day_in_harvest", "rel_pos", "n_harvest_days"]:
        feat[c] = pd.to_numeric(feat[c], errors="coerce")

```

```

if "tallos_pred_baseline_dia" not in feat.columns:
    raise ValueError("features_curva: falta tallos_pred_baseline_dia")
feat["tallos_pred_baseline_dia"] = pd.to_numeric(feat["tallos_pred_baseline_dia"], errors="coerce").fillna(0.0)

if "tallos_proy" in feat.columns:
    feat["tallos_proy"] = pd.to_numeric(feat["tallos_proy"], errors="coerce").fillna(0.0)
else:
    feat["tallos_proy"] = 0.0

# Ensure daily cols exist
for c in NUM_COLS_DAILY:
    if c not in feat.columns:
        feat[c] = np.nan
for c in CAT_COLS:
    if c not in feat.columns:
        feat[c] = "UNKNOWN"

# Universe panel
key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
uni_k = uni[key].drop_duplicates()
take = list(dict.fromkeys(key + ["tallos_pred_baseline_dia", "tallos_proy"] + NUM_COLS_DAILY + CAT_COLS))
panel = uni_k.merge(feat[take], on=key, how="left")

# Canon cats
panel["variedad_canon"] = _canon_str(panel["variedad_canon"])
panel["area"] = _canon_str(panel["area"].fillna("UNKNOWN"))
panel["tipo_sp"] = _canon_str(panel["tipo_sp"].fillna("UNKNOWN"))

# Harvest mask
dih = pd.to_numeric(panel["day_in_harvest"], errors="coerce")
nh = pd.to_numeric(panel["n_harvest_days"], errors="coerce")
is_h = dih.notna() & nh.notna() & (dih >= 1) & (nh >= 1) & (dih <= nh)
panel["is_harvest"] = is_h
is_h_np = is_h.to_numpy(dtype=bool)

# rel_pos fallback if missing
if panel["rel_pos"].isna().any():
    di2 = pd.to_numeric(panel["day_in_harvest"], errors="coerce").astype(float)
    nh2 = pd.to_numeric(panel["n_harvest_days"], errors="coerce").astype(float)
    panel["rel_pos"] = np.clip((di2 - 0.5) / (nh2 + EPS), REL_CLIP, 1.0 - REL_CLIP)

# =====
# 1) Predict alpha/beta (cycle-level) using the beta model's expected features
# =====
feat_names_beta = meta_beta.get("feature_names", [])
num_cols_beta = meta_beta.get("feature_cols_numeric", [])
if not feat_names_beta:
    raise ValueError("beta metrics.json no contiene feature_names")
if not isinstance(num_cols_beta, list):
    num_cols_beta = []

cyc_key = ["ciclo_id", "bloque_base", "variedad_canon", "area", "tipo_sp"]
cyc = _compute_beta_cycle_features(panel, cyc_key=cyc_key, needed=num_cols_beta)

# Build X for beta (dummies + align)
Xb = cyc[num_cols_beta + CAT_COLS].copy() if num_cols_beta else cyc[CAT_COLS].copy()
for c in num_cols_beta:
    if c not in Xb.columns:
        Xb[c] = np.nan
    Xb[c] = pd.to_numeric(Xb[c], errors="coerce")
for c in CAT_COLS:
    if c not in Xb.columns:
        Xb[c] = "UNKNOWN"
    Xb[c] = _canon_str(Xb[c].fillna("UNKNOWN"))

Xb = pd.get_dummies(Xb, columns=CAT_COLS, dummy_na=True)
for c in feat_names_beta:
    if c not in Xb.columns:
        Xb[c] = 0.0
Xb = Xb[feat_names_beta]

```

```

z_a = model_a.predict(Xb)
z_b = model_b.predict(Xb)

alpha = 1.0 + np.exp(np.clip(pd.to_numeric(pd.Series(z_a), errors="coerce").fillna(0.0).to_numpy(dtype=float), -6, 6))
beta = 1.0 + np.exp(np.clip(pd.to_numeric(pd.Series(z_b), errors="coerce").fillna(0.0).to_numpy(dtype=float), -6, 6))

cyc["alpha_pred"] = np.clip(alpha, 1.05, 100.0)
cyc["beta_pred"] = np.clip(beta, 1.05, 100.0)

panel = panel.merge(
    cyc[cyc_key + ["alpha_pred", "beta_pred"]],
    on=cyc_key,
    how="left",
)

# =====
# 2) share_beta per cycle (shape prior)
# =====
panel = panel.sort_values(["ciclo_id", "fecha"], kind="mergesort").reset_index(drop=True)

share_beta = np.zeros(len(panel), dtype=float)
for cid, idx in panel.groupby("ciclo_id", dropna=False).indices.items():
    ii = np.array(list(idx), dtype=int)
    m = panel.loc[ii, "is_harvest"].to_numpy(dtype=bool)
    if not m.any():
        continue
    rel = panel.loc[ii[m], "rel_pos"].to_numpy(dtype=float)
    a = float(pd.to_numeric(panel.loc[ii[m], "alpha_pred"], errors="coerce").dropna().iloc[0]) if panel.loc[ii[m], "alpha_pred"].notna().any() else 2.0
    b = float(pd.to_numeric(panel.loc[ii[m], "beta_pred"], errors="coerce").dropna().iloc[0]) if panel.loc[ii[m], "beta_pred"].notna().any() else 2.0
    a = max(a, 1.05)
    b = max(b, 1.05)
    sh = _beta_share(rel, a, b)
    share_beta[ii[m]] = sh

# =====
# 3) baseline share (harvest) for robust mixing
# =====
baseline = pd.to_numeric(panel["tallos_pred_baseline_dia"], errors="coerce").fillna(0.0).to_numpy(dtype=float)
b_h = np.where(is_h_np, baseline, 0.0)
sb = pd.Series(b_h).groupby(panel["ciclo_id"], dropna=False).transform("sum").to_numpy(dtype=float)
base_share = np.where(sb > 0, b_h / sb, 0.0)

# =====
# 4) Predict daily multiplier (clima/GDC/etc)
# =====
feat_names_mult = meta_mult.get("feature_names", [])
if not feat_names_mult:
    raise ValueError("mult metrics.json no contiene feature_names")

Xd = panel[NUM_COLS_DAILY + CAT_COLS].copy()
for c in NUM_COLS_DAILY:
    Xd[c] = pd.to_numeric(Xd[c], errors="coerce")
for c in CAT_COLS:
    Xd[c] = _canon_str(Xd[c].fillna("UNKNOWN"))

Xd = pd.get_dummies(Xd, columns=CAT_COLS, dummy_na=True)
for c in feat_names_mult:
    if c not in Xd.columns:
        Xd[c] = 0.0
Xd = Xd[feat_names_mult]

log_mult = model_mult.predict(Xd)
log_mult = pd.to_numeric(pd.Series(log_mult), errors="coerce").fillna(0.0).to_numpy(dtype=float)
log_mult = np.clip(log_mult, LOG_MULT_CLIP[0], LOG_MULT_CLIP[1])
mult = np.exp(log_mult)
mult = np.where(is_h_np, mult, 0.0)

```

```

# =====
# 5) Combine shares: share ? share_beta * mult, plus small baseline mixing
# =====
share_raw = share_beta * mult
share_raw = np.where(is_h_np, share_raw, 0.0)

lam = float(LAMBDA_BASE)
share_raw = np.where(is_h_np, (1.0 - lam) * share_raw + lam * base_share, 0.0)

s0 = pd.Series(share_raw).groupby(panel["ciclo_id"], dropna=False).transform("sum").to_numpy(dtype=float)
share = np.where(s0 > 0, share_raw / s0, base_share)
panel["share_curva_mll"] = share

panel["share_smooth"] = _smooth_share_centered(panel, "share_curva_mll", "is_harvest")

# =====
# 6) Convert to tallos/day using cyc_total (tallos_proy preferred)
# =====
tproy = pd.to_numeric(panel["tallos_proy"], errors="coerce").fillna(0.0).to_numpy(dtype=float)
cyc_tproy = pd.Series(tproy).groupby(panel["ciclo_id"], dropna=False).transform("max").to_numpy(dtype=float)
t)
cyc_total = np.where(cyc_tproy > 0, cyc_tproy, sb)

tallos_pre_cap = cyc_total * panel["share_smooth"].to_numpy(dtype=float)

# =====
# 7) Join cap and apply cap + redistribute within each cycle
# =====
panel = panel.merge(
    cap[["area", "tipo_sp", "variedad_canon", "cap_dia"]].drop_duplicates(),
    on=["area", "tipo_sp", "variedad_canon"],
    how="left",
)
cap_dia = pd.to_numeric(panel["cap_dia"], errors="coerce").fillna(CAP_FALLBACK).to_numpy(dtype=float)

tallos_cap = np.zeros(len(panel), dtype=float)
uncovered = np.zeros(len(panel), dtype=float) # store per row as 0; aggregate per cycle later

for cid, idx in panel.groupby("ciclo_id", dropna=False).indices.items():
    ii = np.array(list(idx), dtype=int)
    m = panel.loc[ii, "is_harvest"].to_numpy(dtype=bool)
    if not m.any():
        continue
    tc, exc = _apply_cap_and_redistribute(tallos_pre_cap[ii], cap_dia[ii], m)
    tallos_cap[ii] = tc
    if exc > 0:
        uncovered[ii[m]] = exc # mark on harvest rows

# Post-cap share (for audit)
share_post_cap = np.zeros(len(panel), dtype=float)
for cid, idx in panel.groupby("ciclo_id", dropna=False).indices.items():
    ii = np.array(list(idx), dtype=int)
    m = panel.loc[ii, "is_harvest"].to_numpy(dtype=bool)
    if not m.any():
        continue
    tot2 = float(np.sum(tallos_cap[ii][m]))
    if tot2 > 0:
        share_post_cap[ii[m]] = tallos_cap[ii][m] / tot2

# =====
# 8) Factor vs baseline (harvest) using baseline_h_dia = cyc_total * base_share
# =====
# factor = peso diario final (share), ya capeado
factor_raw = np.where(is_h_np, share_post_cap, 0.0).astype(float)
factor = factor_raw.copy() # ya está en [0,1] y suma 1 por ciclo (harvest)

# flags opcionales
was_capped_factor = np.zeros(len(panel), dtype="int8")

factor = np.where(np.isfinite(factor), factor, 1.0)

```

```

was_capped_factor = ((factor_raw < FACTOR_MIN) | (factor_raw > FACTOR_MAX)).astype("int8")

# =====
# Output
# =====
out = panel[key].copy()
out["factor_curva_mll_raw"] = factor_raw
out["factor_curva_mll"] = factor
out["mll_version"] = f"beta={beta_dir.name}|mult={mult_dir.name}"
out["created_at"] = pd.Timestamp.utcnow()

# Audit fields
out["alpha_pred"] = pd.to_numeric(panel["alpha_pred"], errors="coerce")
out["beta_pred"] = pd.to_numeric(panel["beta_pred"], errors="coerce")
out["log_mult_pred"] = log_mult
out["mult_pred"] = mult

out["share_beta"] = share_beta
out["share_base"] = base_share
out["share_pre_cap"] = panel["share_smooth"].to_numpy(dtype=float)
out["tallos_pred_mll_dia_pre_cap"] = tallos_pre_cap
out["cap_dia"] = cap_dia
out["tallos_pred_mll_dia_cap"] = tallos_cap
out["share_post_cap"] = share_post_cap
out["uncovered_excess_cycle"] = uncovered # non-zero if not enough slack in cycle
out["was_capped_factor"] = was_capped_factor
out["factor_curva_mll_raw"] = factor_raw
out["factor_curva_mll"] = factor

out = out.sort_values(["ciclo_id", "bloque_base", "variedad_canon", "fecha"]).reset_index(drop=True)

# Duplicate-column hard check before writing
cols = pd.Index(out.columns.astype(str))
if not cols.is_unique:
    dup = cols[cols.duplicated()].unique().tolist()
    raise ValueError(f"[FATAL] columnas duplicadas en OUT: {dup}")

write_parquet(out, OUT_PATH)
print(f"OK -> {OUT_PATH} | rows={len(out):,} | version={out['mll_version'].iloc[0] if len(out) else ''}")

if __name__ == "__main__":
    main(beta_version=None, mult_version=None)

-----
[79/164] FILE: \src\models\mll\apply_curva_cdf_dia.py
-----

from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

from common.io import read_parquet, write_parquet

# =====
# Paths
# =====
FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_mll.parquet")
REGISTRY_ROOT = Path("models_registry/mll/curva_cdf_dia")
OUT_PATH = Path("data/gold/pred_factor_curva_mll.parquet")

# =====
# Model columns (must match training)
# =====
NUM_COLS = [
    "day_in_harvest",
    "rel_pos",

```

```

    "n_harvest_days",
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
]
CAT_COLS = ["variedad_canon", "area", "tipo_sp"]
CAT_COLS_MERGE = ["area", "tipo_sp"]

# =====
# Statistical adjustments (NOT capacity)
# =====
SMOOTH_WIN = 5    # rolling mean centered on share (per cycle)
EPS = 1e-12
FACTOR_MIN, FACTOR_MAX = 0.2, 5.0

# =====
# Helpers
# =====
def _latest_version_dir() -> Path:
    if not REGISTRY_ROOT.exists():
        raise FileNotFoundError(f"No existe {REGISTRY_ROOT}")
    dirs = [p for p in REGISTRY_ROOT.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {REGISTRY_ROOT}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _coalesce_cols(df: pd.DataFrame, out_col: str, candidates: list[str]) -> None:
    if out_col in df.columns:
        base = df[out_col]
    else:
        base = pd.Series([pd.NA] * len(df), index=df.index)

    for c in candidates:
        if c in df.columns:
            base = base.where(base.notna(), df[c])
    df[out_col] = base

def _dedupe_columns(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.Index(df.columns.astype(str))
    if cols.is_unique:
        return df

    out = df.copy()
    seen: dict[str, list[int]] = {}
    for i, c in enumerate(out.columns.astype(str)):

```



```

        seen.setdefault(c, []).append(i)

keep_series: dict[str, pd.Series] = {}
for c, idxs in seen.items():
    if len(idxs) == 1:
        keep_series[c] = out.iloc[:, idxs[0]]
    else:
        s = out.iloc[:, idxs[0]]
        for j in idxs[1:]:
            s2 = out.iloc[:, j]
            s = s.where(s.notna(), s2)
        keep_series[c] = s

ordered: list[str] = []
for c in out.columns.astype(str):
    if c not in ordered:
        ordered.append(c)

return pd.DataFrame({c: keep_series[c] for c in ordered})

def _baseline_share_and_cdf(tmp: pd.DataFrame, is_h_np: np.ndarray) -> tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    baseline_share: b_h / sum(b_h) per cycle (harvest only)
    baseline_cdf: cumsum(baseline_share) per cycle (already sorted)
    sb: sum baseline harvest per cycle (transform)
    """
    baseline = pd.to_numeric(tmp["tallos_pred_baseline_dia"], errors="coerce").fillna(0.0).to_numpy(dtype=float)
    b_h = np.where(is_h_np, baseline, 0.0)

    sb = pd.Series(b_h).groupby(tmp["ciclo_id"], dropna=False).transform("sum").to_numpy(dtype=float)
    base_share = np.where(sb > 0, b_h / sb, 0.0)

    base_cdf = pd.Series(base_share).groupby(tmp["ciclo_id"], dropna=False).cumsum().to_numpy(dtype=float)
    base_cdf = np.clip(base_cdf, 0.0, 1.0)
    return base_share, base_cdf, sb

def _smooth_share_centered(tmp: pd.DataFrame, share_col: str, is_h_col: str) -> np.ndarray:
    """
    Smooth share per cycle (harvest only) via centered rolling mean.
    Then renormalize to sum=1 per cycle (harvest only).
    """
    if SMOOTH_WIN <= 1:
        return tmp[share_col].to_numpy(dtype=float)

    df = tmp[["ciclo_id", share_col, is_h_col]].copy()
    df[share_col] = pd.to_numeric(df[share_col], errors="coerce").fillna(0.0).clip(lower=0.0)
    df.loc[~df[is_h_col], share_col] = 0.0

    sm = (
        df.groupby("ciclo_id", dropna=False)[share_col]
        .rolling(window=SMOOTH_WIN, center=True, min_periods=1)
        .mean()
        .reset_index(level=0, drop=True)
    ).to_numpy(dtype=float)

    is_h = df[is_h_col].to_numpy(dtype=bool)
    sm = np.where(is_h, sm, 0.0)

    s = pd.Series(sm).groupby(df["ciclo_id"], dropna=False).transform("sum").to_numpy(dtype=float)
    sm = np.where(s > 0, sm / s, sm)
    return sm

def _first_nonnull(s: pd.Series) -> float:
    s2 = s.dropna()
    return float(s2.iloc[0]) if len(s2) else np.nan

```

```

def _last_nonnull(s: pd.Series) -> float:
    s2 = s.dropna()
    return float(s2.iloc[-1]) if len(s2) else np.nan

# =====
# Main
# =====
def main(version: str | None = None) -> None:
    ver_dir = _latest_version_dir() if version is None else (REGISTRY_ROOT / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe la versión: {ver_dir}")

    metrics_path = ver_dir / "metrics.json"
    if not metrics_path.exists():
        raise FileNotFoundError(f"No encontré metrics.json en {ver_dir}")

    with open(metrics_path, "r", encoding="utf-8") as f:
        meta = json.load(f)

    model_path = ver_dir / "model_curva_cdf_dia.joblib"
    if not model_path.exists():
        raise FileNotFoundError(f"No encontré modelo: {model_path}")
    model = load(model_path)

    feat = _dedupe_columns(read_parquet(FEATURES_PATH).copy())
    uni = read_parquet(UNIVERSE_PATH).copy()

    # -----
    # Canon keys
    # -----
    feat["ciclo_id"] = feat["ciclo_id"].astype(str)
    feat["fecha"] = _to_date(feat["fecha"])
    feat["bloque_base"] = _canon_int(feat["bloque_base"])
    feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
    for c in ["area", "tipo_sp"]:
        if c in feat.columns:
            feat[c] = _canon_str(feat[c])

    _coalesce_cols(feat, "day_in_harvest", ["day_in_harvest", "day_in_harvest_pred", "day_in_harvest_pred_final"])
    _coalesce_cols(feat, "rel_pos", ["rel_pos", "rel_pos_pred", "rel_pos_pred_final"])
    _coalesce_cols(feat, "n_harvest_days", ["n_harvest_days", "n_harvest_days_pred", "n_harvest_days_pred_final"])

    for c in ["day_in_harvest", "rel_pos", "n_harvest_days"]:
        feat[c] = pd.to_numeric(feat[c], errors="coerce")

    if "tallos_pred_baseline_dia" not in feat.columns:
        raise ValueError("features_curva: falta tallos_pred_baseline_dia")
    feat["tallos_pred_baseline_dia"] = pd.to_numeric(feat["tallos_pred_baseline_dia"], errors="coerce").fillna(0.0)

    if "tallos_proy" in feat.columns:
        feat["tallos_proy"] = pd.to_numeric(feat["tallos_proy"], errors="coerce").fillna(0.0)
    else:
        feat["tallos_proy"] = 0.0

    # Ensure model cols exist
    for c in NUM_COLS:
        if c not in feat.columns:
            feat[c] = np.nan
    for c in CAT_COLS:
        if c not in feat.columns:
            feat[c] = "UNKNOWN"

    # Universe
    uni["ciclo_id"] = uni["ciclo_id"].astype(str)
    uni["fecha"] = _to_date(uni["fecha"])
    uni["bloque_base"] = _canon_int(uni["bloque_base"])
    uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

```

```

key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
uni_k = uni[key].drop_duplicates()

feat_take = key + ["tallos_pred_baseline_dia", "tallos_proy"] + NUM_COLS + CAT_COLS_MERGE
feat_take = list(dict.fromkeys(feat_take))
panel = uni_k.merge(feat[feat_take], on=key, how="left")

# Fill defaults for missing matches
for c in NUM_COLS:
    if c not in panel.columns:
        panel[c] = np.nan
for c in CAT_COLS:
    if c not in panel.columns:
        panel[c] = "UNKNOWN"

panel["variedad_canon"] = _canon_str(panel["variedad_canon"])
if "area" in panel.columns:
    panel["area"] = _canon_str(panel["area"].fillna("UNKNOWN"))
if "tipo_sp" in panel.columns:
    panel["tipo_sp"] = _canon_str(panel["tipo_sp"].fillna("UNKNOWN"))

# Harvest mask
dih = pd.to_numeric(panel["day_in_harvest"], errors="coerce")
nh = pd.to_numeric(panel["n_harvest_days"], errors="coerce")
is_h = dih.notna() & nh.notna() & (dih >= 1) & (nh >= 1) & (dih <= nh)
is_h_np = is_h.to_numpy(dtype=bool)

# One-hot aligned with training
X = panel[NUM_COLS + CAT_COLS].copy()
X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)

feat_names = meta.get("feature_names", [])
if not feat_names:
    raise ValueError("metrics.json no contiene feature_names (necesario para alinear dummies).")

for c in feat_names:
    if c not in X.columns:
        X[c] = 0.0
X = X[feat_names]

# Predict CDF
cdf_pred = model.predict(X)
cdf_pred = pd.to_numeric(pd.Series(cdf_pred), errors="coerce").fillna(0.0).to_numpy(dtype=float)
cdf_pred = np.clip(cdf_pred, 0.0, 1.0)
cdf_pred = np.where(is_h_np, cdf_pred, 0.0)

tmp = panel[key + ["tallos_pred_baseline_dia", "tallos_proy", "day_in_harvest", "rel_pos", "n_harvest_days"]].copy()
tmp["mll_version"] = ver_dir.name
tmp["cdf_pred_raw"] = cdf_pred
tmp["is_harvest"] = is_h_np

# Sort within cycle
tmp["_dih_sort"] = pd.to_numeric(tmp["day_in_harvest"], errors="coerce")
tmp["_sort_key"] = np.where(
    tmp["_dih_sort"].notna(),
    tmp["_dih_sort"].astype(float),
    tmp["fecha"].astype("int64").astype(float),
)
tmp = tmp.sort_values(["ciclo_id", "_sort_key"], kind="mergesort").reset_index(drop=True)

# Monotone
tmp["cdf_pred_mono"] = tmp.groupby("ciclo_id", dropna=False)["cdf_pred_raw"].cummax().clip(0.0, 1.0)

# Baseline fallback (already sorted)
base_share, base_cdf, sb = _baseline_share_and_cdf(tmp, tmp["is_harvest"].to_numpy(dtype=bool))

# Anchor within harvest: (cdf - start)/(end-start)
cdf_h = tmp["cdf_pred_mono"].where(tmp["is_harvest"])
cdf_start = cdf_h.groupby(tmp["ciclo_id"], dropna=False).transform(_first_nonnull).to_numpy(dtype=float)
cdf_end = cdf_h.groupby(tmp["ciclo_id"], dropna=False).transform(_last_nonnull).to_numpy(dtype=float)
denom = cdf_end - cdf_start

```

```

cdf_adj = (tmp["cdf_pred_mono"].to_numpy(dtype=float) - cdf_start) / (denom + EPS)
cdf_adj = np.clip(cdf_adj, 0.0, 1.0)

use_model = tmp["is_harvest"].to_numpy(dtype=bool) & np.isfinite(denom) & (denom > 1e-6)
cdf_final = np.where(use_model, cdf_adj, base_cdf)
cdf_final = np.where(tmp["is_harvest"].to_numpy(dtype=bool), cdf_final, 0.0)

tmp["cdf_pred_adj"] = cdf_final
tmp["cdf_pred_adj"] = tmp.groupby("ciclo_id", dropna=False)["cdf_pred_adj"].cummax().clip(0.0, 1.0)

# Force end=1 inside harvest (if harvest exists)
cdf_end2 = tmp["cdf_pred_adj"].where(tmp["is_harvest"]).groupby(tmp["ciclo_id"], dropna=False).transform(_
last_nonnull).to_numpy(dtype=float)
scale = np.where(np.isfinite(cdf_end2) & (cdf_end2 > 1e-6), cdf_end2, 1.0)
tmp["cdf_pred_adj"] = np.where(tmp["is_harvest"], tmp["cdf_pred_adj"] / scale, 0.0)
tmp["cdf_pred_adj"] = tmp["cdf_pred_adj"].clip(0.0, 1.0)

# Share = diff(CDF)
tmp["share_pred_in"] = tmp.groupby("ciclo_id", dropna=False)["cdf_pred_adj"].diff()
tmp["share_pred_in"] = tmp["share_pred_in"].fillna(tmp["cdf_pred_adj"]).astype(float)
tmp["share_pred_in"] = tmp["share_pred_in"].clip(lower=0.0)
tmp["share_pred_in"] = np.where(tmp["is_harvest"].to_numpy(dtype=bool), tmp["share_pred_in"].to_numpy(dtype=
e=float), 0.0)

# Renormalize per cycle (harvest); if sum=0 fallback baseline share
s = tmp.groupby("ciclo_id", dropna=False)["share_pred_in"].transform("sum").astype(float).to_numpy()
share = np.where(s > 0, tmp["share_pred_in"].to_numpy(dtype=float) / s, base_share)
tmp["share_curva_mll"] = share

# Smooth
tmp["share_smooth"] = _smooth_share_centered(tmp, "share_curva_mll", "is_harvest")

# Total per cycle: prefer tallos_proy if >0 else sum baseline harvest (sb)
tproy = pd.to_numeric(tmp["tallos_proy"], errors="coerce").fillna(0.0).to_numpy(dtype=float)
cyc_tproy = pd.Series(tproy).groupby(tmp["ciclo_id"], dropna=False).transform("max").to_numpy(dtype=float)
cyc_total = np.where(cyc_tproy > 0, cyc_tproy, sb)

tallos_mll_dia = cyc_total * tmp["share_smooth"].to_numpy(dtype=float)
tmp["tallos_pred_mll_dia_from_cdf"] = tallos_mll_dia

baseline = pd.to_numeric(tmp["tallos_pred_baseline_dia"], errors="coerce").fillna(0.0).to_numpy(dtype=floa
t)

factor_raw = np.where(tmp["is_harvest"].to_numpy(dtype=bool), tallos_mll_dia / (baseline + 1e-9), 1.0)
factor = np.clip(factor_raw, FACTOR_MIN, FACTOR_MAX)
factor = np.where(np.isfinite(factor), factor, 1.0)

# Flags (FIX: numpy dtype, NOT "Int64")
was_capped_pre = np.where((factor_raw < FACTOR_MIN) | (factor_raw > FACTOR_MAX), 1, 0).astype("int8")
was_capped_post = np.where((factor < FACTOR_MIN) | (factor > FACTOR_MAX), 1, 0).astype("int8")

# Output
out = tmp[key].copy()
out["factor_curva_mll_raw"] = factor_raw
out["factor_curva_mll"] = factor
out["mll_version"] = ver_dir.name
out["created_at"] = pd.Timestamp.utcnow()

# Audit columns (keep names compatible with tus auditorías)
out["_share_source"] = np.where(use_model, "cdf_adj", np.where(sb > 0, "baseline", "zero"))
out["cap_share"] = np.nan # placeholder (si luego metes cap/floor por bins)
out["share_pred_in"] = tmp["share_pred_in"].to_numpy(dtype=float)
out["share_smooth"] = tmp["share_smooth"].to_numpy(dtype=float)
out["share_curva_mll"] = tmp["share_curva_mll"].to_numpy(dtype=float)
out["tallos_pred_mll_dia_smooth"] = tmp["tallos_pred_mll_dia_from_cdf"].to_numpy(dtype=float)
out["factor_curva_mll_raw_smooth"] = factor_raw # aquí factor_raw ya viene del share_smooth
out["was_capped_pre"] = was_capped_pre
out["was_capped_post"] = was_capped_post

out = out.sort_values(["bloque_base", "variedad_canon", "fecha"]).reset_index(drop=True)
write_parquet(out, OUT_PATH)

```

```

print(f"OK -> {OUT_PATH} | rows={len(out):,} | version={ver_dir.name}")

if __name__ == "__main__":
    main(version=None)

-----
[80/164] FILE: \src\models\ml1\apply_curva_share_dia.py
-----

from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

from common.io import read_parquet, write_parquet

# =====
# Paths
# =====
FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_ml1.parquet")

REGISTRY_ROOT = Path("models_registry/ml1/curva_share_dia")

OUT_PATH = Path("data/gold/pred_factor_curva_ml1.parquet")

# =====
# Model columns (must match TRAIN)
# =====
NUM_COLS = [
    "day_in_harvest",
    "rel_pos",
    "n_harvest_days",
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
]

CAT_COLS = ["variedad_canon", "area", "tipo_sp"]

# =====
# Helpers
# =====
def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    # OJO: pandas nullable Int64 (capital I) existe; numpy int64 no sirve si hay NA.

```

```

return pd.to_numeric(s, errors="coerce").astype("Int64")

def _latest_version_dir() -> Path:
    if not REGISTRY_ROOT.exists():
        raise FileNotFoundError(f"No existe {REGISTRY_ROOT}")
    dirs = [p for p in REGISTRY_ROOT.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {REGISTRY_ROOT}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _coalesce_cols(df: pd.DataFrame, out_col: str, candidates: list[str]) -> None:
    if out_col in df.columns:
        base = df[out_col]
    else:
        base = pd.Series([pd.NA] * len(df), index=df.index)

    for c in candidates:
        if c in df.columns:
            base = base.where(base.notna(), df[c])
    df[out_col] = base

def _dedupe_columns(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.Index(df.columns.astype(str))
    if cols.is_unique:
        return df

    out = df.copy()
    seen: dict[str, list[int]] = {}
    for i, c in enumerate(out.columns.astype(str)):
        seen.setdefault(c, []).append(i)

    keep: dict[str, pd.Series] = {}
    for c, idxs in seen.items():
        if len(idxs) == 1:
            keep[c] = out.iloc[:, idxs[0]]
        else:
            s = out.iloc[:, idxs[0]]
            for j in idxs[1:]:
                s2 = out.iloc[:, j]
                s = s.where(s.notna(), s2)
            keep[c] = s

    ordered: list[str] = []
    for c in out.columns.astype(str):
        if c not in ordered:
            ordered.append(c)

    return pd.DataFrame({c: keep[c] for c in ordered})

def _infer_is_harvest(df: pd.DataFrame) -> np.ndarray:
    """
    Define máscara de días dentro de ventana harvest.
    Preferimos:
    - harvest_start_pred / harvest_end_pred (si existen)
    - day_in_harvest (si existe y es >=1)
    - fallback: baseline > 0
    """
    f = pd.to_datetime(df["fecha"], errors="coerce")

    hs_col = None
    he_col = None
    for c in ["harvest_start_pred", "harvest_start", "fecha_inicio_real"]:

```

```

        if c in df.columns:
            hs_col = c
            break
    for c in ["harvest_end_pred", "harvest_end_eff", "harvest_end", "fecha_fin_real"]:
        if c in df.columns:
            he_col = c
            break

    if hs_col and he_col:
        hs = pd.to_datetime(df[hs_col], errors="coerce")
        he = pd.to_datetime(df[he_col], errors="coerce")
        m = hs.notna() & he.notna() & f.notna() & (f >= hs) & (f <= he)
        return m.to_numpy(dtype=bool)

    if "day_in_harvest" in df.columns:
        dih = pd.to_numeric(df["day_in_harvest"], errors="coerce")
        m = dih.notna() & (dih.astype(float) >= 1)
        return m.to_numpy(dtype=bool)

    base = pd.to_numeric(df.get("tallos_pred_baseline_dia", 0.0), errors="coerce").fillna(0.0)
    return (base > 0).to_numpy(dtype=bool)

def _make_relpos_bin(rel_pos: pd.Series) -> pd.Series:
    """
    Bins 0..1 en 6 tramos (consistente con lo que te está saliendo: global rows=6).
    """
    x = pd.to_numeric(rel_pos, errors="coerce")
    bins = [-np.inf, 0.0, 0.10, 0.25, 0.50, 0.75, np.inf]
    labels = ["B0", "B1", "B2", "B3", "B4", "B5"]
    return pd.cut(x, bins=bins, labels=labels, include_lowest=True).astype("object")

def _safe_load_cap_tables(ver_dir: Path) -> tuple[pd.DataFrame | None, pd.DataFrame | None]:
    """
    Intenta cargar:
    - cap_floor_share_by_relpos.parquet (segmentado)
    - cap_floor_share_by_relpos_global.parquet (fallback global)
    desde la carpeta de versión.
    """
    seg = ver_dir / "cap_floor_share_by_relpos.parquet"
    glb = ver_dir / "cap_floor_share_by_relpos_global.parquet"
    seg_df = read_parquet(seg) if seg.exists() else None
    glb_df = read_parquet(glb) if glb.exists() else None
    return seg_df, glb_df

def _apply_cap_floor(
    df: pd.DataFrame,
    share_col: str,
    seg_caps: pd.DataFrame | None,
    glb_caps: pd.DataFrame | None,
) -> pd.DataFrame:
    """
    Aplica cap/floor por (variedad_canon, area, tipo_sp, rel_pos_bin) y fallback global por rel_pos_bin.
    No hardcodea ?punteos/colas?, solo limita extremos estadísticos.
    """
    out = df.copy()

    out[share_col] = pd.to_numeric(out[share_col], errors="coerce").fillna(0.0)
    out[share_col] = out[share_col].clip(lower=0.0)

    # preparar bins
    out["rel_pos_bin"] = _make_relpos_bin(out.get("rel_pos", np.nan))

    cap = pd.Series([np.nan] * len(out), index=out.index)
    floor = pd.Series([np.nan] * len(out), index=out.index)

    # segment caps
    if seg_caps is not None and len(seg_caps):
        s = seg_caps.copy()
        # normalizar nombres esperados mínimos

```

```

for c in ["variedad_canon", "area", "tipo_sp"]:
    if c in s.columns:
        s[c] = _canon_str(s[c])
if "rel_pos_bin" in s.columns:
    s["rel_pos_bin"] = s["rel_pos_bin"].astype("object")

# detectar columnas cap/floor
cap_col = "cap_share" if "cap_share" in s.columns else None
floor_col = "floor_share" if "floor_share" in s.columns else None

keys = [c for c in ["variedad_canon", "area", "tipo_sp", "rel_pos_bin"] if c in s.columns]
if cap_col and keys:
    tmp = out.merge(
        s[keys + [cap_col] + ([floor_col] if floor_col else [])],
        on=keys,
        how="left",
        suffixes=("", "_cap"),
    )
    cap = pd.to_numeric(tmp[cap_col], errors="coerce")
    if floor_col:
        floor = pd.to_numeric(tmp[floor_col], errors="coerce")

# global caps
if glb_caps is not None and len(glb_caps):
    g = glb_caps.copy()
    if "rel_pos_bin" in g.columns:
        g["rel_pos_bin"] = g["rel_pos_bin"].astype("object")
    cap_col_g = "cap_share" if "cap_share" in g.columns else None
    floor_col_g = "floor_share" if "floor_share" in g.columns else None

    if cap_col_g and "rel_pos_bin" in g.columns:
        tmpg = out[["rel_pos_bin"]].merge(g[["rel_pos_bin", cap_col_g] + ([floor_col_g] if floor_col_g else [])],
                                          on="rel_pos_bin", how="left")
        cap_g = pd.to_numeric(tmpg[cap_col_g], errors="coerce")
        floor_g = pd.to_numeric(tmpg[floor_col_g], errors="coerce") if floor_col_g else pd.Series([np.nan]
*len(out), index=out.index)

        cap = cap.where(cap.notna(), cap_g)
        floor = floor.where(floor.notna(), floor_g)

out["cap_share"] = cap
out["floor_share"] = floor

# aplicar
was_cap = pd.Series(False, index=out.index)
was_floor = pd.Series(False, index=out.index)

if out["cap_share"].notna().any():
    m = out[share_col] > out["cap_share"]
    was_cap = was_cap | m.fillna(False)
    out.loc[m.fillna(False), share_col] = out.loc[m.fillna(False), "cap_share"]

if out["floor_share"].notna().any():
    m = out[share_col] < out["floor_share"]
    was_floor = was_floor | m.fillna(False)
    out.loc[m.fillna(False), share_col] = out.loc[m.fillna(False), "floor_share"]

out["was_capped"] = was_cap
out["was_floored"] = was_floor
return out

def _smooth_and_renorm(df: pd.DataFrame, share_col: str, win: int = 5) -> pd.Series:
    """
    Suaviza share por ciclo con rolling mean centrado y renormaliza a sum=1.
    No impone ?porcentajes fijos?; solo reduce picos espurios.
    """
    if win < 3:
        win = 3
    if win % 2 == 0:
        win += 1

```



```

out = np.zeros(len(df), dtype=float)

# asumimos df ya ordenado por ciclo/fecha
grp = df.groupby("ciclo_id", dropna=False, sort=False)
for _, sub in grp:
    idx = sub.index.to_numpy()
    s = pd.to_numeric(sub[share_col], errors="coerce").fillna(0.0).to_numpy(dtype=float)
    if len(s) == 0:
        continue
    # rolling mean centrado
    ss = pd.Series(s).rolling(window=win, center=True, min_periods=max(1, win // 2)).mean().to_numpy(dtype=float)
    ss = np.clip(ss, 0.0, None)
    tot = float(np.nansum(ss))
    if tot > 0:
        ss = ss / tot
    out[idx] = ss

return pd.Series(out, index=df.index, dtype=float)

# =====
# Main
# =====
def main(version: str | None = None) -> None:
    ver_dir = _latest_version_dir() if version is None else (REGISTRY_ROOT / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe la versión: {ver_dir}")

    metrics_path = ver_dir / "metrics.json"
    model_path = ver_dir / "model_curva_share_dia.joblib"
    if not metrics_path.exists():
        raise FileNotFoundError(f"No encontré metrics.json en {ver_dir}")
    if not model_path.exists():
        raise FileNotFoundError(f"No encontré modelo: {model_path}")

    with open(metrics_path, "r", encoding="utf-8") as f:
        meta = json.load(f)

    feature_names = meta.get("feature_names")
    if not feature_names:
        raise ValueError("metrics.json no trae feature_names (necesario para alinear dummies en apply).")

    model = load(model_path)

    # caps
    seg_caps, glb_caps = _safe_load_cap_tables(ver_dir)

    # inputs
    feat = read_parquet(FEATURES_PATH).copy()
    feat = _dedupe_columns(feat)

    _require(
        feat,
        ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia", "tallos_proy"],
        "features_curva",
    )

    feat["ciclo_id"] = feat["ciclo_id"].astype(str)
    feat["fecha"] = _to_date(feat["fecha"])
    feat["bloque_base"] = _canon_int(feat["bloque_base"])
    feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
    for c in ["area", "tipo_sp"]:
        if c in feat.columns:
            feat[c] = _canon_str(feat[c])

    # alias posición
    _coalesce_cols(feat, "day_in_harvest", ["day_in_harvest", "day_in_harvest_pred", "day_in_harvest_pred_final"])
    _coalesce_cols(feat, "rel_pos", ["rel_pos", "rel_pos_pred", "rel_pos_pred_final"])
    _coalesce_cols(feat, "n_harvest_days", ["n_harvest_days", "n_harvest_days_pred", "n_harvest_days_pred_final"])

```

```

1"]])

# asegurar columnas
for c in NUM_COLS:
    if c not in feat.columns:
        feat[c] = np.nan
for c in CAT_COLS:
    if c not in feat.columns:
        feat[c] = "UNKNOWN"

# canon cats (importante antes de dummies)
feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
feat["area"] = _canon_str(feat.get("area", "UNKNOWN")).fillna("UNKNOWN")
feat["tipo_sp"] = _canon_str(feat.get("tipo_sp", "UNKNOWN")).fillna("UNKNOWN")

# numerics
for c in NUM_COLS:
    feat[c] = pd.to_numeric(feat[c], errors="coerce")

feat["tallos_pred_baseline_dia"] = pd.to_numeric(feat["tallos_pred_baseline_dia"], errors="coerce").fillna(0.0).astype(float)
feat["tallos_proy"] = pd.to_numeric(feat["tallos_proy"], errors="coerce").fillna(0.0).astype(float)

# universo
uni = read_parquet(UNIVERSE_PATH).copy()
_require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_mll")
uni["ciclo_id"] = uni["ciclo_id"].astype(str)
uni["fecha"] = _to_date(uni["fecha"])
uni["bloque_base"] = _canon_int(uni["bloque_base"])
uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
uni_k = uni[key].drop_duplicates()

# panel = universe LEFT join feat
panel = uni_k.merge(feat, on=key, how="left", suffixes=("", "_feat"))

# completar cats defaults si quedaron NaN (por universe sin match)
panel["variedad_canon"] = _canon_str(panel["variedad_canon"])
panel["area"] = _canon_str(panel.get("area", "UNKNOWN")).fillna("UNKNOWN")
panel["tipo_sp"] = _canon_str(panel.get("tipo_sp", "UNKNOWN")).fillna("UNKNOWN")

# completar numerics faltantes
for c in NUM_COLS:
    if c not in panel.columns:
        panel[c] = np.nan
    panel[c] = pd.to_numeric(panel[c], errors="coerce")

panel["tallos_pred_baseline_dia"] = pd.to_numeric(panel.get("tallos_pred_baseline_dia", 0.0), errors="coerce").fillna(0.0).astype(float)
panel["tallos_proy"] = pd.to_numeric(panel.get("tallos_proy", 0.0), errors="coerce").fillna(0.0).astype(float)

# infer harvest mask
is_h = _infer_is_harvest(panel)

# X (dummies aligned)
X = panel[NUM_COLS + CAT_COLS].copy()
X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)

# IMPORTANT: align to training feature_names
X = X.reindex(columns=feature_names, fill_value=0)

# predict share raw
share_pred = model.predict(X)
share_pred = pd.to_numeric(pd.Series(share_pred, index=panel.index), errors="coerce").fillna(0.0).to_numpy(dtype=float)
share_pred = np.clip(share_pred, 0.0, None)
share_pred = np.where(is_h, share_pred, 0.0)

panel["share_pred_in"] = share_pred

```

```

# cap/floor PRE
tmp = panel.copy()
tmp["_share_source"] = "model"
tmp["share_curva_mll_raw"] = tmp["share_pred_in"]

tmp = _apply_cap_floor(tmp, "share_curva_mll_raw", seg_caps, glb_caps)
tmp["was_capped_pre"] = tmp["was_capped"]
tmp["was_floored_pre"] = tmp["was_floored"]

# renorm raw share por ciclo (si quedó todo 0, fallback baseline weights)
base = tmp["tallos_pred_baseline_dia"].to_numpy(dtype=float)
base_h = np.where(is_h, base, 0.0)

# sum raw por ciclo
sraw = tmp.groupby("ciclo_id", dropna=False)["share_curva_mll_raw"].transform("sum").to_numpy(dtype=float)
sb = pd.Series(base_h, index=tmp.index).groupby(tmp["ciclo_id"], dropna=False).transform("sum").to_numpy(dtype=float)

share_raw = tmp["share_curva_mll_raw"].to_numpy(dtype=float)
share_ren = np.where(
    sraw > 0,
    share_raw / sraw,
    np.where(sb > 0, base_h / sb, 0.0),
)
tmp["share_curva_mll"] = share_ren

# smooth + renorm
tmp = tmp.sort_values(["ciclo_id", "fecha"], kind="mergesort").reset_index(drop=True)
tmp["share_smooth"] = _smooth_and_renorm(tmp, "share_curva_mll", win=5)

# cap/floor POST sobre share_smooth (opcional pero útil para evitar rebotes negativos/raros)
tmp = _apply_cap_floor(tmp, "share_smooth", seg_caps, glb_caps)
tmp["was_capped_post"] = tmp["was_capped"]
tmp["was_floored_post"] = tmp["was_floored"]

# renorm final post-cap
s2 = tmp.groupby("ciclo_id", dropna=False)["share_smooth"].transform("sum").to_numpy(dtype=float)
tmp["share_smooth"] = np.where(s2 > 0, tmp["share_smooth"].to_numpy(dtype=float) / s2, 0.0)

# cyc_total = tallos_proy max por ciclo (fallback sum baseline)
tproy_max = tmp.groupby("ciclo_id", dropna=False)["tallos_proy"].transform("max").to_numpy(dtype=float)
tproy_max = np.where(np.isfinite(tproy_max), tproy_max, 0.0)
cyc_total = np.where(tproy_max > 0, tproy_max, sb)

tmp["tallos_pred_mll_dia_smooth"] = cyc_total * tmp["share_smooth"].to_numpy(dtype=float)

# factor compatible downstream
eps = 1e-9
tmp["factor_curva_mll_raw_smooth"] = np.where(
    is_h,
    tmp["tallos_pred_mll_dia_smooth"].to_numpy(dtype=float) / (base + eps),
    1.0,
)

FACTOR_MIN, FACTOR_MAX = 0.2, 5.0
tmp["factor_curva_mll_raw"] = tmp["factor_curva_mll_raw_smooth"]
tmp["factor_curva_mll"] = np.clip(tmp["factor_curva_mll_raw"].to_numpy(dtype=float), FACTOR_MIN, FACTOR_MAX)

X) tmp["factor_curva_mll"] = np.where(np.isfinite(tmp["factor_curva_mll"]), tmp["factor_curva_mll"], 1.0)

# metadata
tmp["mll_version"] = ver_dir.name
tmp["created_at"] = pd.Timestamp.utcnow()

# outputs
out_cols = [
    "ciclo_id",
    "fecha",
    "bloque_base",
    "variedad_canon",
    "factor_curva_mll",
    "factor_curva_mll_raw",

```

```

        "mll_version",
        "created_at",
        "_share_source",
        "cap_share",
        "share_pred_in",
        "share_smooth",
        "share_curva_mll",
        "tallos_pred_mll_dia_smooth",
        "factor_curva_mll_raw_smooth",
        "was_capped_pre",
        "was_capped_post",
        "was_floored_pre",
        "was_floored_post",
    ]
    out_cols = [c for c in out_cols if c in tmp.columns]

    out = tmp[out_cols].sort_values(["bloque_base", "variedad_canon", "fecha"], kind="mergesort").reset_index(
drop=True)

    write_parquet(out, OUT_PATH)

    # quick audits
    # share sum by cycle
    ss = out.groupby("ciclo_id", dropna=False)["share_smooth"].sum()
    # mass balance
    tallos_sum = out.groupby("ciclo_id", dropna=False)["tallos_pred_mll_dia_smooth"].sum()
    proy = tmp.groupby("ciclo_id", dropna=False)["tallos_proy"].max()
    max_abs = float((tallos_sum - proy).abs().max()) if len(tallos_sum) else float("nan")

    print(f"OK -> {OUT_PATH} | rows={len(out):,} | version={ver_dir.name}")
    print(f"[CHECK] share_smooth sum min/max: {float(ss.min()):.6f} / {float(ss.max()):.6f}")
    print(f"[CHECK] ciclo mass-balance ML1 vs tallos_proy | max abs diff: {max_abs:.12f}")

if __name__ == "__main__":
    main(version=None)

-----
[81/164] FILE: \src\models\mll\apply_curva_tallos_dia.py
-----
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

from common.io import read_parquet, write_parquet

FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
REGISTRY_ROOT = Path("models_registry/mll/curva_tallos_dia")
OUT_PATH = Path("data/gold/pred_factor_curva_mll.parquet")

NUM_COLS = [
    "tallos_pred_baseline_dia",
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",

```

```

    "month",
    "weekofyear",
]

CAT_COLS = ["variedad_canon", "area", "tipo_sp"]

def _latest_version_dir() -> Path:
    if not REGISTRY_ROOT.exists():
        raise FileNotFoundError(f"No existe {REGISTRY_ROOT}")
    dirs = [p for p in REGISTRY_ROOT.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {REGISTRY_ROOT}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def main(version: str | None = None) -> None:
    ver_dir = _latest_version_dir() if version is None else (REGISTRY_ROOT / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe la versión: {ver_dir}")

    metrics_path = ver_dir / "metrics.json"
    with open(metrics_path, "r", encoding="utf-8") as f:
        meta = json.load(f)

    model_path = ver_dir / "model_curva_tallos_dia.joblib"
    if not model_path.exists():
        raise FileNotFoundError(f"No encontré modelo: {model_path}")

    model = load(model_path)

    df = read_parquet(FEATURES_PATH).copy()

    need = {"fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"FEATURES curva sin columnas necesarias: {sorted(miss)}")
    # después de leer df
    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    if "ciclo_id" in df.columns:
        df["ciclo_id"] = df["ciclo_id"].astype(str)
    df["bloque_base"] = pd.to_numeric(df["bloque_base"], errors="coerce").astype("Int64")
    df["variedad_canon"] = df["variedad_canon"].astype(str).str.upper().str.strip()

    key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    if df.duplicated(subset=key).any():
        df = df.drop_duplicates(subset=key, keep="first")

    # asegurar columnas
    for c in NUM_COLS:
        if c not in df.columns:
            df[c] = np.nan
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"

    X = df[NUM_COLS + CAT_COLS]
    pred = model.predict(X)

    out = df[["fecha", "bloque_base", "variedad_canon"]].copy()
    if "ciclo_id" in df.columns:
        out["ciclo_id"] = df["ciclo_id"]

    out["mll_version"] = ver_dir.name
    out["factor_curva_mll_raw"] = pd.to_numeric(pred, errors="coerce")

    # safety clip (evita volar planificación)
    out["factor_curva_mll"] = out["factor_curva_mll_raw"].clip(lower=0.2, upper=5.0)

    # fallback si quedó NaN por cualquier razón
    out["factor_curva_mll"] = out["factor_curva_mll"].fillna(1.0)

```

```

out["created_at"] = pd.Timestamp.utcnow()

cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "factor_curva_mll", "factor_curva_mll_raw",
"mll_version", "created_at"]
cols = [c for c in cols if c in out.columns]
out = out[cols].sort_values(["bloque_base", "variedad_canon", "fecha"]).reset_index(drop=True)

write_parquet(out, OUT_PATH)
print(f"OK -> {OUT_PATH} | rows={len(out):,} | version={ver_dir.name}")
print(f"best_model: {meta.get('best_model')}")

if __name__ == "__main__":
    main(version=None)

-----
[82/164] FILE: \src\models\mll\apply_desp_ajuste_poscosecha_mll.py
-----

from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

from common.io import read_parquet, write_parquet

IN_UNIVERSE = Path("data/gold/pred_poscosecha_mll_hidr_grado_dia_bloque_destino.parquet")

REG_DESP = Path("models_registry/mll/desp_poscosecha")
REG_AJ = Path("models_registry/mll/ajuste_poscosecha")

OUT_PATH = Path("data/gold/pred_poscosecha_mll_full_grado_dia_bloque_destino.parquet")

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino"]

def _latest_version_dir(root: Path) -> Path:
    if not root.exists():
        raise FileNotFoundError(f"No existe {root}")
    dirs = [p for p in root.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {root}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def _load_meta_and_model(root: Path, version: str | None):
    ver_dir = _latest_version_dir(root) if version is None else (root / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe versión: {ver_dir}")
    with open(ver_dir / "metrics.json", "r", encoding="utf-8") as f:
        meta = json.load(f)
    model = load(ver_dir / "model.joblib")
    return ver_dir.name, meta, model

def main(version_desp: str | None = None, version_aj: str | None = None) -> None:
    v_desp, meta_desp, model_desp = _load_meta_and_model(REG_DESP, version_desp)
    v_aj, meta_aj, model_aj = _load_meta_and_model(REG_AJ, version_aj)

    df = read_parquet(IN_UNIVERSE).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"Universe sin columnas: {sorted(miss)}")

```

```

# dh (nombre esperado)
dh_col = None
for c in ["dh_dias_mll", "dh_dias_pred_mll", "dh_dias_pred", "dh_dias"]:
    if c in df.columns:
        dh_col = c
        break
if dh_col is None:
    raise ValueError("No encontré columna dh (esperaba dh_dias_mll o similar) en el universo.")

df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
df["destino"] = df["destino"].astype(str).str.upper().str.strip()
df[dh_col] = pd.to_numeric(df[dh_col], errors="coerce").astype("Int64")

# fecha_post predicha
df["fecha_post_pred_mll"] = df["fecha"] + pd.to_timedelta(df[dh_col].fillna(0).astype(int), unit="D")

# features calendario por fecha_post
df["dow"] = df["fecha_post_pred_mll"].dt.dayofweek.astype("Int64")
df["month"] = df["fecha_post_pred_mll"].dt.month.astype("Int64")
df["weekofyear"] = df["fecha_post_pred_mll"].dt.isocalendar().week.astype("Int64")

for c in NUM_COLS:
    if c not in df.columns:
        df[c] = np.nan
for c in CAT_COLS:
    if c not in df.columns:
        df[c] = "UNKNOWN"

X = df[NUM_COLS + CAT_COLS]

# DESP
lo_d, hi_d = meta_desp.get("clip_range_apply", [0.05, 1.00])
df["factor_desp_mll_raw"] = pd.to_numeric(pd.Series(model_desp.predict(X)), errors="coerce")
df["factor_desp_mll"] = df["factor_desp_mll_raw"].clip(lower=float(lo_d), upper=float(hi_d))

# AJUSTE
lo_a, hi_a = meta_aj.get("clip_range_apply", [0.80, 1.50])
df["ajuste_mll_raw"] = pd.to_numeric(pd.Series(model_aj.predict(X)), errors="coerce")
df["ajuste_mll"] = df["ajuste_mll_raw"].clip(lower=float(lo_a), upper=float(hi_a))

df["mll_desp_version"] = v_desp
df["mll_ajuste_version"] = v_aj
df["created_at"] = pd.Timestamp.utcnow()

# ---- cálculo final de factor macro poscosecha
# cajas_post = cajas_split * factor_hidr * factor_desp / ajuste
cajas_col = None

# PRIORIDAD: usar SIEMPRE la caja ya splitteada por destino
for c in [
    "cajas_split_grado_dia",
    "cajas_split",
    "cajas_seed_split",
    "cajas_destino_grado_dia",
    "cajas_iniciales",
    "cajas_mll_grado_dia",
    "cajas_mll_grado_dia_in",
]:
    if c in df.columns:
        cajas_col = c
        break

if cajas_col is None:
    raise ValueError(
        "No encontré columna de cajas base para postcosecha. "
        "Se esperaba 'cajas_split_grado_dia' (preferida) u otra equivalente."
    )

df["cajas_post_base_col"] = cajas_col

# (AUDIT opcional)
if ("cajas_split_grado_dia" in df.columns) and ("cajas_mll_grado_dia" in df.columns):

```

```

    _diff = (
        pd.to_numeric(df["cajas_split_grado_dia"], errors="coerce").fillna(0.0)
        - pd.to_numeric(df["cajas_mll_grado_dia"], errors="coerce").fillna(0.0)
    ).abs()
    print(f"[AUDIT] max_abs(split - cajas_mll_grado_dia)={float(_diff.max()):.12f} mean_abs={float(_diff.mean()):.6f}")

    df["cajas_postcosecha_mll"] = (
        pd.to_numeric(df[cajas_col], errors="coerce").fillna(0.0)
        * pd.to_numeric(df.get("factor_hidr_mll"), errors="coerce").fillna(1.0)
        * pd.to_numeric(df.get("factor_desp_mll"), errors="coerce").fillna(1.0)
        / pd.to_numeric(df.get("ajuste_mll"), errors="coerce").replace(0, np.nan).fillna(1.0)
    )

    write_parquet(df, OUT_PATH)
    print(f"OK -> {OUT_PATH} | rows={len(df):,} | desp={v_desp} | ajuste={v_aj} | cajas_base={cajas_col}")

if __name__ == "__main__":
    main(version_desp=None, version_aj=None)

-----
[83/164] FILE: \src\models\mll\apply_dh_poscosecha_mll.py
-----
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

from common.io import read_parquet, write_parquet

IN_UNIVERSE = Path("data/gold/pred_poscosecha_seed_grado_dia_bloque_destino.parquet")
REGISTRY_ROOT = Path("models_registry/mll/dh_poscosecha")
OUT_PATH = Path("data/gold/pred_poscosecha_mll_dh_grado_dia_bloque_destino.parquet")

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino", "grado"]

def _latest_version_dir() -> Path:
    if not REGISTRY_ROOT.exists():
        raise FileNotFoundError(f"No existe {REGISTRY_ROOT}")
    dirs = [p for p in REGISTRY_ROOT.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {REGISTRY_ROOT}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def main(version: str | None = None) -> None:
    if version is None:
        ver_dir = _latest_version_dir()
    else:
        ver_dir = REGISTRY_ROOT / version
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe la versión: {ver_dir}")

    metrics_path = ver_dir / "metrics.json"
    with open(metrics_path, "r", encoding="utf-8") as f:
        meta = json.load(f)

    model_path = ver_dir / "model.joblib"
    if not model_path.exists():
        raise FileNotFoundError(f"No existe: {model_path}")

    df = read_parquet(IN_UNIVERSE).copy()
    df.columns = [str(c).strip() for c in df.columns]

```



```

need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino"}
miss = need - set(df.columns)
if miss:
    raise ValueError(f"Universe seed sin columnas: {sorted(miss)}")

df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
df["grado"] = pd.to_numeric(df["grado"], errors="coerce").astype("Int64")
df["destino"] = df["destino"].astype(str).str.upper().str.strip()

# Features calendario sobre fecha (cosecha proyectada)
df["dow"] = df["fecha"].dt.dayofweek.astype("Int64")
df["month"] = df["fecha"].dt.month.astype("Int64")
df["weekofyear"] = df["fecha"].dt.isocalendar().week.astype("Int64")

# asegurar cols
for c in NUM_COLS:
    if c not in df.columns:
        df[c] = np.nan
for c in CAT_COLS:
    if c not in df.columns:
        df[c] = "UNKNOWN"

model = load(model_path)

X = df[NUM_COLS + CAT_COLS]
pred = model.predict(X)

# clip + redondeo a int días
dh = pd.to_numeric(pd.Series(pred), errors="coerce").clip(lower=0, upper=30)
df["dh_dias_mll_raw"] = dh
df["dh_dias_mll"] = np rint(df["dh_dias_mll_raw"]).astype("Int64")

# fecha_post_pred_mll
df["fecha_post_pred_mll"] = df["fecha"] + pd.to_timedelta(df["dh_dias_mll"].fillna(0).astype(int), unit="D")

df["mll_dh_version"] = ver_dir.name
df["created_at"] = pd.Timestamp.utcnow()

# Mantener todo y agregar columnas MLl
keep_extra = ["dh_dias_mll_raw", "dh_dias_mll", "fecha_post_pred_mll", "mll_dh_version", "created_at"]
for c in keep_extra:
    if c not in df.columns:
        df[c] = pd.NA

write_parquet(df, OUT_PATH)
print(f"OK -> {OUT_PATH} | rows={len(df):,} | version={ver_dir.name} | clip={meta.get('clip_range_apply')}")

```

```

if __name__ == "__main__":
    main(version=None)

```

```

[84/164] FILE: \src\models\mll\apply_dist_grado.py

```

```

from __future__ import annotations

```

```

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

```

```

from common.io import read_parquet, write_parquet

```

```

FEATURES_PATH = Path("data/features/features_cosecha_bloque_fecha.parquet")
REGISTRY_ROOT = Path("models_registry/mll/dist_grado")
OUT_PATH = Path("data/gold/pred_dist_grado_mll.parquet")

```

```

# Debe calzar con train_dist_grado.py
NUM_COLS = [
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
    "share_grado_baseline",
    # features ?futuro? útiles (si existen)
    "day_in_harvest",
    "rel_pos",
    "n_harvest_days",
    "n_dias_cosecha",
]

CAT_COLS = [
    "variedad_canon",
    "tipo_sp",
    "area",
]

def _latest_version_dir() -> Path:
    if not REGISTRY_ROOT.exists():
        raise FileNotFoundError(f"No existe {REGISTRY_ROOT}")
    dirs = [p for p in REGISTRY_ROOT.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {REGISTRY_ROOT}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def _ensure_cols(df: pd.DataFrame) -> pd.DataFrame:
    out = df.copy()

    # defaults numéricas
    for c in NUM_COLS:
        if c not in out.columns:
            out[c] = np.nan

    # defaults categóricas
    for c in CAT_COLS:
        if c not in out.columns:
            out[c] = "UNKNOWN"

    return out

def _renormalize(out: pd.DataFrame) -> pd.DataFrame:
    """
    Asegura share>=0 y sum=1 por (ciclo_id,bloque_base,variedad_canon,fecha).
    """
    out["share_grado_mll_raw"] = pd.to_numeric(out["share_grado_mll_raw"], errors="coerce")
    out["share_grado_mll_raw"] = out["share_grado_mll_raw"].fillna(0.0).clip(lower=0.0)

    grp = ["ciclo_id", "bloque_base", "variedad_canon", "fecha"]
    s = out.groupby(grp, dropna=False)["share_grado_mll_raw"].transform("sum")

    # si suma=0, fallback a baseline y renormaliza baseline
    if (s == 0).any():
        out["share_grado_mll_raw"] = np.where(
            s.to_numpy() > 0,

```

```

        out["share_grado_mll_raw"].to_numpy(),
        pd.to_numeric(out["share_grado_baseline"], errors="coerce").fillna(0.0).to_numpy(),
    )
    s = out.groupby(grp, dropna=False)["share_grado_mll_raw"].transform("sum")

    out["share_grado_mll"] = np.where(s > 0, out["share_grado_mll_raw"] / s, np.nan)
    return out

def main(version: str | None = None) -> None:
    ver_dir = _latest_version_dir() if version is None else (REGISTRY_ROOT / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe la versión: {ver_dir}")

    metrics_path = ver_dir / "metrics.json"
    if not metrics_path.exists():
        raise FileNotFoundError(f"No encontré metrics.json en {ver_dir}")

    with open(metrics_path, "r", encoding="utf-8") as f:
        meta = json.load(f)

    grades = meta.get("grades", [])
    if not grades:
        raise ValueError("metrics.json no trae 'grades'")

    df = read_parquet(FEATURES_PATH).copy()
    df = _ensure_cols(df)

    need = {"ciclo_id", "fecha", "bloque_base", "grado", "share_grado_baseline", "variedad_canon"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"FEATURES sin columnas necesarias: {sorted(miss)}")

    # Tipos sanos
    df["ciclo_id"] = df["ciclo_id"].astype(str)
    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    df["bloque_base"] = pd.to_numeric(df["bloque_base"], errors="coerce").astype("Int64")
    df["grado"] = pd.to_numeric(df["grado"], errors="coerce").astype("Int64")
    df["variedad_canon"] = df["variedad_canon"].astype(str).str.upper().str.strip()

    # base_all asegura TODOS los grados en salida (incluye futuro)
    base_all = df[["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_baseline"]].copy()

    preds_parts: list[pd.DataFrame] = []
    missing_models: list[int] = []

    for g in grades:
        g_int = int(g)
        model_path = ver_dir / f"model_grade_{g_int}.joblib"
        if not model_path.exists():
            missing_models.append(g_int)
            continue

        model = load(model_path)

        sub = df[df["grado"] == g_int].copy()
        if sub.empty:
            continue

        X = sub[NUM_COLS + CAT_COLS]
        pred = model.predict(X)

        sub_out = sub[["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_baseline"]].copy()
        sub_out["mll_version"] = ver_dir.name
        sub_out["share_grado_mll_raw"] = pd.to_numeric(pred, errors="coerce")

        preds_parts.append(sub_out)

    pred_only = pd.concat(preds_parts, ignore_index=True) if preds_parts else pd.DataFrame(
        columns=["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_baseline", "mll_v

```

```

ersion", "share_grado_mll_raw"]
)

# Merge pred con base
out = base_all.merge(
    pred_only[["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado", "share_grado_mll_raw", "mll_
version"]],
    on=["ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado"],
    how="left",
)

out["mll_version"] = out["mll_version"].fillna(ver_dir.name)

# fallback: donde no hay pred (no modelo o NaN), usa baseline
out["share_grado_mll_raw"] = out["share_grado_mll_raw"].where(
    out["share_grado_mll_raw"].notna(),
    out["share_grado_baseline"],
)

# renormalizar válido
out = _renormalize(out)

out["created_at"] = pd.Timestamp.utcnow()

out = out[
    [
        "ciclo_id",
        "fecha",
        "bloque_base",
        "variedad_canon",
        "grado",
        "share_grado_baseline",
        "share_grado_mll",
        "mll_version",
        "created_at",
    ]
].sort_values(["ciclo_id", "bloque_base", "variedad_canon", "fecha", "grado"]).reset_index(drop=True)

write_parquet(out, OUT_PATH)

fmin = pd.to_datetime(out["fecha"].min()).date() if len(out) else None
fmax = pd.to_datetime(out["fecha"].max()).date() if len(out) else None
fut_rate = float((out["fecha"] > pd.Timestamp.today().normalize()).mean()) if len(out) else 0.0
print(f"OK -> {OUT_PATH} | rows={len(out):,} | version={ver_dir.name} | fecha_min={fmin} fecha_max={fmax}")
)

print(f"[CHECK] % filas con fecha > hoy: {fut_rate:.4f}")
if missing_models:
    missing_models = sorted(set(missing_models))
    print(f"[WARN] grados sin modelo en {ver_dir.name}: {missing_models[:30]}{'...' if len(missing_models)
>30 else ''}")

# sanity: sum=1
grp = ["ciclo_id", "bloque_base", "variedad_canon", "fecha"]
s = out.groupby(grp, dropna=False)["share_grado_mll"].sum()
if len(s):
    print(f"[CHECK] share_grado_mll sum min/max: {float(s.min()):.6f} / {float(s.max()):.6f}")

if __name__ == "__main__":
    main(version=None)

-----
[85/164] FILE: \src\models\mll\apply_harvest_window_mll.py
-----
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

```

```

from common.io import read_parquet, write_parquet

FEATURES_PATH = Path("data/features/features_harvest_window_ml1.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/harvest_window")

MEDIANS_PATH = Path("data/silver/dim_mediana_etapas_tipo_sp_variedad_area.parquet")

OUT_PATH = Path("data/gold/pred_harvest_window_ml1.parquet")

# -----
# Helpers
# -----
def _latest_version_dir() -> Path:
    if not REGISTRY_ROOT.exists():
        raise FileNotFoundError(f"No existe {REGISTRY_ROOT}")
    dirs = [p for p in REGISTRY_ROOT.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {REGISTRY_ROOT}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _pick_first(df: pd.DataFrame, candidates: list[str]) -> str | None:
    for c in candidates:
        if c in df.columns:
            return c
    return None

# -----
# Main
# -----
def main(version: str | None = None) -> None:
    ver_dir = _latest_version_dir() if version is None else (REGISTRY_ROOT / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe la versión: {ver_dir}")

    metrics_path = ver_dir / "metrics.json"
    if not metrics_path.exists():
        raise FileNotFoundError(f"No encontré metrics.json en {ver_dir}")

    with open(metrics_path, "r", encoding="utf-8") as f:
        meta = json.load(f)

    model_start_path = ver_dir / "model_start_offset.joblib"
    model_days_path = ver_dir / "model_harvest_days.joblib"
    if not model_start_path.exists():
        raise FileNotFoundError(f"No encontré: {model_start_path}")
    if not model_days_path.exists():
        raise FileNotFoundError(f"No encontré: {model_days_path}")

    model_start = load(model_start_path)
    model_days = load(model_days_path)

    # -----
    # Features base (ciclos a predecir)
    # -----
    df = read_parquet(FEATURES_PATH).copy()

```

```

df.columns = [str(c).strip() for c in df.columns]

need = {"ciclo_id", "bloque_base", "fecha_sp", "area", "tipo_sp"}
miss = need - set(df.columns)
if miss:
    raise ValueError(f"features_harvest_window_mll: faltan columnas {sorted(miss)}")

# variedad_canon: si no existe, fallback
if "variedad_canon" not in df.columns:
    vcol = _pick_first(df, ["variedad_std", "variedad", "variedad_raw", "variedad_id"])
    if vcol is None:
        df["variedad_canon"] = "UNKNOWN"
    else:
        df["variedad_canon"] = df[vcol]

df["ciclo_id"] = df["ciclo_id"].astype(str)
df["fecha_sp"] = _to_date(df["fecha_sp"])
df["bloque_base"] = _canon_int(df["bloque_base"])
df["variedad_canon"] = _canon_str(df["variedad_canon"])
df["area"] = _canon_str(df["area"])
df["tipo_sp"] = _canon_str(df["tipo_sp"])

# -----
# Medianas (fallback): tu esquema real
# -----
med = read_parquet(MEDIANS_PATH).copy()
med.columns = [str(c).strip() for c in med.columns]

# columnas obligatorias del archivo real
need_med = {"tipo_sp", "variedad_std", "area", "mediana_dias_veg", "mediana_dias_harvest"}
miss_med = need_med - set(med.columns)
if miss_med:
    raise ValueError(
        "dim_mediana_etapas_tipo_sp_variedad_area: faltan columnas esperadas "
        f"{sorted(miss_med)}. Cols={list(med.columns)}"
    )

med["area"] = _canon_str(med["area"])
med["tipo_sp"] = _canon_str(med["tipo_sp"])
med["variedad_canon"] = _canon_str(med["variedad_std"])

# map a nombres estándar que usa el apply
med["d_start_med"] = pd.to_numeric(med["mediana_dias_veg"], errors="coerce")
med["n_days_med"] = pd.to_numeric(med["mediana_dias_harvest"], errors="coerce")

med = med[["area", "variedad_canon", "tipo_sp", "d_start_med", "n_days_med"]].drop_duplicates()

# -----
# Predicción MLL
# -----
num_cols = meta.get("num_cols", [])
cat_cols = meta.get("cat_cols", [])

# asegurar columnas faltantes
for c in num_cols:
    if c not in df.columns:
        df[c] = np.nan
for c in cat_cols:
    if c not in df.columns:
        df[c] = "UNKNOWN"

X = df[num_cols + cat_cols]
d_start_pred = pd.to_numeric(model_start.predict(X), errors="coerce")
n_days_pred = pd.to_numeric(model_days.predict(X), errors="coerce")

# clips duros
d_start_pred = np.clip(d_start_pred, 0, 180)
n_days_pred = np.clip(n_days_pred, 1, 180)

out = df[["ciclo_id", "bloque_base", "variedad_canon", "area", "tipo_sp", "fecha_sp"]].copy()
out["mll_version"] = ver_dir.name
out["d_start_pred_mll"] = d_start_pred

```

```

out["n_harvest_days_pred_m11"] = n_days_pred

# -----
# Fallback a mediana por segmento
# -----
out = out.merge(med, on=["area", "variedad_canon", "tipo_sp"], how="left")

out["d_start_pred_final"] = out["d_start_pred_m11"]
out["n_harvest_days_pred_final"] = out["n_harvest_days_pred_m11"]

miss_start = out["d_start_pred_final"].isna()
miss_days = out["n_harvest_days_pred_final"].isna()

out.loc[miss_start, "d_start_pred_final"] = out.loc[miss_start, "d_start_med"]
out.loc[miss_days, "n_harvest_days_pred_final"] = out.loc[miss_days, "n_days_med"]

# fallback global final
out["d_start_pred_final"] = out["d_start_pred_final"].fillna(90.0).clip(0, 180)
out["n_harvest_days_pred_final"] = out["n_harvest_days_pred_final"].fillna(40.0).clip(1, 180)

out["harvest_start_pred"] = out["fecha_sp"] + pd.to_timedelta(out["d_start_pred_final"], unit="D")
out["harvest_end_pred"] = out["harvest_start_pred"] + pd.to_timedelta(out["n_harvest_days_pred_final"] - 1
, unit="D")

out["start_source"] = np.where(miss_start, "median_segment", "m11_model")
out["days_source"] = np.where(miss_days, "median_segment", "m11_model")

out["created_at"] = pd.Timestamp.utcnow()

out = out[
    [
        "ciclo_id",
        "bloque_base",
        "variedad_canon",
        "area",
        "tipo_sp",
        "fecha_sp",
        "harvest_start_pred",
        "harvest_end_pred",
        "d_start_pred_final",
        "n_harvest_days_pred_final",
        "start_source",
        "days_source",
        "m11_version",
        "created_at",
    ]
].sort_values(["bloque_base", "variedad_canon", "fecha_sp"]).reset_index(drop=True)

write_parquet(out, OUT_PATH)
print(f"OK -> {OUT_PATH} | rows={len(out):,} | version={ver_dir.name}")
print(f"[COVERAGE] start_source=m11_model: {float((out['start_source']=='m11_model').mean()):.2%}")
print(f"[COVERAGE] days_source=m11_model: {float((out['days_source']=='m11_model').mean()):.2%}")

fmin = pd.to_datetime(out["harvest_start_pred"].min()).date() if len(out) else None
fmax = pd.to_datetime(out["harvest_end_pred"].max()).date() if len(out) else None
print(f"[RANGE] harvest_start_pred..harvest_end_pred: {fmin} .. {fmax}")

if __name__ == "__main__":
    main(version=None)

```

```

-----
[86/164] FILE: \src\models\m11\apply_hidr_poscosecha_m11.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

```

```

from common.io import read_parquet, write_parquet

IN_UNIVERSE = Path("data/gold/pred_poscosecha_mll_dh_grado_dia_bloque_destino.parquet")
REGISTRY_ROOT = Path("models_registry/mll/hidr_poscosecha")
OUT_PATH = Path("data/gold/pred_poscosecha_mll_hidr_grado_dia_bloque_destino.parquet")

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino", "grado"]

def _latest_version_dir() -> Path:
    if not REGISTRY_ROOT.exists():
        raise FileNotFoundError(f"No existe {REGISTRY_ROOT}")
    dirs = [p for p in REGISTRY_ROOT.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {REGISTRY_ROOT}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def main(version: str | None = None) -> None:
    if version is None:
        ver_dir = _latest_version_dir()
    else:
        ver_dir = REGISTRY_ROOT / version
        if not ver_dir.exists():
            raise FileNotFoundError(f"No existe la versión: {ver_dir}")

    metrics_path = ver_dir / "metrics.json"
    with open(metrics_path, "r", encoding="utf-8") as f:
        meta = json.load(f)

    model_path = ver_dir / "model.joblib"
    if not model_path.exists():
        raise FileNotFoundError(f"No existe: {model_path}")

    df = read_parquet(IN_UNIVERSE).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"Universe sin columnas: {sorted(miss)}")

    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    df["grado"] = pd.to_numeric(df["grado"], errors="coerce").astype("Int64")
    df["destino"] = df["destino"].astype(str).str.upper().str.strip()

    df["dow"] = df["fecha"].dt.dayofweek.astype("Int64")
    df["month"] = df["fecha"].dt.month.astype("Int64")
    df["weekofyear"] = df["fecha"].dt.isocalendar().week.astype("Int64")

    for c in NUM_COLS:
        if c not in df.columns:
            df[c] = np.nan
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"

    model = load(model_path)
    X = df[NUM_COLS + CAT_COLS]
    pred = model.predict(X)

    lo, hi = meta.get("clip_range_apply", [0.80, 3.00])
    df["factor_hidr_mll_raw"] = pd.to_numeric(pd.Series(pred), errors="coerce")
    df["factor_hidr_mll"] = df["factor_hidr_mll_raw"].clip(lower=float(lo), upper=float(hi))

    df["mll_hidr_version"] = ver_dir.name
    df["created_at"] = pd.Timestamp.utcnow()

    write_parquet(df, OUT_PATH)

```



```

print(f"OK -> {OUT_PATH} | rows={len(df):,} | version={ver_dir.name} | clip={lo}->{hi}")

if __name__ == "__main__":
    main(version=None)

-----
[87/164] FILE: \src\models\ml1\apply_peso_tallo_grado.py
-----
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import load

from common.io import read_parquet, write_parquet

FEATURES_PATH = Path("data/features/features_peso_tallo_grado_bloque_dia.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/peso_tallo_grado")
OUT_PATH = Path("data/gold/pred_peso_tallo_grado_ml1.parquet")

NUM_COLS = [
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
    "peso_tallo_baseline_g",
]
CAT_COLS = ["variedad_canon", "tipo_sp", "area"]

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _latest_version_dir() -> Path:
    if not REGISTRY_ROOT.exists():
        raise FileNotFoundError(f"No existe {REGISTRY_ROOT}")
    dirs = [p for p in REGISTRY_ROOT.iterdir() if p.is_dir()]
    if not dirs:
        raise FileNotFoundError(f"No hay versiones dentro de {REGISTRY_ROOT}")
    return sorted(dirs, key=lambda p: p.name)[-1]

def main(version: str | None = None) -> None:
    ver_dir = _latest_version_dir() if version is None else (REGISTRY_ROOT / version)
    if not ver_dir.exists():
        raise FileNotFoundError(f"No existe la versión: {ver_dir}")

```

```

metrics_path = ver_dir / "metrics.json"
with open(metrics_path, "r", encoding="utf-8") as f:
    meta = json.load(f)

df = read_parquet(FEATURES_PATH).copy()
df.columns = [str(c).strip() for c in df.columns]

need = {"fecha", "bloque_base", "variedad_canon", "grado", "peso_tallo_baseline_g"}
miss = need - set(df.columns)
if miss:
    raise ValueError(f"FEATURES sin columnas necesarias: {sorted(miss)}")

# Canonización llaves para merges aguas abajo
df["fecha"] = _to_date(df["fecha"])
df["bloque_base"] = _canon_int(df["bloque_base"])
df["grado"] = _canon_int(df["grado"])
df["variedad_canon"] = _canon_str(df["variedad_canon"])

# Asegurar columnas
for c in NUM_COLS:
    if c not in df.columns:
        df[c] = np.nan
for c in CAT_COLS:
    if c not in df.columns:
        df[c] = "UNKNOWN"

# IMPORTANTE: predecir todos los grados presentes (no solo los del meta)
grados_features = sorted([int(x) for x in df["grado"].dropna().unique().tolist()])
grados_meta = meta.get("grades", [])
grados_meta = [int(x) for x in grados_meta] if grados_meta else []
grades_all = sorted(set(grados_features) | set(grados_meta))

preds = []

for g in grades_all:
    sub = df[df["grado"] == g].copy()
    if sub.empty:
        continue

    model_path = ver_dir / f"model_grade_{g}.joblib"
    if model_path.exists():
        model = load(model_path)
        X = sub[NUM_COLS + CAT_COLS]
        sub["factor_peso_tallo_mll_raw"] = model.predict(X)
        sub["peso_model_fallback_const1"] = 0
    else:
        sub["factor_peso_tallo_mll_raw"] = 1.0
        sub["peso_model_fallback_const1"] = 1

    preds.append(
        sub[
            [
                "fecha",
                "bloque_base",
                "variedad_canon",
                "grado",
                "peso_tallo_baseline_g",
                "factor_peso_tallo_mll_raw",
                "peso_model_fallback_const1",
            ]
        ]
    )

out = pd.concat(preds, ignore_index=True) if preds else pd.DataFrame(
    columns=[
        "fecha",
        "bloque_base",
        "variedad_canon",
        "grado",
        "peso_tallo_baseline_g",
        "factor_peso_tallo_mll_raw",
    ]
)

```

```

        "peso_model_fallback_const1",
    ]
)

# Clip coherente con train
out["factor_peso_tallo_mll"] = (
    pd.to_numeric(out["factor_peso_tallo_mll_raw"], errors="coerce")
    .clip(lower=0.60, upper=1.60)
)

base = pd.to_numeric(out["peso_tallo_baseline_g"], errors="coerce")
out["peso_tallo_mll_g"] = np.where(
    base.fillna(0) > 0,
    base * out["factor_peso_tallo_mll"],
    np.nan,
)

out["mll_version"] = ver_dir.name
out["created_at"] = pd.Timestamp.utcnow()

# Unicidad defensiva por llaves (por si features trae duplicados)
keys = ["fecha", "bloque_base", "variedad_canon", "grado"]
if out.duplicated(subset=keys).any():
    out = (
        out.groupby(keys, dropna=False, as_index=False)
        .agg(
            peso_tallo_baseline_g=("peso_tallo_baseline_g", "mean"),
            factor_peso_tallo_mll_raw=("factor_peso_tallo_mll_raw", "mean"),
            factor_peso_tallo_mll=("factor_peso_tallo_mll", "mean"),
            peso_tallo_mll_g=("peso_tallo_mll_g", "mean"),
            peso_model_fallback_const1=("peso_model_fallback_const1", "max"),
            mll_version=("mll_version", "first"),
            created_at=("created_at", "first"),
        )
    )

out = out[
    [
        "fecha",
        "bloque_base",
        "variedad_canon",
        "grado",
        "peso_tallo_baseline_g",
        "factor_peso_tallo_mll_raw",
        "factor_peso_tallo_mll",
        "peso_tallo_mll_g",
        "peso_model_fallback_const1",
        "mll_version",
        "created_at",
    ]
].sort_values(["bloque_base", "variedad_canon", "fecha", "grado"]).reset_index(drop=True)

write_parquet(out, OUT_PATH)
print(f"OK -> {OUT_PATH} | rows={len(out):,} | version={ver_dir.name}")

if __name__ == "__main__":
    main(version=None)

```

```

-----
[88/164] FILE: \src\models\mll\train_ajuste_poscosecha_mll.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import json
import warnings

```

```

import numpy as np
import pandas as pd

```

```

from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

DIM_PATH = Path("data/silver/dim_mermas_ajuste_fecha_post_destino.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/ajuste_poscosecha")

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino"]

def _make_pipeline(model) -> Pipeline:
    num_pipe = Pipeline(steps=[("imputer", SimpleImputer(strategy="median"))])
    cat_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="most_frequent")),
            ("onehot", OneHotEncoder(handle_unknown="ignore")),
        ]
    )
    pre = ColumnTransformer(
        transformers=[("num", num_pipe, NUM_COLS), ("cat", cat_pipe, CAT_COLS)],
        remainder="drop",
        sparse_threshold=0.0,
    )
    return Pipeline(steps=[("pre", pre), ("model", model)])

def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0, random_state=0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=400,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }

def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize().dropna()
    if d.empty:
        return []
    uniq = np.array(sorted(d.unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []
    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]
    folds = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        vs, ve = vb.min(), vb.max()
        tr = (all_dates < vs).to_numpy()
        va = ((all_dates >= vs) & (all_dates <= ve)).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:

```

```

        folds.append((np.where(tr)[0], np.where(va)[0]))
if not folds:
    cut = uniq[int(len(uniq) * 0.8)]
    tr = (all_dates < cut).to_numpy()
    va = (all_dates >= cut).to_numpy()
    if tr.sum() > 0 and va.sum() > 0:
        folds = [(np.where(tr)[0], np.where(va)[0])]
return folds

def main() -> None:
    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    dim = read_parquet(DIM_PATH).copy()
    dim.columns = [str(c).strip() for c in dim.columns]

    need = {"fecha_post", "destino", "ajuste"}
    miss = need - set(dim.columns)
    if miss:
        raise ValueError(f"dim_mermas_ajuste_fecha_post_destino sin columnas: {sorted(miss)}")

    df = dim.copy()
    df["fecha_post"] = pd.to_datetime(df["fecha_post"], errors="coerce").dt.normalize()
    df["destino"] = df["destino"].astype(str).str.upper().str.strip()
    df["y"] = pd.to_numeric(df["ajuste"], errors="coerce")

    df = df[df["fecha_post"].notna() & df["destino"].notna() & df["y"].notna()].copy()
    df["y"] = df["y"].clip(lower=0.80, upper=1.50) # <- evita el ruido del 0.5 ?cap extremo?

    if len(df) < 100:
        raise ValueError(f"Poca data para entrenar ajuste (n={len(df)}).")

    df["dow"] = df["fecha_post"].dt.dayofweek.astype("Int64")
    df["month"] = df["fecha_post"].dt.month.astype("Int64")
    df["weekofyear"] = df["fecha_post"].dt.isocalendar().week.astype("Int64")

    X = df[NUM_COLS + CAT_COLS].copy()
    y = df["y"].astype(float).to_numpy()

    folds = _time_folds(df["fecha_post"], n_folds=4)
    if not folds:
        raise ValueError("No pude armar folds temporales para ajuste.")

    models = _candidate_models()

    best_name, best_score, best_model = None, None, None
    all_metrics = {}

    for name, model in models.items():
        pipe = _make_pipeline(model)
        maes = []
        for tr_idx, va_idx in folds:
            pipe.fit(X.iloc[tr_idx], y[tr_idx])
            pred = pipe.predict(X.iloc[va_idx])
            maes.append(float(np.mean(np.abs(y[va_idx] - pred))))
        mae_mean = float(np.mean(maes))
        mae_std = float(np.std(maes))
        score = 0.85 * mae_mean + 0.15 * mae_std

        all_metrics[name] = {"mae_mean": mae_mean, "mae_std": mae_std, "score": float(score),
                             "n_rows": int(len(X)), "n_folds": int(len(folds))}
        if (best_score is None) or (score < best_score):
            best_score, best_name, best_model = score, name, model

    best_pipe = _make_pipeline(best_model)
    best_pipe.fit(X, y)

    model_path = out_dir / "model.joblib"
    dump(best_pipe, model_path)

```

```

summary = {
    "version": version,
    "created_at_utc": datetime.utcnow().isoformat(),
    "dim_path": str(DIM_PATH).replace("\\", "/"),
    "target": "ajuste",
    "clip_range_apply": [0.80, 1.50],
    "features": {"num": NUM_COLS, "cat": CAT_COLS},
    "best_model": best_name,
    "metrics": all_metrics,
    "model_path": str(model_path).replace("\\", "/"),
}

with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
    json.dump(summary, f, ensure_ascii=False, indent=2)

print(f"OK -> {out_dir}/ (model.joblib + metrics.json) | best={best_name} score={best_score:.6f}")

if __name__ == "__main__":
    main()

-----
[89/164] FILE: \src\models\ml1\train_curva_beta_multiplier_dia.py
-----

from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import dump
from sklearn.ensemble import HistGradientBoostingRegressor

from common.io import read_parquet

TRAINSET_PATH = Path("data/features/trainset_curva_beta_multiplier_dia.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/curva_beta_multiplier_dia")

NUM_COLS = [
    "day_in_harvest", "rel_pos", "n_harvest_days",
    "pct_avance_real", "dia_rel_cosecha_real", "gdc_acum_real",
    "rainfall_mm_dia", "horas_lluvia", "temp_avg_dia", "solar_energy_j_m2_dia",
    "wind_speed_avg_dia", "wind_run_dia", "gdc_dia",
    "dias_desde_sp", "gdc_acum_desde_sp",
    "dow", "month", "weekofyear",
]
CAT_COLS = ["variedad_canon", "area", "tipo_sp"]

def _make_version() -> str:
    return pd.Timestamp.utcnow().strftime("%Y%m%d_%H%M%S")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def main() -> None:
    if not TRAINSET_PATH.exists():
        raise FileNotFoundError(f"No existe: {TRAINSET_PATH}")

    created_at = pd.Timestamp.utcnow()
    version = _make_version()
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(TRAINSET_PATH).copy()

    # Ensure cols exist
    for c in NUM_COLS:
        if c not in df.columns:
            df[c] = np.nan
    for c in CAT_COLS:
        if c not in df.columns:

```

```

df[c] = "UNKNOWN"

for c in CAT_COLS:
    df[c] = _canon_str(df[c].fillna("UNKNOWN"))

y = pd.to_numeric(df["y_log_mult"], errors="coerce")
ok = y.notna()
df = df[ok].copy()
y = y[ok].astype(float)

# weights: dar más peso a días con mayor real (para que aprenda supresión por clima también se puede)
if "tallos_real_dia" in df.columns:
    w = pd.to_numeric(df["tallos_real_dia"], errors="coerce").fillna(0.0).to_numpy(dtype=float)
    w = np.sqrt(np.clip(w, 0.0, None))
    w = np.where(w > 0, w / np.median(w[w > 0]), 1.0)
    w = np.clip(w, 0.2, 5.0)
else:
    w = np.ones(len(df), dtype=float)

# X
X = df[NUM_COLS + CAT_COLS].copy()
for c in NUM_COLS:
    X[c] = pd.to_numeric(X[c], errors="coerce")
X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)

model = HistGradientBoostingRegressor(
    loss="squared_error",
    max_depth=6,
    learning_rate=0.06,
    max_leaf_nodes=31,
    min_samples_leaf=120,      # más alto para suavidad
    l2_regularization=1e-4,
    random_state=42,
)
model.fit(X, y.to_numpy(dtype=float), sample_weight=w)

dump(model, out_dir / "model_log_mult.joblib")

meta = {
    "created_at": str(created_at),
    "version": version,
    "target": "y_log_mult = log(share_real/(share_beta+eps)) (daily multiplier on beta prior)",
    "feature_cols_numeric": NUM_COLS,
    "feature_cols_categorical": CAT_COLS,
    "feature_names": list(X.columns),
    "n_train_rows": int(len(X)),
    "clip_note": "apply will clip multiplier exp(y) to stable bounds",
}
with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
    json.dump(meta, f, indent=2, ensure_ascii=False)

print(f"OK -> {out_dir} | n_train_rows={len(X):,}")

if __name__ == "__main__":
    main()

-----
[90/164] FILE: \src\models\ml1\train_curva_beta_params.py
-----
from __future__ import annotations

from pathlib import Path
import json
import numpy as np
import pandas as pd
from joblib import dump
from sklearn.ensemble import HistGradientBoostingRegressor

from common.io import read_parquet

TRAINSET_PATH = Path("data/features/trainset_curva_beta_params.parquet")

```

```

REGISTRY_ROOT = Path("models_registry/ml1/curva_beta_params")

# Columnas categóricas
CAT_COLS = ["variedad_canon", "area", "tipo_sp"]

# Targets (alpha, beta) siempre > 1
# Modelamos  $z = \log(\text{target} - 1)$  para asegurar positividad al aplicar:  $\text{target} = 1 + \exp(z)$ 
TARGET_ALPHA = "alpha"
TARGET_BETA = "beta"

def _make_version() -> str:
    return pd.Timestamp.utcnow().strftime("%Y%m%d_%H%M%S")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def main() -> None:
    if not TRAINSET_PATH.exists():
        raise FileNotFoundError(f"No existe: {TRAINSET_PATH}")

    created_at = pd.Timestamp.utcnow()
    version = _make_version()
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(TRAINSET_PATH).copy()

    # Canon cats
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"
        df[c] = _canon_str(df[c].fillna("UNKNOWN"))

    # targets
    df[TARGET_ALPHA] = pd.to_numeric(df[TARGET_ALPHA], errors="coerce")
    df[TARGET_BETA] = pd.to_numeric(df[TARGET_BETA], errors="coerce")

    ok = df[TARGET_ALPHA].notna() & df[TARGET_BETA].notna()
    df = df[ok].copy()
    if len(df) < 50:
        raise ValueError(f"Trainset muy pequeño ({len(df)}). Revisa build_targets o MIN_REAL_TOTAL.")

    # Features numéricas: todo excepto ids/cats/targets
    drop_cols = {"ciclo_id", "bloque_base", "created_at", TARGET_ALPHA, TARGET_BETA}
    num_cols = [c for c in df.columns if c not in drop_cols and c not in CAT_COLS]
    # limpia
    for c in num_cols:
        df[c] = pd.to_numeric(df[c], errors="coerce")

    X = df[num_cols + CAT_COLS].copy()
    X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)

    # Targets transformados
    y_a = np.log(np.clip(df[TARGET_ALPHA].to_numpy(dtype=float) - 1.0, 1e-6, None))
    y_b = np.log(np.clip(df[TARGET_BETA].to_numpy(dtype=float) - 1.0, 1e-6, None))

    # sample_weight: más peso a ciclos con mayor real_total si existe
    if "real_total" in df.columns:
        w = pd.to_numeric(df["real_total"], errors="coerce").fillna(0.0).to_numpy(dtype=float)
        w = np.sqrt(np.clip(w, 0.0, None))
        w = np.where(w > 0, w / np.median(w[w > 0]), 1.0)
        w = np.clip(w, 0.2, 5.0)
    else:
        w = np.ones(len(df), dtype=float)

    # Modelos
    model_a = HistGradientBoostingRegressor(
        loss="squared_error",
        max_depth=6,
        learning_rate=0.06,
        max_leaf_nodes=31,
        min_samples_leaf=60,

```



```

        l2_regularization=1e-4,
        random_state=42,
    )
    model_b = HistGradientBoostingRegressor(
        loss="squared_error",
        max_depth=6,
        learning_rate=0.06,
        max_leaf_nodes=31,
        min_samples_leaf=60,
        l2_regularization=1e-4,
        random_state=42,
    )

    model_a.fit(X, y_a, sample_weight=w)
    model_b.fit(X, y_b, sample_weight=w)

    dump(model_a, out_dir / "model_alpha.joblib")
    dump(model_b, out_dir / "model_beta.joblib")

    meta = {
        "created_at": str(created_at),
        "version": version,
        "target": "alpha,beta params of Beta PDF on rel_pos; trained on real harvest shares (cycle-level)",
        "transform": "z = log(param - 1), param = 1 + exp(z) (ensures >1 unimodal)",
        "feature_cols_numeric": num_cols,
        "feature_cols_categorical": CAT_COLS,
        "feature_names": list(X.columns),
        "n_train_rows": int(len(X)),
        "n_groups": int(df[["ciclo_id", "bloque_base", "variedad_canon"]].drop_duplicates().shape[0]),
    }
    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(meta, f, indent=2, ensure_ascii=False)

    print(f"OK -> {out_dir} | n_train_rows={len(X):,} | n_groups={meta['n_groups']:,}")

if __name__ == "__main__":
    main()

-----
[91/164] FILE: \src\models\ml1\train_curva_cdf_dia.py
-----

from __future__ import annotations

from pathlib import Path
import json

import numpy as np
import pandas as pd
from joblib import dump
from sklearn.ensemble import HistGradientBoostingRegressor

from common.io import read_parquet

FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_ml1.parquet")
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")
DIM_VAR_PATH = Path("data/silver/dim_variedad_canon.parquet")

REGISTRY_ROOT = Path("models_registry/ml1/curva_cdf_dia")

# Inputs numéricos (si faltan, se crean como NaN)
NUM_COLS = [
    "day_in_harvest",
    "rel_pos",
    "n_harvest_days",
    "pct_avance_real",          # opcional
    "dia_rel_cosecha_real",    # opcional
    "gdc_acum_real",          # opcional
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",

```

```

"solar_energy_j_m2_dia",
"wind_speed_avg_dia",
"wind_run_dia",
"gdc_dia",
"dias_desde_sp",
"gdc_acum_desde_sp",
"dow",
"month",
"weekofyear",
]

CAT_COLS = ["variedad_canon", "area", "tipo_sp"]
CAT_COLS_MERGE = ["area", "tipo_sp"]

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

def _make_version() -> str:
    # reemplazo de utcnow (deprecated)
    return pd.Timestamp.now("UTC").strftime("%Y%m%d_%H%M%S")

def _coalesce_cols(df: pd.DataFrame, out_col: str, candidates: list[str]) -> None:
    if out_col in df.columns:
        base = df[out_col]
    else:
        base = pd.Series([pd.NA] * len(df), index=df.index)
    for c in candidates:
        if c in df.columns:
            base = base.where(base.notna(), df[c])
    df[out_col] = base

def _dedupe_columns(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.Index(df.columns.astype(str))
    if cols.is_unique:
        return df

    out = df.copy()
    seen: dict[str, list[int]] = {}
    for i, c in enumerate(out.columns.astype(str)):
        seen.setdefault(c, []).append(i)

    keep_series: dict[str, pd.Series] = {}
    for c, idxs in seen.items():
        if len(idxs) == 1:
            keep_series[c] = out.iloc[:, idxs[0]]
        else:
            s = out.iloc[:, idxs[0]]
            for j in idxs[1:]:
                s2 = out.iloc[:, j]
                s = s.where(s.notna(), s2)
            keep_series[c] = s

    ordered: list[str] = []
    for c in out.columns.astype(str):

```

```

        if c not in ordered:
            ordered.append(c)

    return pd.DataFrame({c: keep_series[c] for c in ordered})

def _load_var_map(dim_var: pd.DataFrame) -> dict[str, str]:
    _require(dim_var, ["variedad_raw", "variedad_canon"], "dim_variedad_canon")
    dv = dim_var.copy()
    dv["variedad_raw"] = _canon_str(dv["variedad_raw"])
    dv["variedad_canon"] = _canon_str(dv["variedad_canon"])
    dv = dv.dropna(subset=["variedad_raw", "variedad_canon"]).drop_duplicates(subset=["variedad_raw"])
    return dict(zip(dv["variedad_raw"], dv["variedad_canon"]))

def main() -> None:
    for p in [FEATURES_PATH, UNIVERSE_PATH, PROG_PATH, DIM_VAR_PATH]:
        if not p.exists():
            raise FileNotFoundError(f"No existe: {p}")

    created_at = pd.Timestamp.now("UTC")
    version = _make_version()
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    # ---- var map (PROG raw -> canon)
    dim_var = read_parquet(DIM_VAR_PATH).copy()
    var_map = _load_var_map(dim_var)

    # ---- features (baseline + clima + term + pred window)
    feat = _dedupe_columns(read_parquet(FEATURES_PATH).copy())
    if not pd.Index(feat.columns.astype(str)).is_unique:
        dup = pd.Index(feat.columns.astype(str))[pd.Index(feat.columns.astype(str)).duplicated()].unique().tolist()
        raise ValueError(f"features_curva aún tiene columnas duplicadas: {dup}")

    _require(
        feat,
        ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia", "tallos_proy"],
        "features_curva",
    )

    feat["ciclo_id"] = feat["ciclo_id"].astype(str)
    feat["fecha"] = _to_date(feat["fecha"])
    feat["bloque_base"] = _canon_int(feat["bloque_base"])
    feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
    for c in ["area", "tipo_sp"]:
        if c in feat.columns:
            feat[c] = _canon_str(feat[c])

    # aliases desde *_pred
    _coalesce_cols(feat, "day_in_harvest", ["day_in_harvest", "day_in_harvest_pred", "day_in_harvest_pred_final"])
    _coalesce_cols(feat, "rel_pos", ["rel_pos", "rel_pos_pred", "rel_pos_pred_final"])
    _coalesce_cols(feat, "n_harvest_days", ["n_harvest_days", "n_harvest_days_pred", "n_harvest_days_pred_final"])

    for c in ["day_in_harvest", "rel_pos", "n_harvest_days"]:
        feat[c] = pd.to_numeric(feat[c], errors="coerce")

    # asegurar columnas
    for c in NUM_COLS:
        if c not in feat.columns:
            feat[c] = np.nan
    for c in CAT_COLS:
        if c not in feat.columns:
            feat[c] = "UNKNOWN"

    feat["tallos_pred_baseline_dia"] = pd.to_numeric(feat["tallos_pred_baseline_dia"], errors="coerce").fillna(0.0)
    feat["tallos_proy"] = pd.to_numeric(feat["tallos_proy"], errors="coerce").fillna(0.0)

```

```

# ---- universe
uni = read_parquet(UNIVERSE_PATH).copy()
_require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon"], "universe_harvest_grid_m11")
uni["ciclo_id"] = uni["ciclo_id"].astype(str)
uni["fecha"] = _to_date(uni["fecha"])
uni["bloque_base"] = _canon_int(uni["bloque_base"])
uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
uni_k = uni[key].drop_duplicates()

# ---- progreso real (panelizado + zeros)
prog = read_parquet(PROG_PATH).copy()
_require(prog, ["ciclo_id", "fecha", "bloque_base", "variedad", "tallos_real_dia"], "dim_cosecha_progress_
bloque_fecha")
prog["ciclo_id"] = prog["ciclo_id"].astype(str)
prog["fecha"] = _to_date(prog["fecha"])
prog["bloque_base"] = _canon_int(prog["bloque_base"])
prog["variedad_raw"] = _canon_str(prog["variedad"])
prog["variedad_canon"] = prog["variedad_raw"].map(var_map).fillna(prog["variedad_raw"])
prog["variedad_canon"] = _canon_str(prog["variedad_canon"])
prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce").fillna(0.0)

prog_k_cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_real_dia"]
if "pct_avance_real" in prog.columns:
    prog_k_cols.append("pct_avance_real")
prog_k = prog[prog_k_cols].drop_duplicates(subset=key)

feat_take = key + ["tallos_pred_baseline_dia", "tallos_proy"] + NUM_COLS + CAT_COLS_MERGE
feat_take = list(dict.fromkeys(feat_take))

panel = (
    uni_k
    .merge(feat[feat_take], on=key, how="left")
    .merge(prog_k, on=key, how="left")
)

# fill 0 en días no registrados
panel["tallos_real_dia"] = pd.to_numeric(panel["tallos_real_dia"], errors="coerce").fillna(0.0)

# asegurar features
for c in NUM_COLS:
    if c not in panel.columns:
        panel[c] = np.nan
for c in CAT_COLS:
    if c not in panel.columns:
        panel[c] = "UNKNOWN"

panel["variedad_canon"] = _canon_str(panel["variedad_canon"])
if "area" in panel.columns:
    panel["area"] = _canon_str(panel["area"].fillna("UNKNOWN"))
if "tipo_sp" in panel.columns:
    panel["tipo_sp"] = _canon_str(panel["tipo_sp"].fillna("UNKNOWN"))

# ---- train cycles con señal real
cyc_sum_real = panel.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform("sum").astype(float)
train = panel[cyc_sum_real > 0].copy()

# ---- máscara harvest (si no hay day_in_harvest, igual entrena con rel_pos; pero preferimos day_in_harves
t)
dih = pd.to_numeric(train["day_in_harvest"], errors="coerce")
nh = pd.to_numeric(train["n_harvest_days"], errors="coerce")
is_h = dih.notna() & nh.notna() & (dih >= 1) & (nh >= 1) & (dih <= nh)
train.loc[~is_h, "tallos_real_dia"] = 0.0

# ---- target: CDF real por ciclo (ordenado por day_in_harvest; fallback a fecha si dih falta)
train["_dih_sort"] = pd.to_numeric(train["day_in_harvest"], errors="coerce")

# FIX PANDAS: Series.view ya no existe -> usar astype("int64") en datetime64[ns]
# (fecha es datetime64[ns] por _to_date)
fecha_int64 = pd.to_datetime(train["fecha"], errors="coerce").astype("int64")

```

```

train["_sort_key"] = np.where(
    train["_dih_sort"].notna().to_numpy(),
    train["_dih_sort"].astype("float64").to_numpy(),
    fecha_int64.astype("float64").to_numpy(),
)

train = train.sort_values(["ciclo_id", "_sort_key"], kind="mergesort")

tot = train.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform("sum").astype(float)
cum = train.groupby("ciclo_id", dropna=False)["tallos_real_dia"].cumsum().astype(float)
train["cdf_real"] = np.where(tot > 0, cum / tot, np.nan)

# ---- sample_weight: si existe pct_avance_real, ramp 2%..5%; si no, usar cdf_real como proxy
if "pct_avance_real" in train.columns and train["pct_avance_real"].notna().any():
    pav = pd.to_numeric(train["pct_avance_real"], errors="coerce").fillna(0.0).astype(float)
    sw = np.clip((pav - 0.02) / 0.03, 0.2, 1.0)
else:
    cdfp = pd.to_numeric(train["cdf_real"], errors="coerce").fillna(0.0).astype(float)
    sw = np.clip((cdfp - 0.02) / 0.03, 0.2, 1.0)
train["sample_weight"] = sw

# ---- X/y
y = pd.to_numeric(train["cdf_real"], errors="coerce")
X = train[NUM_COLS + CAT_COLS].copy()

ok = y.notna()
X = X.loc[ok].copy()
y = y.loc[ok].astype(float)
sample_weight = train.loc[ok, "sample_weight"].astype(float).to_numpy()

# one-hot + persist feature_names
X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)

model = HistGradientBoostingRegressor(
    loss="squared_error",
    max_depth=6,
    learning_rate=0.08,
    max_leaf_nodes=31,
    min_samples_leaf=80,
    l2_regularization=1e-4,
    random_state=42,
)
model.fit(X, y, sample_weight=sample_weight)

dump(model, out_dir / "model_curva_cdf_dia.joblib")

meta = {
    "created_at": str(created_at),
    "version": version,
    "best_model": "HistGradientBoostingRegressor",
    "target": "cdf_real per-cycle (panelized on universe; missing prog days filled with 0; monotonic enforced at apply)",
    "position_cols": "coalesce day_in_harvest_pred/rel_pos_pred/n_harvest_days_pred -> day_in_harvest/rel_pos/n_harvest_days",
    "gating": "sample_weight ramps between 2%..5% using pct_avance_real if exists else cdf_real proxy",
    "feature_cols_numeric": NUM_COLS,
    "feature_cols_categorical": CAT_COLS,
    "feature_names": list(X.columns),
    "n_train_rows": int(len(X)),
    "n_cycles_train": int(train.loc[ok, "ciclo_id"].nunique()),
}
with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
    json.dump(meta, f, indent=2, ensure_ascii=False)

print(f"OK -> {out_dir} | n_train_rows={len(X):,} | n_cycles={meta['n_cycles_train']: ,}")

if __name__ == "__main__":
    main()

```

[92/164] FILE: \src\models\m11\train_curva_share_dia.py

```
-----  
from __future__ import annotations
```

```
from pathlib import Path  
import json  
import numpy as np  
import pandas as pd  
from joblib import dump  
from sklearn.ensemble import HistGradientBoostingRegressor
```

```
from common.io import read_parquet
```

```
FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")  
UNIVERSE_PATH = Path("data/gold/universe_harvest_grid_m11.parquet")  
PROG_PATH = Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet")  
DIM_VAR_PATH = Path("data/silver/dim_variedad_canon.parquet")
```

```
REGISTRY_ROOT = Path("models_registry/m11/curva_share_dia")
```

```
# -----
```

```
# Model columns
```

```
# -----
```

```
NUM_COLS = [  
    "day_in_harvest",  
    "rel_pos",  
    "n_harvest_days",  
    "pct_avance_real",          # <- opcional en data, pero el modelo la puede usar si existe  
    "dia_rel_cosecha_real",  
    "gdc_acum_real",  
    "rainfall_mm_dia",  
    "horas_lluvia",  
    "temp_avg_dia",  
    "solar_energy_j_m2_dia",  
    "wind_speed_avg_dia",  
    "wind_run_dia",  
    "gdc_dia",  
    "dias_desde_sp",  
    "gdc_acum_desde_sp",  
    "dow",  
    "month",  
    "weekofyear",  
]  
CAT_COLS = ["variedad_canon", "area", "tipo_sp"]
```

```
# -----
```

```
# Helpers
```

```
# -----
```

```
def _to_date(s: pd.Series) -> pd.Series:  
    return pd.to_datetime(s, errors="coerce").dt.normalize()
```

```
def _canon_int(s: pd.Series) -> pd.Series:  
    return pd.to_numeric(s, errors="coerce").astype("Int64")
```

```
def _canon_str(s: pd.Series) -> pd.Series:  
    return s.astype(str).str.upper().str.strip()
```

```
def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:  
    miss = [c for c in cols if c not in df.columns]  
    if miss:  
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")
```

```
def _make_version() -> str:  
    return pd.Timestamp.utcnow().strftime("%Y%m%d_%H%M%S")
```

```

def _dedupe_columns(df: pd.DataFrame) -> pd.DataFrame:
    cols = pd.Index(df.columns.astype(str))
    if cols.is_unique:
        return df

    out = df.copy()
    seen: dict[str, list[int]] = {}
    for i, c in enumerate(out.columns.astype(str)):
        seen.setdefault(c, []).append(i)

    keep: dict[str, pd.Series] = {}
    for c, idxs in seen.items():
        s = out.iloc[:, idxs[0]]
        for j in idxs[1:]:
            s2 = out.iloc[:, j]
            s = s.where(s.notna(), s2)
        keep[c] = s

    ordered: list[str] = []
    for c in out.columns.astype(str):
        if c not in ordered:
            ordered.append(c)

    return pd.DataFrame({c: keep[c] for c in ordered})

def _load_var_map(dim_var: pd.DataFrame) -> dict[str, str]:
    _require(dim_var, ["variedad_raw", "variedad_canon"], "dim_variedad_canon")
    dv = dim_var.copy()
    dv["variedad_raw"] = _canon_str(dv["variedad_raw"])
    dv["variedad_canon"] = _canon_str(dv["variedad_canon"])
    dv = dv.dropna(subset=["variedad_raw", "variedad_canon"]).drop_duplicates(subset=["variedad_raw"])
    return dict(zip(dv["variedad_raw"], dv["variedad_canon"]))

def _relpos_bin(rel_pos: pd.Series) -> pd.Series:
    x = pd.to_numeric(rel_pos, errors="coerce").astype(float)
    bins = [-np.inf, 0.05, 0.15, 0.30, 0.60, 0.85, np.inf]
    labels = ["00-05", "05-15", "15-30", "30-60", "60-85", "85-100"]
    return pd.cut(x, bins=bins, labels=labels)

def _ndays_bucket(n: pd.Series) -> pd.Series:
    x = pd.to_numeric(n, errors="coerce")
    return pd.cut(
        x.astype(float),
        bins=[-np.inf, 30, 45, 60, np.inf],
        labels=["<=30", "31-45", "46-60", ">60"],
    )

def _safe_quantile(s: pd.Series, q: float) -> float:
    v = pd.to_numeric(s, errors="coerce").dropna().astype(float)
    if len(v) < 30:
        return float("nan")
    return float(v.quantile(q))

def main() -> None:
    for p in [FEATURES_PATH, UNIVERSE_PATH, PROG_PATH, DIM_VAR_PATH]:
        if not p.exists():
            raise FileNotFoundError(f"No existe: {p}")

    created_at = pd.Timestamp.utcnow()
    version = _make_version()
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    # -----
    # Dim variedad canon
    # -----

```

```

dim_var = read_parquet(DIM_VAR_PATH).copy()
var_map = _load_var_map(dim_var)

# -----
# Features
# -----
feat = read_parquet(FEATURES_PATH).copy()
feat = _dedupe_columns(feat)

_require(
    feat,
    ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia", "tallos_proy"],
    "features_curva",
)

feat["ciclo_id"] = feat["ciclo_id"].astype(str)
feat["fecha"] = _to_date(feat["fecha"])
feat["bloque_base"] = _canon_int(feat["bloque_base"])
feat["variedad_canon"] = _canon_str(feat["variedad_canon"])
for c in ["area", "tipo_sp"]:
    if c in feat.columns:
        feat[c] = _canon_str(feat[c])

# asegurar columnas del modelo (num+cat) en FEAT
for c in NUM_COLS:
    if c not in feat.columns:
        feat[c] = np.nan
for c in CAT_COLS:
    if c not in feat.columns:
        feat[c] = "UNKNOWN"

feat["tallos_pred_baseline_dia"] = pd.to_numeric(feat["tallos_pred_baseline_dia"], errors="coerce").fillna(0.0)
feat["tallos_proy"] = pd.to_numeric(feat["tallos_proy"], errors="coerce").fillna(0.0)

# -----
# Universe (positional)
# -----
uni = read_parquet(UNIVERSE_PATH).copy()
_require(uni, ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "rel_pos_pred", "n_harvest_days_pred"], "universe_harvest_grid_mll")

uni["ciclo_id"] = uni["ciclo_id"].astype(str)
uni["fecha"] = _to_date(uni["fecha"])
uni["bloque_base"] = _canon_int(uni["bloque_base"])
uni["variedad_canon"] = _canon_str(uni["variedad_canon"])

key = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
uni_k = uni[key + ["rel_pos_pred", "day_in_harvest_pred", "n_harvest_days_pred"]].drop_duplicates(subset=key)

# -----
# Prog real (solo real + opcional pct_avance_real)
# -----
prog = read_parquet(PROG_PATH).copy()
_require(prog, ["ciclo_id", "fecha", "bloque_base", "variedad", "tallos_real_dia"], "dim_cosecha_progress_bloque_fecha")

prog["ciclo_id"] = prog["ciclo_id"].astype(str)
prog["fecha"] = _to_date(prog["fecha"])
prog["bloque_base"] = _canon_int(prog["bloque_base"])

prog["variedad_raw"] = _canon_str(prog["variedad"])
prog["variedad_canon"] = prog["variedad_raw"].map(var_map).fillna(prog["variedad_raw"])

prog["tallos_real_dia"] = pd.to_numeric(prog["tallos_real_dia"], errors="coerce").fillna(0.0)

prog_take = ["ciclo_id", "fecha", "bloque_base", "variedad_canon", "tallos_real_dia"]
if "pct_avance_real" in prog.columns:
    prog_take.append("pct_avance_real")
prog_k = prog[prog_take].drop_duplicates(subset=key)

```



```

# -----
# Panel
# -----
# Importante: traer SOLO lo que necesitamos de feat (evita duplicados raros)
feat_take = key + ["tallos_pred_baseline_dia", "tallos_proy"] + NUM_COLS + ["area", "tipo_sp"]
feat_take = [c for c in dict.fromkeys(feat_take) if c in feat.columns]

panel = (
    uni_k.merge(feat[feat_take], on=key, how="left")
    .merge(prog_k, on=key, how="left")
)

# Asegurar TODAS las columnas del modelo en el panel (incl. pct_avance_real)
for c in NUM_COLS:
    if c not in panel.columns:
        panel[c] = np.nan
for c in CAT_COLS:
    if c not in panel.columns:
        panel[c] = "UNKNOWN"

# Missing prog day => 0 real
panel["tallos_real_dia"] = pd.to_numeric(panel["tallos_real_dia"], errors="coerce").fillna(0.0)

# Position: usar *_pred como eje de forma
panel["rel_pos"] = pd.to_numeric(panel["rel_pos"], errors="coerce")
panel["rel_pos"] = panel["rel_pos"].where(panel["rel_pos"].notna(), pd.to_numeric(panel["rel_pos_pred"], errors="coerce"))

panel["day_in_harvest"] = pd.to_numeric(panel["day_in_harvest"], errors="coerce")
panel["day_in_harvest"] = panel["day_in_harvest"].where(panel["day_in_harvest"].notna(), pd.to_numeric(panel["day_in_harvest_pred"], errors="coerce"))

panel["n_harvest_days"] = pd.to_numeric(panel["n_harvest_days"], errors="coerce")
panel["n_harvest_days"] = panel["n_harvest_days"].where(panel["n_harvest_days"].notna(), pd.to_numeric(panel["n_harvest_days_pred"], errors="coerce"))

# Canon categóricas
panel["variedad_canon"] = _canon_str(panel["variedad_canon"])
panel["area"] = _canon_str(panel.get("area", "UNKNOWN").fillna("UNKNOWN"))
panel["tipo_sp"] = _canon_str(panel.get("tipo_sp", "UNKNOWN").fillna("UNKNOWN"))

# -----
# Train set
# -----
cyc_sum = panel.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform("sum").astype(float)
train = panel[cyc_sum > 0].copy()

denom = train.groupby("ciclo_id", dropna=False)["tallos_real_dia"].transform("sum").astype(float)
train["share_real"] = np.where(denom > 0, train["tallos_real_dia"].astype(float) / denom, np.nan)

# sample_weight por avance si existe; si no, weight=1
pav = pd.to_numeric(train.get("pct_avance_real", np.nan), errors="coerce").fillna(0.0).astype(float)
sw = np.ones(len(train), dtype=float)
if "pct_avance_real" in train.columns:
    sw = np.clip((pav - 0.02) / 0.03, 0.2, 1.0)
train["sample_weight"] = sw

# X/y
X = train[NUM_COLS + CAT_COLS].copy()
y = pd.to_numeric(train["share_real"], errors="coerce")
ok = y.notna()

X = X.loc[ok].copy()
y = y.loc[ok].astype(float)
sample_weight = train.loc[ok, "sample_weight"].astype(float).to_numpy()

# one-hot + guardar feature names
X = pd.get_dummies(X, columns=CAT_COLS, dummy_na=True)

model = HistGradientBoostingRegressor(
    loss="squared_error",
    max_depth=6,

```

```

        learning_rate=0.08,
        max_leaf_nodes=31,
        min_samples_leaf=80,
        l2_regularization=1e-4,
        random_state=42,
    )
model.fit(X, y, sample_weight=sample_weight)
dump(model, out_dir / "model_curva_share_dia.joblib")

# -----
# Calibration caps/floors (learned)
# -----
cal = train[["ciclo_id", "variedad_canon", "area", "tipo_sp", "rel_pos", "n_harvest_days", "share_real"]].
copy()
cal["rel_pos_bin"] = _relpos_bin(cal["rel_pos"])
cal["n_days_bucket"] = _ndays_bucket(cal["n_harvest_days"])

seg_cols = ["variedad_canon", "area", "tipo_sp", "n_days_bucket", "rel_pos_bin"]
g = cal.groupby(seg_cols, dropna=False)["share_real"]

cap = g.apply(lambda s: _safe_quantile(s, 0.99)).rename("cap_share_p99")
flo = g.apply(lambda s: _safe_quantile(s, 0.01)).rename("floor_share_p01")
cal_tab = pd.concat([cap, flo], axis=1).reset_index()

g2 = cal.groupby(["rel_pos_bin"], dropna=False)["share_real"]
cal_glob = pd.DataFrame({
    "rel_pos_bin": g2.apply(lambda s: _safe_quantile(s, 0.99)).index.astype(str),
    "cap_share_p99_global": g2.apply(lambda s: _safe_quantile(s, 0.99)).to_numpy(),
    "floor_share_p01_global": g2.apply(lambda s: _safe_quantile(s, 0.01)).to_numpy(),
})

cal_tab["created_at"] = created_at
cal_glob["created_at"] = created_at

cal_tab_path = out_dir / "cap_floor_share_by_relpos.parquet"
cal_glob_path = out_dir / "cap_floor_share_by_relpos_global.parquet"
cal_tab.to_parquet(cal_tab_path, index=False)
cal_glob.to_parquet(cal_glob_path, index=False)

meta = {
    "created_at": str(created_at),
    "version": version,
    "best_model": "HistGradientBoostingRegressor",
    "target": "share_real per-cycle (panelized on universe; missing prog days filled with 0)",
    "feature_cols_numeric": NUM_COLS,
    "feature_cols_categorical": CAT_COLS,
    "feature_names": list(X.columns),
    "n_train_rows": int(len(X)),
    "n_cycles_train": int(train["ciclo_id"].nunique()),
    "calibration": {
        "rel_pos_bins": ["00-05", "05-15", "15-30", "30-60", "60-85", "85-100"],
        "segmentation": seg_cols,
        "cap_quantile": 0.99,
        "floor_quantile": 0.01,
        "min_samples_quantile": 30,
    },
}

with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
    json.dump(meta, f, indent=2, ensure_ascii=False)

print(f"OK -> {out_dir} | n_train_rows={len(X):,} | n_cycles={meta['n_cycles_train']:,}")
print(f"OK -> {cal_tab_path.name} | rows={len(cal_tab):,}")
print(f"OK -> {cal_glob_path.name} | rows={len(cal_glob):,}")

if __name__ == "__main__":
    main()

```

```

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

FEATURES_PATH = Path("data/features/features_curva_cosecha_bloque_dia.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/curva_tallos_dia")

NUM_COLS = [
    # baseline
    "tallos_pred_baseline_dia",
    # progreso / etapa
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    # clima / gdc
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    # SP
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    # calendario
    "dow",
    "month",
    "weekofyear",
]

CAT_COLS = [
    "variedad_canon",
    "area",
    "tipo_sp",
]

def _wape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = np.sum(np.abs(y_true))
    if denom <= 1e-12:
        return float(np.nan)
    return float(np.sum(np.abs(y_true - y_pred)) / denom)

def _smape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = (np.abs(y_true) + np.abs(y_pred))
    denom = np.where(denom < 1e-12, 1e-12, denom)
    return float(np.mean(2.0 * np.abs(y_pred - y_true) / denom))

def _make_pipeline(model) -> Pipeline:

```

```

num_pipe = Pipeline(
    steps=[
        ("imputer", SimpleImputer(strategy="median")),
    ]
)
cat_pipe = Pipeline(
    steps=[
        ("imputer", SimpleImputer(strategy="most_frequent")),
        ("onehot", OneHotEncoder(handle_unknown="ignore")),
    ]
)
pre = ColumnTransformer(
    transformers=[
        ("num", num_pipe, NUM_COLS),
        ("cat", cat_pipe, CAT_COLS),
    ],
    remainder="drop",
)
return Pipeline(steps=[("pre", pre), ("model", model)])

```

```

def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0, random_state=0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=400,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }

```

```

def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    d = d.dropna()
    if d.empty:
        return []

    uniq = np.array(sorted(d.unique()))
    if len(uniq) < 10:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []

    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 3)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]

    folds = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()

    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        valid_start = vb.min()
        valid_end = vb.max()

        tr = (all_dates < valid_start).to_numpy()
        va = ((all_dates >= valid_start) & (all_dates <= valid_end)).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))

    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]

```

```

return folds

def _score_fold(y_true: np.ndarray, y_pred: np.ndarray) -> dict:
    mae = float(np.mean(np.abs(y_true - y_pred)))
    wape = _wape(y_true, y_pred)
    smape = _smape(y_true, y_pred)
    return {"mae": mae, "wape": wape, "smape": smape}

def main() -> None:
    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(FEATURES_PATH).copy()

    need = {"fecha", "bloque_base", "variedad_canon", "tallos_pred_baseline_dia", "factor_tallos_dia_clipped"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"features_curva_cosecha_bloque_dia sin columnas: {sorted(miss)}")

    # target
    df["y"] = pd.to_numeric(df["factor_tallos_dia_clipped"], errors="coerce")

    # training rows: donde hay y finito
    train = df[np.isfinite(df["y"].to_numpy())].copy()
    if train.empty:
        raise ValueError("No hay filas con target (factor_tallos_dia_clipped) finito. Revisa real coverage.")

    # asegurar columnas
    for c in NUM_COLS:
        if c not in train.columns:
            train[c] = np.nan
    for c in CAT_COLS:
        if c not in train.columns:
            train[c] = "UNKNOWN"

    # folds temporales
    folds = _time_folds(train["fecha"], n_folds=4)
    if not folds:
        raise ValueError("No pude construir splits temporales. Revisa rango de fechas en FEATURES.")

    X_all = train[NUM_COLS + CAT_COLS]
    y_all = train["y"].to_numpy(dtype=float)

    models = _candidate_models()

    summary: dict = {
        "version": version,
        "created_at_utc": datetime.utcnow().isoformat(),
        "features_path": str(FEATURES_PATH),
        "n_rows_train_total": int(len(train)),
        "n_folds": int(len(folds)),
        "best_model": None,
        "metrics": {},
        "model_path": None,
    }

    best_name = None
    best_score = None
    best_model = None

    for name, model in models.items():
        pipe = _make_pipeline(model)

        fold_stats = []
        for tr_idx, va_idx in folds:
            X_tr = X_all.iloc[tr_idx]
            y_tr = y_all[tr_idx]
            X_va = X_all.iloc[va_idx]

```

```

        y_va = y_all[va_idx]

        pipe.fit(X_tr, y_tr)
        pred = pipe.predict(X_va)

        fold_stats.append(_score_fold(y_va, pred))

    maes = np.array([m["mae"] for m in fold_stats], dtype=float)
    wapes = np.array([m["wape"] for m in fold_stats], dtype=float)

    mae_mean = float(np.nanmean(maes))
    mae_std = float(np.nanstd(maes))
    wape_mean = float(np.nanmean(wapes))

    # score compuesto (más bajo es mejor)
    score = 0.60 * mae_mean + 0.30 * (wape_mean if np.isfinite(wape_mean) else mae_mean) + 0.10 * mae_std

    summary["metrics"][name] = {
        "mae_mean": mae_mean,
        "mae_std": mae_std,
        "wape_mean": wape_mean,
        "score": float(score),
        "n_rows": int(len(train)),
        "n_folds": int(len(folds)),
    }

    if (best_score is None) or (score < best_score):
        best_score = score
        best_name = name
        best_model = model

assert best_name is not None and best_model is not None

# fit final
best_pipe = _make_pipeline(best_model)
best_pipe.fit(X_all, y_all)

model_path = out_dir / "model_curva_tallos_dia.joblib"
dump(best_pipe, model_path)

summary["best_model"] = best_name
summary["model_path"] = str(model_path).replace("\\", "/")

with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
    json.dump(summary, f, ensure_ascii=False, indent=2)

print(f"[ML1 curva_tallos_dia] best={best_name} score={best_score:.6f}")
print(f"OK -> {out_dir}/ (model + metrics.json)")

if __name__ == "__main__":
    main()

```

```

-----
[94/164] FILE: \src\models\ml1\train_desp_poscosecha_ml1.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import json
import warnings

```

```

import numpy as np
import pandas as pd
from joblib import dump

```

```

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer

```

```

from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

DIM_PATH = Path("data/silver/dim_mermas_ajuste_fecha_post_destino.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/desp_poscosecha")

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino"]

def _make_pipeline(model) -> Pipeline:
    num_pipe = Pipeline(steps=[("imputer", SimpleImputer(strategy="median"))])
    cat_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="most_frequent")),
            ("onehot", OneHotEncoder(handle_unknown="ignore")),
        ]
    )
    pre = ColumnTransformer(
        transformers=[("num", num_pipe, NUM_COLS), ("cat", cat_pipe, CAT_COLS)],
        remainder="drop",
        sparse_threshold=0.0, # dense para HGB
    )
    return Pipeline(steps=[("pre", pre), ("model", model)])

def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0, random_state=0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=400,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }

def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    d = d.dropna()
    if d.empty:
        return []
    uniq = np.array(sorted(d.unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []
    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]
    folds = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        vs, ve = vb.min(), vb.max()
        tr = (all_dates < vs).to_numpy()
        va = ((all_dates >= vs) & (all_dates <= ve)).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))
    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()

```

```

        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]
    return folds

def main() -> None:
    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    dim = read_parquet(DIM_PATH).copy()
    dim.columns = [str(c).strip() for c in dim.columns]

    need = {"fecha_post", "destino", "factor_desp"}
    miss = need - set(dim.columns)
    if miss:
        raise ValueError(f"dim_mermas_ajuste_fecha_post_destino sin columnas: {sorted(miss)}")

    df = dim.copy()
    df["fecha_post"] = pd.to_datetime(df["fecha_post"], errors="coerce").dt.normalize()
    df["destino"] = df["destino"].astype(str).str.upper().str.strip()
    df["y"] = pd.to_numeric(df["factor_desp"], errors="coerce")

    df = df[df["fecha_post"].notna() & df["destino"].notna() & df["y"].notna()].copy()
    df["y"] = df["y"].clip(lower=0.05, upper=1.00)

    if len(df) < 100:
        raise ValueError(f"Poca data para entrenar desp (n={len(df)}).")

    df["dow"] = df["fecha_post"].dt.dayofweek.astype("Int64")
    df["month"] = df["fecha_post"].dt.month.astype("Int64")
    df["weekofyear"] = df["fecha_post"].dt.isocalendar().week.astype("Int64")

    X = df[NUM_COLS + CAT_COLS].copy()
    y = df["y"].astype(float).to_numpy()

    folds = _time_folds(df["fecha_post"], n_folds=4)
    if not folds:
        raise ValueError("No pude armar folds temporales para desp.")

    models = _candidate_models()

    best_name, best_score, best_model = None, None, None
    all_metrics = {}

    for name, model in models.items():
        pipe = _make_pipeline(model)
        maes = []
        for tr_idx, va_idx in folds:
            pipe.fit(X.iloc[tr_idx], y[tr_idx])
            pred = pipe.predict(X.iloc[va_idx])
            maes.append(float(np.mean(np.abs(y[va_idx] - pred))))
        mae_mean = float(np.mean(maes))
        mae_std = float(np.std(maes))
        score = 0.85 * mae_mean + 0.15 * mae_std

        all_metrics[name] = {"mae_mean": mae_mean, "mae_std": mae_std, "score": float(score),
                             "n_rows": int(len(X)), "n_folds": int(len(folds))}
        if (best_score is None) or (score < best_score):
            best_score, best_name, best_model = score, name, model

    best_pipe = _make_pipeline(best_model)
    best_pipe.fit(X, y)

    model_path = out_dir / "model.joblib"
    dump(best_pipe, model_path)

    summary = {
        "version": version,
        "created_at_utc": datetime.utcnow().isoformat(),
        "dim_path": str(DIM_PATH).replace("\\", "/"),
        "target": "factor_desp",
    }

```



```

        "clip_range_apply": [0.05, 1.00],
        "features": {"num": NUM_COLS, "cat": CAT_COLS},
        "best_model": best_name,
        "metrics": all_metrics,
        "model_path": str(model_path).replace("\\", "/"),
    }

    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)

    print(f"OK -> {out_dir}/ (model.joblib + metrics.json) | best={best_name} score={best_score:.6f}")

if __name__ == "__main__":
    main()

-----
[95/164] FILE: \src\models\ml1\train_dh_poscosecha_ml1.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

FACT_PATH = Path("data/silver/fact_hidratacion_real_post_grado_destino.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/dh_poscosecha")

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino", "grado"] # OJO: aquí vamos a forzar ambos a string para evitar dtype mixto

# =====
# Utils
# =====
def _wape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = float(np.sum(np.abs(y_true)))
    if denom <= 1e-12:
        return float("nan")
    return float(np.sum(np.abs(y_true - y_pred)) / denom)

def _make_ohe() -> OneHotEncoder:
    # compat sklearn viejo/nuevo
    try:
        return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        return OneHotEncoder(handle_unknown="ignore", sparse=False)

def _make_pipeline(model) -> Pipeline:
    # num -> median
    num_pipe = Pipeline(steps=[("imputer", SimpleImputer(strategy="median"))])

    # cat -> constant (evita error "most_frequent ... could not convert string to float")

```

```

cat_pipe = Pipeline(
    steps=[
        ("imputer", SimpleImputer(strategy="constant", fill_value="UNKNOWN")),
        ("onehot", _make_ohe()),
    ]
)

pre = ColumnTransformer(
    transformers=[
        ("num", num_pipe, NUM_COLS),
        ("cat", cat_pipe, CAT_COLS),
    ],
    remainder="drop",
    sparse_threshold=0.0, # fuerza salida densa
)

return Pipeline(steps=[("pre", pre), ("model", model)])

def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0), # Ridge NO lleva random_state
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=400,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }

def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    d = d.dropna()
    if d.empty:
        return []

    uniq = np.array(sorted(d.unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []

    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]

    folds: list[tuple[np.ndarray, np.ndarray]] = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        valid_start = vb.min()
        valid_end = vb.max()

        tr = (all_dates < valid_start).to_numpy()
        va = ((all_dates >= valid_start) & (all_dates <= valid_end)).to_numpy()

        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))

    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]
    return folds

```

```

def _score_fold(y_true: np.ndarray, y_pred: np.ndarray) -> dict:
    mae = float(np.mean(np.abs(y_true - y_pred)))
    wape = _wape(y_true, y_pred)
    return {"mae": mae, "wape": wape}

# =====
# Main
# =====
def main() -> None:
    if not FACT_PATH.exists():
        raise FileNotFoundError(f"No existe: {FACT_PATH}")

    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    fact = read_parquet(FACT_PATH).copy()
    fact.columns = [str(c).strip() for c in fact.columns]

    need = {"fecha_cosecha", "fecha_post", "dh_dias", "grado", "destino"}
    miss = need - set(fact.columns)
    if miss:
        raise ValueError(f"fact_hidratacion_real_post_grado_destino sin columnas: {sorted(miss)}")

    df = fact.copy()

    # Canon básico
    df["fecha_cosecha"] = pd.to_datetime(df["fecha_cosecha"], errors="coerce").dt.normalize()
    df["dh_dias"] = pd.to_numeric(df["dh_dias"], errors="coerce")
    df["grado"] = pd.to_numeric(df["grado"], errors="coerce").astype("Int64")

    # destino: string robusto
    df["destino"] = df["destino"].astype(str).str.upper().str.strip()

    # filtros target
    df = df[df["fecha_cosecha"].notna()].copy()
    df = df[df["dh_dias"].notna()].copy()
    df = df[df["dh_dias"].between(0, 30)].copy()
    df = df[df["grado"].notna()].copy()

    if df.empty:
        raise ValueError("No hay datos válidos para entrenar DH.")

    # Features calendario (basadas en fecha_cosecha)
    df["dow"] = df["fecha_cosecha"].dt.dayofweek
    df["month"] = df["fecha_cosecha"].dt.month
    df["weekofyear"] = df["fecha_cosecha"].dt.isocalendar().week.astype(int)

    # Fuerza numérico limpio (por si vienen objetos raros)
    for c in NUM_COLS:
        df[c] = pd.to_numeric(df[c], errors="coerce")

    # >>> FIX CLAVE: fuerza cat a string (evita dtype mixto que dispara el error)
    df["grado"] = df["grado"].astype("Int64").astype(str).replace("<NA>", "UNKNOWN")
    df["destino"] = df["destino"].fillna("UNKNOWN").astype(str)

    X = df[NUM_COLS + CAT_COLS].copy()
    y = df["dh_dias"].astype(float).to_numpy()

    folds = _time_folds(df["fecha_cosecha"], n_folds=4)
    if not folds:
        raise ValueError("No pude armar folds temporales para DH.")

    models = _candidate_models()

    best_name: str | None = None
    best_score: float | None = None
    best_model = None
    all_metrics: dict[str, dict] = {}

    for name, model in models.items():

```

```

pipe = _make_pipeline(model)
fold_stats = []

for tr_idx, va_idx in folds:
    X_tr = X.iloc[tr_idx]
    y_tr = y[tr_idx]
    X_va = X.iloc[va_idx]
    y_va = y[va_idx]

    pipe.fit(X_tr, y_tr)
    pred = pipe.predict(X_va)

    fold_stats.append(_score_fold(y_va, pred))

maes = np.array([m["mae"] for m in fold_stats], dtype=float)
wapes = np.array([m["wape"] for m in fold_stats], dtype=float)

mae_mean = float(np.nanmean(maes))
mae_std = float(np.nanstd(maes))
wape_mean = float(np.nanmean(wapes))

# score: prioriza MAE + WAPE, con penalización por varianza
score = 0.75 * mae_mean + 0.15 * (wape_mean if np.isfinite(wape_mean) else mae_mean) + 0.10 * mae_std

all_metrics[name] = {
    "mae_mean": mae_mean,
    "mae_std": mae_std,
    "wape_mean": wape_mean,
    "score": float(score),
    "n_rows": int(len(X)),
    "n_folds": int(len(folds)),
}

if (best_score is None) or (score < best_score):
    best_score = score
    best_name = name
    best_model = model

assert best_name is not None and best_model is not None and best_score is not None

best_pipe = _make_pipeline(best_model)
best_pipe.fit(X, y)

model_path = out_dir / "model.joblib"
dump(best_pipe, model_path)

summary = {
    "version": version,
    "created_at_utc": datetime.utcnow().isoformat(),
    "fact_path": str(FACT_PATH).replace("\\", "/"),
    "target": "dh_dias",
    "clip_range_apply": [0, 30],
    "features": {"num": NUM_COLS, "cat": CAT_COLS},
    "best_model": best_name,
    "metrics": all_metrics,
    "model_path": str(model_path).replace("\\", "/"),
}

with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
    json.dump(summary, f, ensure_ascii=False, indent=2)

print(f"OK -> {out_dir} / (model.joblib + metrics.json) | best={best_name} score={best_score:.6f}")

if __name__ == "__main__":
    main()

```

```

-----
[96/164] FILE: \src\models\ml1\train_dist_grado.py
-----
from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

# -----
# Config
# -----
FEATURES_PATH = Path("data/features/features_cosecha_bloque_fecha.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/dist_grado")

# numéricas (si faltan, se crean con NaN)
NUM_COLS = [
    # etapa/progreso
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    # clima
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    # estado térmico SP
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    # calendario
    "dow",
    "month",
    "weekofyear",
    # baseline
    "share_grado_baseline",
]

# categóricas (si faltan, UNKNOWN)
CAT_COLS = [
    "variedad_canon", # <-- clave
    "tipo_sp",
    "area",
]

# -----
# Metrics
# -----
def _wape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = np.sum(np.abs(y_true))
    if denom <= 1e-12:
        return float(np.nan)
    return float(np.sum(np.abs(y_true - y_pred)) / denom)

def _smape(y_true: np.ndarray, y_pred: np.ndarray) -> float:

```

```

denom = (np.abs(y_true) + np.abs(y_pred))
denom = np.where(denom < 1e-12, 1e-12, denom)
return float(np.mean(2.0 * np.abs(y_pred - y_true) / denom))

def _score_fold(y_true: np.ndarray, y_pred: np.ndarray) -> dict:
    mae = float(np.mean(np.abs(y_true - y_pred)))
    wape = _wape(y_true, y_pred)
    smape = _smape(y_true, y_pred)
    return {"mae": mae, "wape": wape, "smape": smape}

# -----
# Pipeline / models
# -----
def _make_pipeline(model) -> Pipeline:
    num_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="median")),
        ]
    )
    cat_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="most_frequent")),
            ("onehot", OneHotEncoder(handle_unknown="ignore")),
        ]
    )

    pre = ColumnTransformer(
        transformers=[
            ("num", num_pipe, NUM_COLS),
            ("cat", cat_pipe, CAT_COLS),
        ],
        remainder="drop",
    )
    return Pipeline(steps=[("pre", pre), ("model", model)])

def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0, random_state=0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=300,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }

# -----
# Time folds
# -----
def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    """
    Forward-chaining por fechas únicas.
    Fallback a holdout 80/20 si hay poca data.
    """
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    d = d.dropna()
    if d.empty:
        return []

    uniq = np.array(sorted(d.unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []

```

```

n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
valid_blocks = np.array_split(uniq, n_folds + 1)[1:]

folds = []
all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
for vb in valid_blocks:
    if len(vb) == 0:
        continue
    valid_start = vb.min()
    valid_end = vb.max()

    tr = (all_dates < valid_start).to_numpy()
    va = ((all_dates >= valid_start) & (all_dates <= valid_end)).to_numpy()

    if tr.sum() > 0 and va.sum() > 0:
        folds.append((np.where(tr)[0], np.where(va)[0]))

if not folds:
    cut = uniq[int(len(uniq) * 0.8)]
    tr = (all_dates < cut).to_numpy()
    va = (all_dates >= cut).to_numpy()
    if tr.sum() > 0 and va.sum() > 0:
        folds = [(np.where(tr)[0], np.where(va)[0])]

return folds

# -----
# Main
# -----
def main() -> None:
    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(FEATURES_PATH)

    # Compat: si viene "variedad" y no "variedad_canon", lo usamos.
    if "variedad_canon" not in df.columns and "variedad" in df.columns:
        df["variedad_canon"] = df["variedad"]

    # Validaciones mínimas
    need = {"fecha", "bloque_base", "grado", "share_grado_real", "share_grado_baseline"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"features_cosecha_bloque_fecha.parquet sin columnas: {sorted(miss)}")

    # Training dataset: solo donde hay target real
    train_df = df[df["share_grado_real"].notna()].copy()

    # (Opcional) entrenar solo en ventana real si existe
    if "en_ventana_cosecha_real" in train_df.columns:
        train_df = train_df[train_df["en_ventana_cosecha_real"] == 1].copy()

    # Asegurar columnas
    for c in NUM_COLS:
        if c not in train_df.columns:
            train_df[c] = np.nan
    for c in CAT_COLS:
        if c not in train_df.columns:
            train_df[c] = "UNKNOWN"

    # Tipos
    train_df["grado"] = pd.to_numeric(train_df["grado"], errors="coerce").astype("Int64")

    grades = sorted(train_df["grado"].dropna().unique().tolist())
    if not grades:
        raise ValueError("No encontré grados para entrenar (columna 'grado').")

    models = _candidate_models()

```

```

summary: dict = {
    "version": version,
    "created_at_utc": datetime.utcnow().isoformat(),
    "features_path": str(FEATURES_PATH),
    "n_rows_train_total": int(len(train_df)),
    "n_folds_default": 4,
    "grades": grades,
    "best_by_grade": {},
}

# Entrenar por grado
for g in grades:
    gdf = train_df[train_df["grado"] == g].copy()

    # seguridad extra: target numérico
    y = pd.to_numeric(gdf["share_grado_real"], errors="coerce").to_numpy(dtype=float)
    X = gdf[NUM_COLS + CAT_COLS].copy()

    mask = np.isfinite(y)
    X = X.loc[mask].reset_index(drop=True)
    y = y[mask]

    if len(y) < 30:
        # muy poca data: igual entrenamos, pero con holdout mínimo
        pass

    g_folds = _time_folds(gdf.loc[mask, "fecha"], n_folds=4)
    if not g_folds:
        d = pd.to_datetime(gdf.loc[mask, "fecha"], errors="coerce").dt.normalize()
        cut = d.quantile(0.8)
        tr_idx = np.where(d < cut)[0]
        va_idx = np.where(d >= cut)[0]
        if len(tr_idx) == 0 or len(va_idx) == 0:
            # no hay split, entrenamos sin validación (pero dejamos métricas NaN)
            g_folds = []
        else:
            g_folds = [(tr_idx, va_idx)]

    best_name = None
    best_score = None
    best_model = None
    all_metrics = {}

    for name, model in models.items():
        pipe = _make_pipeline(model)

        fold_stats = []
        if g_folds:
            for tr_idx, va_idx in g_folds:
                X_tr = X.iloc[tr_idx]
                y_tr = y[tr_idx]
                X_va = X.iloc[va_idx]
                y_va = y[va_idx]

                pipe.fit(X_tr, y_tr)
                pred = pipe.predict(X_va)

                fold_stats.append(_score_fold(y_va, pred))

            maes = np.array([m["mae"] for m in fold_stats], dtype=float)
            wapes = np.array([m["wape"] for m in fold_stats], dtype=float)
            smapes = np.array([m["smape"] for m in fold_stats], dtype=float)

            mae_mean = float(np.nanmean(maes))
            mae_std = float(np.nanstd(maes))
            wape_mean = float(np.nanmean(wapes))
            smape_mean = float(np.nanmean(smapes))

            # score compuesto (más bajo es mejor)
            score = 0.55 * mae_mean + 0.30 * (wape_mean if np.isfinite(wape_mean) else mae_mean) + 0.15 *
mae_std
        else:

```



```

        # sin folds posibles
        mae_mean = float("nan")
        mae_std = float("nan")
        wape_mean = float("nan")
        smape_mean = float("nan")
        score = float("inf")

    all_metrics[name] = {
        "mae_mean": mae_mean,
        "mae_std": mae_std,
        "wape_mean": wape_mean,
        "smape_mean": smape_mean,
        "score": float(score),
        "n_rows": int(len(X)),
        "n_folds": int(len(g_folds)),
    }

    if (best_score is None) or (score < best_score):
        best_score = score
        best_name = name
        best_model = model

# Fit final con todo el historial (grado g)
assert best_name is not None and best_model is not None
best_pipe = _make_pipeline(best_model)
best_pipe.fit(X, y)

# Guardar artefacto
model_path = out_dir / f"model_grade_{str(int(g))}.joblib"
dump(best_pipe, model_path)

summary["best_by_grade"][str(int(g))] = {
    "best_model": best_name,
    "metrics": all_metrics,
    "model_path": str(model_path).replace("\\", "/"),
}

print(f"[ML1 dist_grado] grado={int(g)} best={best_name} score={best_score}")

with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
    json.dump(summary, f, ensure_ascii=False, indent=2)

print(f"OK -> {out_dir}/ (models + metrics.json)")

if __name__ == "__main__":
    main()

-----
[97/164] FILE: \src\models\ml1\train_harvest_window_ml1.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

```

```
FEATURES_PATH = Path("data/features/features_harvest_window_ml1.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/harvest_window")
```

```
NUM_COLS = [
    "tallos_proy",
    "sp_month",
    "sp_weekofyear",
    "sp_doy",
    "sp_dow",
]
```

```
CAT_COLS = [
    "variedad_canon",
    "area",
    "tipo_sp",
]
```

```
def _make_pipe() -> Pipeline:
    num_pipe = Pipeline([("imputer", SimpleImputer(strategy="median"))])
    cat_pipe = Pipeline(
        [
            ("imputer", SimpleImputer(strategy="most_frequent")),
            ("onehot", OneHotEncoder(handle_unknown="ignore")),
        ]
    )
    pre = ColumnTransformer(
        [
            ("num", num_pipe, NUM_COLS),
            ("cat", cat_pipe, CAT_COLS),
        ],
        remainder="drop",
    )
    # HGB: robusto, no se va al infinito, generaliza bien con onehot sparse
    model = HistGradientBoostingRegressor(
        loss="squared_error",
        random_state=0,
        max_depth=6,
        learning_rate=0.07,
        max_iter=500,
        min_samples_leaf=25, # pooling implícito para segmentos chicos
        l2_regularization=1.0,
        early_stopping=True,
        validation_fraction=0.15,
    )
    return Pipeline([("pre", pre), ("model", model)])
```

```
def _time_split(df: pd.DataFrame, date_col: str = "fecha_sp") -> tuple[np.ndarray, np.ndarray]:
    d = pd.to_datetime(df[date_col], errors="coerce").dt.normalize()
    ok = d.notna()
    if ok.sum() < 50:
        # fallback simple
        idx = np.arange(len(df))
        cut = int(len(idx) * 0.8)
        return idx[:cut], idx[cut:]

    uniq = np.array(sorted(d[ok].unique()))
    cut = uniq[int(len(uniq) * 0.8)]
    tr = np.where(d < cut)[0]
    va = np.where(d >= cut)[0]
    if len(tr) == 0 or len(va) == 0:
        idx = np.arange(len(df))
        cut2 = int(len(idx) * 0.8)
        return idx[:cut2], idx[cut2:]
    return tr, va
```

```
def _mae(y, p) -> float:
```

```

y = np.asarray(y, dtype=float)
p = np.asarray(p, dtype=float)
return float(np.nanmean(np.abs(y - p)))

def main() -> None:
    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(FEATURES_PATH).copy()
    need = {"ciclo_id", "fecha_sp", "variedad_canon", "area", "tipo_sp"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"features_harvest_window_mll sin columnas: {sorted(miss)}")

    # asegurar columnas
    for c in NUM_COLS:
        if c not in df.columns:
            df[c] = np.nan
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"

    # ---- train start offset
    df_start = df[df["d_start_real"].notna()].copy()
    if df_start.empty:
        raise ValueError("No hay d_start_real para entrenar.")

    Xs = df_start[NUM_COLS + CAT_COLS]
    ys = pd.to_numeric(df_start["d_start_real"], errors="coerce").to_numpy(dtype=float)

    tr, va = _time_split(df_start, "fecha_sp")
    pipe_start = _make_pipe()
    pipe_start.fit(Xs.iloc[tr], ys[tr])
    pred_va = pipe_start.predict(Xs.iloc[va])
    mae_start = _mae(ys[va], pred_va)

    # ---- train harvest days
    df_days = df[df["n_harvest_days_real"].notna()].copy()
    if df_days.empty:
        raise ValueError("No hay n_harvest_days_real para entrenar.")

    Xd = df_days[NUM_COLS + CAT_COLS]
    yd = pd.to_numeric(df_days["n_harvest_days_real"], errors="coerce").to_numpy(dtype=float)

    tr2, va2 = _time_split(df_days, "fecha_sp")
    pipe_days = _make_pipe()
    pipe_days.fit(Xd.iloc[tr2], yd[tr2])
    pred_va2 = pipe_days.predict(Xd.iloc[va2])
    mae_days = _mae(yd[va2], pred_va2)

    # fit final full
    pipe_start.fit(Xs, ys)
    pipe_days.fit(Xd, yd)

    dump(pipe_start, out_dir / "model_start_offset.joblib")
    dump(pipe_days, out_dir / "model_harvest_days.joblib")

    meta = {
        "version": version,
        "created_at_utc": datetime.utcnow().isoformat(),
        "features_path": str(FEATURES_PATH).replace("\\", "/"),
        "n_train_start": int(len(df_start)),
        "n_train_days": int(len(df_days)),
        "mae_start_days_holdout": mae_start,
        "mae_harvest_days_holdout": mae_days,
        "num_cols": NUM_COLS,
        "cat_cols": CAT_COLS,
        "clips": {"d_start": [0, 180], "n_harvest_days": [1, 180]},
    }
    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:

```

```

        json.dump(meta, f, ensure_ascii=False, indent=2)

    print(f"[ML1 harvest_window] version={version}")
    print(f"MAE start_offset (days): {mae_start:.3f}")
    print(f"MAE harvest_days (days): {mae_days:.3f}")
    print(f"OK -> {out_dir}/ (models + metrics.json)")

if __name__ == "__main__":
    main()

-----
[98/164] FILE: \src\models\ml1\train_hidr_poscosecha_ml1.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

FACT_PATH = Path("data/silver/fact_hidratacion_real_post_grado_destino.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/hidr_poscosecha")

# numéricas (calendario)
NUM_COLS = ["dow", "month", "weekofyear"]

# categóricas separadas por tipo (EVITA el crash de most_frequent con mezcla dtype)
CAT_STR_COLS = ["destino"] # string
CAT_NUM_COLS = ["grado"] # num/cat

def _make_ohe() -> OneHotEncoder:
    # compat sklearn viejo/nuevo
    try:
        return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        return OneHotEncoder(handle_unknown="ignore", sparse=False)

def _make_pipeline(model) -> Pipeline:
    num_pipe = Pipeline(steps=[("imputer", SimpleImputer(strategy="median"))])

    cat_str_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="constant", fill_value="UNKNOWN")),
            ("onehot", _make_ohe()),
        ]
    )

    cat_num_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="constant", fill_value=-1)),
            ("onehot", _make_ohe()),
        ]
    )

```

```

pre = ColumnTransformer(
    transformers=[
        ("num", num_pipe, NUM_COLS),
        ("cat_str", cat_str_pipe, CAT_STR_COLS),
        ("cat_num", cat_num_pipe, CAT_NUM_COLS),
    ],
    remainder="drop",
    sparse_threshold=0.0, # fuerza denso (evita temas con HGB)
)

return Pipeline(steps=[("pre", pre), ("model", model)])

def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=400,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    }

def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    if d.isna().all():
        return []

    uniq = np.array(sorted(d.dropna().unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []

    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]

    folds: list[tuple[np.ndarray, np.ndarray]] = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        valid_start = vb.min()
        valid_end = vb.max()
        tr = (all_dates < valid_start).to_numpy()
        va = ((all_dates >= valid_start) & (all_dates <= valid_end)).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))

    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]
    return folds

def main() -> None:
    if not FACT_PATH.exists():
        raise FileNotFoundError(f"No existe: {FACT_PATH}")

    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

```

```

fact = read_parquet(FACT_PATH).copy()
fact.columns = [str(c).strip() for c in fact.columns]

need = {"fecha_cosecha", "grado", "destino"}
miss = need - set(fact.columns)
if miss:
    raise ValueError(f"fact_hidratacion_real_post_grado_destino sin columnas: {sorted(miss)}")

# target: factor_hidr si existe; si no, 1+hidr_pct
if "factor_hidr" in fact.columns:
    fact["y"] = pd.to_numeric(fact["factor_hidr"], errors="coerce")
    target_name = "factor_hidr"
elif "hidr_pct" in fact.columns:
    fact["y"] = 1.0 + pd.to_numeric(fact["hidr_pct"], errors="coerce")
    target_name = "1+hidr_pct"
else:
    raise ValueError("No encuentro ni factor_hidr ni hidr_pct en fact_hidratacion_real_post_grado_destino.")
")

fact["fecha_cosecha"] = pd.to_datetime(fact["fecha_cosecha"], errors="coerce").dt.normalize()

# grado num/cat (se queda num, se onehotea como categoría)
fact["grado"] = pd.to_numeric(fact["grado"], errors="coerce")

# destino str
fact["destino"] = fact["destino"].astype(str).str.upper().str.strip()
fact.loc[fact["destino"].isin(["", "NAN", "NONE", "NULL"]), "destino"] = np.nan

# peso base para ponderación si existe
w = None
if "peso_base_g" in fact.columns:
    w = pd.to_numeric(fact["peso_base_g"], errors="coerce")
elif "tallos" in fact.columns:
    w = pd.to_numeric(fact["tallos"], errors="coerce")

df = fact[fact["fecha_cosecha"].notna() & fact["grado"].notna() & fact["y"].notna()].copy()

# caps razonables (alineado a tu lógica histórica)
df["y"] = pd.to_numeric(df["y"], errors="coerce").clip(lower=0.80, upper=3.00)

if len(df) < 200:
    raise ValueError(f"Poca data para entrenar hidr (n={len(df)}).")

# features calendario
df["dow"] = df["fecha_cosecha"].dt.dayofweek.astype("Int64")
df["month"] = df["fecha_cosecha"].dt.month.astype("Int64")
df["weekofyear"] = df["fecha_cosecha"].dt.isocalendar().week.astype("Int64")

X = df[NUM_COLS + CAT_STR_COLS + CAT_NUM_COLS].copy()
y = df["y"].astype(float).to_numpy()

# sample_weight (opcional)
sample_weight = None
if w is not None:
    ww = w.loc[df.index]
    ww = pd.to_numeric(ww, errors="coerce").fillna(0.0).astype(float)
    # si viene todo 0, no sirve
    if float(ww.sum()) > 0:
        sample_weight = ww.to_numpy()

folds = _time_folds(df["fecha_cosecha"], n_folds=4)
if not folds:
    raise ValueError("No pude armar folds temporales para hidr.")

models = _candidate_models()

best_name = None
best_score = None
best_model = None
all_metrics: dict[str, dict] = {}

```

```

for name, model in models.items():
    pipe = _make_pipeline(model)
    maes = []
    for tr_idx, va_idx in folds:
        X_tr, y_tr = X.iloc[tr_idx], y[tr_idx]
        X_va, y_va = X.iloc[va_idx], y[va_idx]

        fit_kwargs = {}
        if sample_weight is not None:
            fit_kwargs["model__sample_weight"] = sample_weight[tr_idx]

        pipe.fit(X_tr, y_tr, **fit_kwargs)
        pred = pipe.predict(X_va)
        maes.append(float(np.mean(np.abs(y_va - pred))))

    mae_mean = float(np.mean(maes))
    mae_std = float(np.std(maes))
    score = 0.85 * mae_mean + 0.15 * mae_std

    all_metrics[name] = {
        "mae_mean": mae_mean,
        "mae_std": mae_std,
        "score": float(score),
        "n_rows": int(len(X)),
        "n_folds": int(len(folds)),
        "uses_sample_weight": bool(sample_weight is not None),
    }

    if (best_score is None) or (score < best_score):
        best_score = score
        best_name = name
        best_model = model

assert best_name is not None and best_model is not None

best_pipe = _make_pipeline(best_model)

fit_kwargs = {}
if sample_weight is not None:
    fit_kwargs["model__sample_weight"] = sample_weight

best_pipe.fit(X, y, **fit_kwargs)

model_path = out_dir / "model.joblib"
dump(best_pipe, model_path)

summary = {
    "version": version,
    "created_at_utc": datetime.utcnow().isoformat(),
    "fact_path": str(FACT_PATH).replace("\\", "/"),
    "target": target_name,
    "clip_range_apply": [0.80, 3.00],
    "features": {
        "num": NUM_COLS,
        "cat_str": CAT_STR_COLS,
        "cat_num": CAT_NUM_COLS,
    },
    "best_model": best_name,
    "metrics": all_metrics,
    "uses_sample_weight": bool(sample_weight is not None),
    "model_path": str(model_path).replace("\\", "/"),
}

with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
    json.dump(summary, f, ensure_ascii=False, indent=2)

print(f"OK -> {out_dir}/ (model.joblib + metrics.json) | best={best_name} score={best_score:.6f}")

if __name__ == "__main__":
    main()

```

```

-----
[99/164] FILE: \src\models\ml1\train_peso_tallo_grado.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import warnings

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.linear_model import Ridge
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, HistGradientBoostingRegressor
from sklearn.dummy import DummyRegressor

from common.io import read_parquet

warnings.filterwarnings("ignore")

FEATURES_PATH = Path("data/features/features_peso_tallo_grado_bloque_dia.parquet")
REGISTRY_ROOT = Path("models_registry/ml1/peso_tallo_grado")

# =====
# Config
# =====
CLIP_LOW, CLIP_HIGH = 0.60, 1.60

NUM_COLS = [
    "pct_avance_real",
    "dia_rel_cosecha_real",
    "gdc_acum_real",
    "rainfall_mm_dia",
    "horas_lluvia",
    "temp_avg_dia",
    "solar_energy_j_m2_dia",
    "wind_speed_avg_dia",
    "wind_run_dia",
    "gdc_dia",
    "dias_desde_sp",
    "gdc_acum_desde_sp",
    "dow",
    "month",
    "weekofyear",
    "peso_tallo_baseline_g",
]

CAT_COLS = ["variedad_canon", "tipo_sp", "area"]

# Candidatos para reconstruir target si viene vac o
REAL_W_CANDS = [
    "peso_tallo_real_g",
    "peso_tallo_real",
    "peso_tallo_g_real",
    "peso_tallo_obs_g",
    "peso_tallo_prom_g",
    "peso_tallo_avg_g",
    "peso_tallo_mediana_g",
    "peso_tallo_g",
    "peso_real_g",
    "peso_real",
]

BASE_W_CANDS = [
    "peso_tallo_baseline_g",
    "peso_tallo_baseline",
]

```



```

    "peso_baseline_g",
    "peso_base_g",
]

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _wape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = float(np.sum(np.abs(y_true)))
    if denom <= 1e-12:
        return float("nan")
    return float(np.sum(np.abs(y_true - y_pred)) / denom)

def _smape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    denom = (np.abs(y_true) + np.abs(y_pred))
    denom = np.where(denom < 1e-12, 1e-12, denom)
    return float(np.mean(2.0 * np.abs(y_pred - y_true) / denom))

def _make_ohe() -> OneHotEncoder:
    # compat sklearn viejo/nuevo
    try:
        return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        return OneHotEncoder(handle_unknown="ignore", sparse=False)

def _make_pipeline(model) -> Pipeline:
    # num -> median (robusto)
    num_pipe = Pipeline(steps=[("imputer", SimpleImputer(strategy="median"))])

    # cat -> constant (evita bugs dtype mixto)
    cat_pipe = Pipeline(
        steps=[
            ("imputer", SimpleImputer(strategy="constant", fill_value="UNKNOWN")),
            ("onehot", _make_ohe()),
        ]
    )

    pre = ColumnTransformer(
        transformers=[
            ("num", num_pipe, NUM_COLS),
            ("cat", cat_pipe, CAT_COLS),
        ],
        remainder="drop",
        sparse_threshold=0.0, # fuerza dense
    )
    return Pipeline(steps=[("pre", pre), ("model", model)])

def _candidate_models() -> dict[str, object]:
    return {
        "ridge": Ridge(alpha=1.0),
        "gbr": GradientBoostingRegressor(random_state=0),
        "hgb": HistGradientBoostingRegressor(random_state=0),
        "rf": RandomForestRegressor(
            n_estimators=300,
            random_state=0,
            n_jobs=-1,
            min_samples_leaf=5,
        ),
    },

```

```

}

def _time_folds(dates: pd.Series, n_folds: int = 4) -> list[tuple[np.ndarray, np.ndarray]]:
    d = pd.to_datetime(dates, errors="coerce").dt.normalize()
    if d.isna().all():
        return []

    uniq = np.array(sorted(d.dropna().unique()))
    if len(uniq) < 6:
        cut = uniq[int(len(uniq) * 0.8)]
        all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        return [(np.where(tr)[0], np.where(va)[0])] if tr.sum() > 0 and va.sum() > 0 else []

    n_folds = int(min(max(2, n_folds), max(2, len(uniq) // 2)))
    valid_blocks = np.array_split(uniq, n_folds + 1)[1:]

    folds: list[tuple[np.ndarray, np.ndarray]] = []
    all_dates = pd.to_datetime(dates, errors="coerce").dt.normalize()
    for vb in valid_blocks:
        if len(vb) == 0:
            continue
        valid_start = vb.min()
        valid_end = vb.max()
        tr = (all_dates < valid_start).to_numpy()
        va = ((all_dates >= valid_start) & (all_dates <= valid_end)).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds.append((np.where(tr)[0], np.where(va)[0]))

    if not folds:
        cut = uniq[int(len(uniq) * 0.8)]
        tr = (all_dates < cut).to_numpy()
        va = (all_dates >= cut).to_numpy()
        if tr.sum() > 0 and va.sum() > 0:
            folds = [(np.where(tr)[0], np.where(va)[0])]
    return folds

def _score_fold(y_true: np.ndarray, y_pred: np.ndarray) -> dict:
    mae = float(np.mean(np.abs(y_true - y_pred)))
    wape = _wape(y_true, y_pred)
    smape = _smape(y_true, y_pred)
    return {"mae": mae, "wape": wape, "smape": smape}

def _pick_col(df: pd.DataFrame, cands: list[str]) -> str | None:
    cols = list(df.columns)
    for c in cands:
        if c in cols:
            return c
    # match case-insensitive / trimmed
    norm = {str(c).strip().upper(): c for c in cols}
    for c in cands:
        k = str(c).strip().upper()
        if k in norm:
            return norm[k]
    return None

def _ensure_target(df: pd.DataFrame) -> tuple[pd.DataFrame, dict]:
    """
    Garantiza factor_peso_tallo_clipped:
    - si ya hay valores -> listo
    - si está vacío -> intenta reconstruir desde peso_real / peso_baseline
    """
    info = {}

    if "factor_peso_tallo_clipped" in df.columns:
        nn = int(df["factor_peso_tallo_clipped"].notna().sum())
        info["target_present_nonnull"] = nn

```

```

        if nn > 0:
            return df, info

    real_col = _pick_col(df, REAL_W_CANDS)
    base_col = _pick_col(df, BASE_W_CANDS)

    info["real_col_used"] = real_col
    info["base_col_used"] = base_col

    if real_col is None or base_col is None:
        # no se puede reconstruir
        if "factor_peso_tallo_clipped" not in df.columns:
            df["factor_peso_tallo_clipped"] = np.nan
        info["target_rebuilt"] = False
        info["target_rebuilt_reason"] = "missing real/base columns"
        return df, info

    real = pd.to_numeric(df[real_col], errors="coerce")
    base = pd.to_numeric(df[base_col], errors="coerce")

    factor = np.where((base > 0) & np.isfinite(base), real / base, np.nan)
    factor = pd.Series(factor, index=df.index)
    factor = factor.replace([np.inf, -np.inf], np.nan)

    df["factor_peso_tallo_raw"] = factor
    df["factor_peso_tallo_clipped"] = pd.to_numeric(df["factor_peso_tallo_raw"], errors="coerce").clip(
        lower=CLIP_LOW, upper=CLIP_HIGH
    )

    info["target_rebuilt"] = True
    info["target_rebuilt_nonnull"] = int(df["factor_peso_tallo_clipped"].notna().sum())
    return df, info

def _fit_constant_pipeline(constant_value: float) -> Pipeline:
    """
    Modelo fallback: predice constante (ej. 1.0).
    Se "fitea" sobre 1 fila dummy para dejar el pipeline serializable.
    """
    model = DummyRegressor(strategy="constant", constant=float(constant_value))
    pipe = _make_pipeline(model)

    X_dummy = pd.DataFrame(
        {
            **{c: [0.0] for c in NUM_COLS},
            **{c: ["UNKNOWN"] for c in CAT_COLS},
        }
    )
    y_dummy = np.array([float(constant_value)], dtype=float)
    pipe.fit(X_dummy, y_dummy)
    return pipe

def main() -> None:
    if not FEATURES_PATH.exists():
        raise FileNotFoundError(f"No existe: {FEATURES_PATH}")

    version = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
    out_dir = REGISTRY_ROOT / version
    out_dir.mkdir(parents=True, exist_ok=True)

    df = read_parquet(FEATURES_PATH).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha", "bloque_base", "variedad_canon", "grado", "peso_tallo_baseline_g"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"features_peso_tallo_grado_bloque_dia.parquet sin columnas: {sorted(miss)}")

    # Canon llaves/fechas
    df["fecha"] = _to_date(df["fecha"])
    df["bloque_base"] = _canon_int(df["bloque_base"])

```

```

df["grado"] = _canon_int(df["grado"])
df["variedad_canon"] = _canon_str(df["variedad_canon"])

# Asegurar cat cols
for c in ["tipo_sp", "area"]:
    if c not in df.columns:
        df[c] = "UNKNOWN"
df["tipo_sp"] = _canon_str(df["tipo_sp"].fillna("UNKNOWN"))
df["area"] = _canon_str(df["area"].fillna("UNKNOWN"))

# Asegurar num cols (evita strings raros en num)
for c in NUM_COLS:
    if c not in df.columns:
        df[c] = np.nan
    df[c] = pd.to_numeric(df[c], errors="coerce")

# Construir/garantizar target
df, tinfo = _ensure_target(df)

# Train set = donde target válido + fecha válida + grado válido
train_df = df[
    df["factor_peso_tallo_clipped"].notna()
    & df["fecha"].notna()
    & df["grado"].notna()
].copy()

grades = sorted(df["grado"].dropna().astype(int).unique().tolist())
models = _candidate_models()

summary: dict = {
    "version": version,
    "created_at_utc": datetime.utcnow().isoformat(),
    "features_path": str(FEATURES_PATH).replace("\\", "/"),
    "target": "factor_peso_tallo_clipped",
    "clip_range": [CLIP_LOW, CLIP_HIGH],
    "n_rows_total": int(len(df)),
    "n_rows_train_total": int(len(train_df)),
    "grades_seen": [int(g) for g in grades],
    "target_build_info": tinfo,
    "best_by_grade": {},
}

if not grades:
    # no grados => igual guardo métricas y salgo
    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)
    print(f"[WARN] No hay grados en features. OK -> {out_dir}/metrics.json")
    return

if train_df.empty:
    # no hay target: guardo modelos constantes por grado para no romper downstream
    print("[WARN] No hay filas con target para entrenar. Se generan modelos constantes (1.0) por grado.")
    for g in grades:
        g = int(g)
        pipe = _fit_constant_pipeline(1.0)
        model_path = out_dir / f"model_grade_{g}.joblib"
        dump(pipe, model_path)
        summary["best_by_grade"][str(g)] = {
            "best_model": "dummy_const_1.0",
            "metrics": {},
            "model_path": str(model_path).replace("\\", "/"),
            "n_rows_grade": 0,
        }

    with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
        json.dump(summary, f, ensure_ascii=False, indent=2)

    print(f"OK -> {out_dir}/ (models const + metrics.json)")
    return

# Entrenamiento por grado
for g in grades:

```

```

g = int(g)
gdf = train_df[train_df["grado"].astype(int) == g].copy()

if len(gdf) < 30:
    # fallback modelo constante por grado
    pipe = _fit_constant_pipeline(1.0)
    model_path = out_dir / f"model_grade_{g}.joblib"
    dump(pipe, model_path)
    summary["best_by_grade"][str(g)] = {
        "best_model": "dummy_const_1.0",
        "metrics": {},
        "model_path": str(model_path).replace("\\", "/"),
        "n_rows_grade": int(len(gdf)),
        "note": "poca data; fallback const",
    }
    print(f"[ML1 peso_tallo] grado={g} -> fallback const=1.0 (poca data n={len(gdf)})")
    continue

# Features/target
X = gdf[NUM_COLS + CAT_COLS].copy()
for c in CAT_COLS:
    X[c] = _canon_str(X[c].fillna("UNKNOWN"))

y = pd.to_numeric(gdf["factor_peso_tallo_clipped"], errors="coerce").to_numpy(dtype=float)
mask = np.isfinite(y)
if mask.sum() < 30:
    pipe = _fit_constant_pipeline(1.0)
    model_path = out_dir / f"model_grade_{g}.joblib"
    dump(pipe, model_path)
    summary["best_by_grade"][str(g)] = {
        "best_model": "dummy_const_1.0",
        "metrics": {},
        "model_path": str(model_path).replace("\\", "/"),
        "n_rows_grade": int(len(gdf)),
        "note": "target inválido; fallback const",
    }
    print(f"[ML1 peso_tallo] grado={g} -> fallback const=1.0 (target inválido)")
    continue

X = X.loc[gdf.index[mask]].copy()
y = y[mask]

folds = _time_folds(gdf.loc[X.index, "fecha"], n_folds=4)
if not folds:
    # fallback split 80/20 por fecha
    d = pd.to_datetime(gdf.loc[X.index, "fecha"], errors="coerce").dt.normalize()
    d = d.dropna()
    if d.empty:
        folds = []
    else:
        cut = d.quantile(0.8)
        d_arr = pd.to_datetime(gdf.loc[X.index, "fecha"], errors="coerce").dt.normalize().to_numpy()
        tr_idx = np.where(d_arr < np.datetime64(cut))[0]
        va_idx = np.where(d_arr >= np.datetime64(cut))[0]
        if len(tr_idx) > 0 and len(va_idx) > 0:
            folds = [(tr_idx, va_idx)]

if not folds:
    pipe = _fit_constant_pipeline(1.0)
    model_path = out_dir / f"model_grade_{g}.joblib"
    dump(pipe, model_path)
    summary["best_by_grade"][str(g)] = {
        "best_model": "dummy_const_1.0",
        "metrics": {},
        "model_path": str(model_path).replace("\\", "/"),
        "n_rows_grade": int(len(X)),
        "note": "no folds; fallback const",
    }
    print(f"[ML1 peso_tallo] grado={g} -> fallback const=1.0 (no folds)")
    continue

best_name = None

```

```

best_score = None
best_model = None
all_metrics: dict[str, dict] = {}

for name, model in models.items():
    pipe = _make_pipeline(model)
    fold_stats = []
    for tr_idx, va_idx in folds:
        X_tr = X.iloc[tr_idx]
        y_tr = y[tr_idx]
        X_va = X.iloc[va_idx]
        y_va = y[va_idx]

        pipe.fit(X_tr, y_tr)
        pred = pipe.predict(X_va)

        fold_stats.append(_score_fold(y_va, pred))

    maes = np.array([m["mae"] for m in fold_stats], dtype=float)
    wapes = np.array([m["wape"] for m in fold_stats], dtype=float)
    smapes = np.array([m["smape"] for m in fold_stats], dtype=float)

    mae_mean = float(np.nanmean(maes))
    mae_std = float(np.nanstd(maes))
    wape_mean = float(np.nanmean(wapes))
    smape_mean = float(np.nanmean(smapes))

    # score híbrido (MAE + estabilidad)
    score = 0.60 * mae_mean + 0.25 * (wape_mean if np.isfinite(wape_mean) else mae_mean) + 0.15 * mae_
std

    all_metrics[name] = {
        "mae_mean": mae_mean,
        "mae_std": mae_std,
        "wape_mean": wape_mean,
        "smape_mean": smape_mean,
        "score": float(score),
        "n_rows": int(len(X)),
        "n_folds": int(len(folds)),
    }

    if (best_score is None) or (score < best_score):
        best_score = score
        best_name = name
        best_model = model

assert best_name is not None and best_model is not None

best_pipe = _make_pipeline(best_model)
best_pipe.fit(X, y)

model_path = out_dir / f"model_grade_{g}.joblib"
dump(best_pipe, model_path)

summary["best_by_grade"][str(g)] = {
    "best_model": best_name,
    "metrics": all_metrics,
    "model_path": str(model_path).replace("\\", "/"),
    "n_rows_grade": int(len(X)),
}

print(f"[ML1 peso_tallo] grado={g} best={best_name} score={best_score:.6f}")

with open(out_dir / "metrics.json", "w", encoding="utf-8") as f:
    json.dump(summary, f, ensure_ascii=False, indent=2)

print(f"OK -> {out_dir}/ (models + metrics.json)")

if __name__ == "__main__":
    main()

```

```
-----  
[100/164] FILE: \src\models\ml2\apply_ajuste_poscosecha_ml2.py  
-----
```

```
from __future__ import annotations
```

```
import argparse  
import json  
from pathlib import Path
```

```
import numpy as np  
import pandas as pd  
from joblib import load
```

```
from src.common.io import read_parquet, write_parquet
```

```
def _project_root() -> Path:  
    return Path(__file__).resolve().parents[3]
```

```
ROOT = _project_root()  
DATA = ROOT / "data"  
GOLD = DATA / "gold"  
EVAL = DATA / "eval" / "ml2"  
MODELS_DIR = DATA / "models" / "ml2"
```

```
IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_desp_grado_dia_bloque_destino_final.parquet"
```

```
OUT_FINAL = GOLD / "pred_poscosecha_ml2_ajuste_grado_dia_bloque_destino_final.parquet"  
OUT_BACKTEST = EVAL / "backtest_factor_ml2_ajuste_poscosecha.parquet"
```

```
# -----  
# TZ-safe helpers  
# -----
```

```
def _to_naive_utc(ts: pd.Timestamp) -> pd.Timestamp:  
    if ts.tzinfo is None:  
        return ts  
    return ts.tz_convert("UTC").tz_localize(None)
```

```
def _as_of_date_naive() -> pd.Timestamp:  
    # regla: hoy-1  
    t = _to_naive_utc(pd.Timestamp.utcnow())  
    return (t.normalize() - pd.Timedelta(days=1))
```

```
def _to_date_naive(s: pd.Series) -> pd.Series:  
    dt = pd.to_datetime(s, errors="coerce")  
    try:  
        if getattr(dt.dt, "tz", None) is not None:  
            dt = dt.dt.tz_convert("UTC").dt.tz_localize(None)  
    except Exception:  
        pass  
    return dt.dt.normalize()
```

```
def _canon_str(s: pd.Series) -> pd.Series:  
    return s.astype(str).str.upper().str.strip()
```

```
def _resolve_fecha_post_pred(df: pd.DataFrame) -> str:  
    for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml1", "fecha_post_pred"]:  
        if c in df.columns:  
            return c  
    raise KeyError("No encuentro columna fecha_post_pred (final/used/ml1/seed).")
```

```
def _resolve_ajuste_ml1(df: pd.DataFrame) -> str:  
    for c in ["ajuste_ml1", "factor_ajuste_ml1", "factor_ajuste_seed", "factor_ajuste"]:  
        if c in df.columns:
```

```

        return c
    raise KeyError("No encuentro ajuste ML1 (esperaba ajuste_mll o factor_ajuste_*).")

def _weight_series(df: pd.DataFrame) -> pd.Series:
    for c in ["tallos_w", "tallos", "tallos_total_ml2", "tallos_total"]:
        if c in df.columns:
            return pd.to_numeric(df[c], errors="coerce").fillna(0.0)
    return pd.Series(1.0, index=df.index, dtype="float64")

def _latest_model(prefix: str = "ajuste_poscosecha_ml2_") -> Path:
    if not MODELS_DIR.exists():
        raise FileNotFoundError(f"No existe {MODELS_DIR}")
    files = sorted(MODELS_DIR.glob(f"{prefix}*.pkl"))
    if not files:
        raise FileNotFoundError(f"No encontré modelos en {MODELS_DIR} con prefijo {prefix}")
    return files[-1]

def _meta_path_from_model(model_path: Path) -> Path:
    # ? NO usar with_suffix("_meta.json") (inválido)
    p = str(model_path)
    if p.endswith(".pkl"):
        return Path(p.replace(".pkl", "_meta.json"))
    return model_path.with_suffix(".json")

def main(mode: str = "backtest", model_file: str | None = None) -> None:
    as_of = _as_of_date_naive()

    model_path = Path(model_file) if model_file else _latest_model()
    meta_path = _meta_path_from_model(model_path)

    meta = {}
    if meta_path.exists():
        with open(meta_path, "r", encoding="utf-8") as f:
            meta = json.load(f)

    df = read_parquet(IN_UNIVERSE).copy()
    df.columns = [str(c).strip() for c in df.columns]

    if "fecha" not in df.columns or "destino" not in df.columns:
        raise ValueError("Universe debe tener fecha y destino.")

    # ? tz-safe
    df["fecha"] = _to_date_naive(df["fecha"])
    df["destino"] = _canon_str(df["destino"])

    fecha_post_col = _resolve_fecha_post_pred(df)
    df[fecha_post_col] = _to_date_naive(df[fecha_post_col])

    ajuste_mll_col = _resolve_ajuste_mll(df)
    df[ajuste_mll_col] = pd.to_numeric(df[ajuste_mll_col], errors="coerce")

    # ? filtro as_of consistente
    df = df[df["fecha"].notna() & (df["fecha"] <= as_of)].copy()

    # features calendario
    df["dow"] = df[fecha_post_col].dt.dayofweek.astype("Int64")
    df["month"] = df[fecha_post_col].dt.month.astype("Int64")
    df["weekofyear"] = df[fecha_post_col].dt.isocalendar().week.astype("Int64")
    df["w"] = _weight_series(df)

    # columnas para el modelo
    NUM_COLS = ["dow", "month", "weekofyear"]
    CAT_COLS = ["destino"]
    if "grado" in df.columns:
        CAT_COLS.append("grado")
    df["grado"] = pd.to_numeric(df["grado"], errors="coerce").astype("Int64")

    for c in NUM_COLS:

```



```

        if c not in df.columns:
            df[c] = np.nan
    for c in CAT_COLS:
        if c not in df.columns:
            df[c] = "UNKNOWN"

    X = df[NUM_COLS + CAT_COLS]

    model = load(model_path)
    pred = model.predict(X)

    # delta: log(real/ml1)
    clip = float(meta.get("clip_delta", 1.2))
    df["delta_log_ajuste_ml2_raw"] = pd.to_numeric(pd.Series(pred), errors="coerce")
    df["delta_log_ajuste_ml2"] = df["delta_log_ajuste_ml2_raw"].clip(lower=-clip, upper=clip)

    # aplicar: ajuste_final = ajuste_ml1 * exp(delta)
    base = pd.to_numeric(df[ajuste_ml1_col], errors="coerce").replace(0, np.nan)
    df["factor_ajuste_final"] = (base * np.exp(df["delta_log_ajuste_ml2"])).astype(float)

    # guardarrailes
    clip_factor = meta.get("clip_factor_apply", [0.50, 2.00])
    try:
        lo, hi = float(clip_factor[0]), float(clip_factor[1])
    except Exception:
        lo, hi = 0.50, 2.00
    df["factor_ajuste_final"] = df["factor_ajuste_final"].clip(lower=lo, upper=hi)

    df["ml2_ajuste_model_file"] = model_path.name
    df["ml2_ajuste_clip_delta"] = clip
    df["as_of_date"] = as_of
    df["created_at"] = pd.Timestamp.utcnow()

    # backtest factor output + final output
    if mode.lower() == "backtest":
        bt_cols = ["fecha", fecha_post_col, "destino", "delta_log_ajuste_ml2", "w", "as_of_date", "created_at"]

        if "grado" in df.columns:
            bt_cols.insert(3, "grado")
        bt = df[bt_cols].copy()
        EVAL.mkdir(parents=True, exist_ok=True)
        write_parquet(bt, OUT_BACKTEST)
        print(f"[OK] BACKTEST factor: {OUT_BACKTEST} rows={len(bt):,}")

    write_parquet(df, OUT_FINAL)
    print(f"[OK] BACKTEST final : {OUT_FINAL} rows={len(df):,}")
    print(f"      model={model_path.name} clip_delta={clip} as_of_date={as_of.date()}")

if __name__ == "__main__":
    p = argparse.ArgumentParser()
    p.add_argument("--mode", default="backtest", choices=["backtest", "prod"])
    p.add_argument("--model_file", default=None)
    args = p.parse_args()
    main(mode=args.mode, model_file=args.model_file)

-----
[101/164] FILE: \src\models\ml2\apply_desp_poscosecha_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json

import numpy as np
import pandas as pd
from joblib import load

from src.common.io import read_parquet, write_parquet

```

```

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS = DATA / "models" / "ml2"

IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_hidr_grado_dia_bloque_destino_final.parquet"

OUT_FACTOR = EVAL / "backtest_factor_ml2_desp_poscosecha.parquet"
OUT_FINAL = GOLD / "pred_poscosecha_ml2_desp_grado_dia_bloque_destino_final.parquet"

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino", "grado"]

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _as_of_date() -> pd.Timestamp:
    return (pd.Timestamp.now().normalize() - pd.Timedelta(days=1))

def _latest_model(prefix: str) -> Path:
    files = sorted(MODELS.glob(f"{prefix}_*.pkl"))
    if not files:
        raise FileNotFoundError(f"No encuentro modelos en {MODELS} con prefijo {prefix}_*.pkl")
    return files[-1]

def _resolve_fecha_post_pred(df: pd.DataFrame) -> str:
    for c in ["fecha_post_pred_final", "fecha_post_pred_used", "fecha_post_pred_ml1", "fecha_post_pred"]:
        if c in df.columns:
            return c
    raise KeyError("No encuentro fecha_post_pred_* en universe.")

def _resolve_factor_desp_ml1(df: pd.DataFrame) -> str:
    for c in ["factor_desp_ml1", "factor_desp_pred_ml1", "factor_desp"]:
        if c in df.columns:
            return c
    raise KeyError("No encuentro factor_desp ML1 en universe.")

def main(mode: str = "backtest", model_file: str | None = None) -> None:
    if mode not in {"backtest", "prod"}:
        raise ValueError("--mode debe ser backtest o prod")

    as_of = _as_of_date()

    df = read_parquet(IN_UNIVERSE).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"Universe sin columnas: {sorted(miss)}")

```

```

df["fecha"] = _to_date(df["fecha"])
df = df.loc[df["fecha"].notna() & (df["fecha"] <= as_of)].copy()

df["destino"] = _canon_str(df["destino"])
if pd.api.types.is_numeric_dtype(df["grado"]):
    df["grado"] = _canon_int(df["grado"])
else:
    df["grado"] = _canon_str(df["grado"])

fpp = _resolve_fecha_post_pred(df)
fd_mll = _resolve_factor_desp_mll(df)

df[fpp] = _to_date(df[fpp])
df["fecha_post_pred_used"] = df[fpp]

# Features calendario
df["dow"] = df["fecha_post_pred_used"].dt.dayofweek.astype("Int64")
df["month"] = df["fecha_post_pred_used"].dt.month.astype("Int64")
df["weekofyear"] = df["fecha_post_pred_used"].dt.isocalendar().week.astype("Int64")

# ensure cols exist
for c in NUM_COLS:
    if c not in df.columns:
        df[c] = pd.NA
for c in CAT_COLS:
    if c not in df.columns:
        df[c] = "UNKNOWN"

X = df[NUM_COLS + CAT_COLS].copy()

model_path = (MODELS / model_file) if model_file else _latest_model("desp_poscosecha_ml2")
pipe = load(model_path)

pred = pd.to_numeric(pd.Series(pipe.predict(X)), errors="coerce")
# target era log_ratio_desp_clipped => pred es log_ratio_desp_pred
df["log_ratio_desp_pred"] = pred

# factor_desp_final = factor_desp_mll * exp(pred)
df["factor_desp_mll"] = pd.to_numeric(df[fd_mll], errors="coerce")
df["factor_desp_final_raw"] = df["factor_desp_mll"] * np.exp(df["log_ratio_desp_pred"].astype(float))

# clip factor final
df["factor_desp_final"] = pd.to_numeric(df["factor_desp_final_raw"], errors="coerce").clip(lower=0.05, upper=1.00)

# outputs
EVAL.mkdir(parents=True, exist_ok=True)
GOLD.mkdir(parents=True, exist_ok=True)

fac = df[
    [
        "fecha",
        "fecha_post_pred_used",
        "bloque_base",
        "variedad_canon",
        "grado",
        "destino",
        "factor_desp_mll",
        "log_ratio_desp_pred",
        "factor_desp_final",
    ]
].copy()
fac["model_file"] = model_path.name
fac["as_of_date"] = as_of
fac["created_at"] = pd.Timestamp(datetime.now()).normalize()

write_parquet(fac, OUT_FACTOR)
write_parquet(df, OUT_FINAL)

print(f"[OK] BACKTEST factor: {OUT_FACTOR} rows={len(fac):,}")
print(f"[OK] BACKTEST final : {OUT_FINAL} rows={len(df):,}")
print(f"      model={model_path.name} as_of_date={as_of.date()}")

```

```

if __name__ == "__main__":
    # argparse minimalista (sin depender de fire/typer)
    import argparse

    ap = argparse.ArgumentParser()
    ap.add_argument("--mode", default="backtest", choices=["backtest", "prod"])
    ap.add_argument("--model-file", default=None)
    args = ap.parse_args()

    main(mode=args.mode, model_file=args.model_file)

-----
[102/164] FILE: \src\models\ml2\apply_dh_poscosecha_ml2.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import numpy as np
import pandas as pd
from joblib import load

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS = DATA / "models" / "ml2"

# ? IMPORTANTE: usar FULL (no el dh ?delgado?)
IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_full_grado_dia_bloque_destino.parquet"

OUT_FACTOR = EVAL / "backtest_factor_ml2_dh_poscosecha.parquet"
OUT_FINAL = GOLD / "pred_poscosecha_ml2_dh_grado_dia_bloque_destino_final.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _as_of_date_today_minus_1() -> pd.Timestamp:
    return (pd.Timestamp.now().normalize() - pd.Timedelta(days=1)).normalize()

def _latest_model(prefix: str) -> tuple[Path, dict]:
    files = sorted(MODELS.glob(f"{prefix}_*.pkl"))
    if not files:
        raise FileNotFoundError(f"No hay modelos en {MODELS} con prefijo {prefix}_*.pkl")
    model_path = files[-1]
    meta_path = MODELS / (model_path.stem + "_meta.json")
    if not meta_path.exists():
        raise FileNotFoundError(f"No existe meta: {meta_path}")
    with open(meta_path, "r", encoding="utf-8") as f:
        meta = json.load(f)

```

```

return model_path, meta

def _pick_first(df: pd.DataFrame, cands: list[str]) -> str | None:
    for c in cands:
        if c in df.columns:
            return c
    return None

def _resolve_dh_mll_col(df: pd.DataFrame) -> str:
    c = _pick_first(df, ["dh_dias_mll", "dh_dias_pred_mll", "dh_dias", "dh_dias_med"])
    if c is None:
        raise KeyError("No encuentro DH baseline en universe. Espero: dh_dias_mll / dh_dias_pred_mll / dh_dias / dh_dias_med")
    return c

def _resolve_fecha_post_pred_mll(df: pd.DataFrame) -> str | None:
    return _pick_first(df, ["fecha_post_pred_mll", "fecha_post_pred", "fecha_post_pred_seed"])

def main(mode: str = "backtest") -> None:
    as_of_date = _as_of_date_today_minus_1()
    created_at = pd.Timestamp.utcnow()

    model_path, meta = _latest_model("dh_poscosecha_ml2")
    pipe = load(model_path)

    df = read_parquet(IN_UNIVERSE).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino"}
    miss = need - set(df.columns)
    if miss:
        raise KeyError(f"Universe sin columnas: {sorted(miss)}")

    # Canon
    df["fecha"] = _to_date(df["fecha"])
    df["destino"] = _canon_str(df["destino"])
    df["grado"] = _canon_int(df["grado"])
    df["variedad_canon"] = _canon_str(df["variedad_canon"])
    df["bloque_base"] = _canon_str(df["bloque_base"])

    # filtro backtest <= as_of_date
    df = df.loc[df["fecha"].notna() & (df["fecha"] <= as_of_date)].copy()

    dh_mll_col = _resolve_dh_mll_col(df)
    df["dh_mll_used"] = pd.to_numeric(df[dh_mll_col], errors="coerce")

    # Features calendario sobre fecha de cosecha (misma lógica que tu ML1)
    df["dow"] = df["fecha"].dt.dayofweek.astype("Int64")
    df["month"] = df["fecha"].dt.month.astype("Int64")
    df["weekofyear"] = df["fecha"].dt.isocalendar().week.astype("Int64")

    # X
    # (si quieres meter grado/destino como cat, hazlo igual que train; aquí asumo que tu train ya coincide)
    X = df[["dow", "month", "weekofyear", "destino", "grado"]].copy()
    # Asegurar tipos
    X["destino"] = X["destino"].astype(str)
    X["grado"] = pd.to_numeric(X["grado"], errors="coerce").astype("Int64").astype(str)

    # Predicción delta (en días)
    pred_delta = pd.to_numeric(pd.Series(pipe.predict(X)), errors="coerce")

    clip_delta = float(meta.get("clip_delta_days", 5))
    pred_delta_clip = pred_delta.clip(lower=-clip_delta, upper=clip_delta)

    # final DH
    df["dh_delta_ml2_raw"] = pred_delta
    df["dh_delta_ml2"] = pred_delta_clip

```

```

df["dh_dias_final"] = (
    pd.to_numeric(df["dh_ml1_used"], errors="coerce")
    .fillna(0.0)
    .astype(float)
    + df["dh_delta_ml2"].fillna(0.0).astype(float)
)
df["dh_dias_final"] = np rint(df["dh_dias_final"]).astype("Int64").clip(lower=0, upper=30)

# fecha_post_pred_final (si ya existe una fecha_post_pred_ml1, la recalculamos; si no, la creamos)
df["fecha_post_pred_final"] = df["fecha"] + pd.to_timedelta(df["dh_dias_final"].fillna(0).astype(int), unit="D")

# meta
df["ml2_dh_model_file"] = model_path.name
df["as_of_date"] = as_of_date
df["created_at"] = created_at

# factor output (ligero)
out_factor = df[[
    "fecha", "bloque_base", "variedad_canon", "grado", "destino",
    "dh_ml1_used", "dh_delta_ml2_raw", "dh_delta_ml2", "dh_dias_final",
    "fecha_post_pred_final",
    "ml2_dh_model_file", "as_of_date", "created_at"
]].copy()

EVAL.mkdir(parents=True, exist_ok=True)
GOLD.mkdir(parents=True, exist_ok=True)

write_parquet(out_factor, OUT_FACTOR)
write_parquet(df, OUT_FINAL)

print(f"[OK] BACKTEST factor: {OUT_FACTOR} rows={len(out_factor):,}")
print(f"[OK] BACKTEST final : {OUT_FINAL} rows={len(df):,}")
print(f"      model={model_path.name} clip_delta={clip_delta} as_of_date={as_of_date.date()}")

if __name__ == "__main__":
    import argparse

    ap = argparse.ArgumentParser()
    ap.add_argument("--mode", default="backtest", choices=["backtest"])
    args = ap.parse_args()

    main(mode=args.mode)

-----
[103/164] FILE: \src\models\ml2\apply_harvest_horizon_ml2.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import argparse
import json
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"
MODELS_DIR = DATA_DIR / "models" / "ml2"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

```

```

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"
IN_GRID_ML1 = GOLD_DIR / "universe_harvest_grid_ml1.parquet"
IN_CLIMA = SILVER_DIR / "dim_clima_bloque_dia.parquet"

# ML2 Inicio (SoH) outputs
IN_SOH_FINAL_PROD = GOLD_DIR / "pred_harvest_start_final_ml2.parquet"
IN_SOH_FACTOR_PROD = GOLD_DIR / "factors" / "factor_ml2_harvest_start.parquet"

IN_SOH_FINAL_BT = EVAL_DIR / "backtest_pred_harvest_start_final_ml2.parquet"
IN_SOH_FACTOR_BT = EVAL_DIR / "backtest_factor_ml2_harvest_start.parquet"

# Outputs
OUT_FACTOR_PROD = GOLD_DIR / "factors" / "factor_ml2_harvest_horizon.parquet"
OUT_FINAL_PROD = GOLD_DIR / "pred_harvest_horizon_final_ml2.parquet"

OUT_FACTOR_BT = EVAL_DIR / "backtest_factor_ml2_harvest_horizon.parquet"
OUT_FINAL_BT = EVAL_DIR / "backtest_pred_harvest_horizon_final_ml2.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _safe_div(a: pd.Series, b: pd.Series) -> pd.Series:
    b = b.replace(0, np.nan)
    return (a / b).fillna(0.0)

def _build_cycle_header(grid: pd.DataFrame) -> pd.DataFrame:
    g = grid.copy()
    g["harvest_start_pred"] = _to_date(g["harvest_start_pred"])
    g["harvest_end_pred"] = _to_date(g["harvest_end_pred"])
    g["fecha_sp"] = _to_date(g["fecha_sp"])

    head = (
        g.groupby("ciclo_id", as_index=False)
        .agg(
            bloque_base=("bloque_base", "first"),
            variedad_canon=("variedad_canon", "first"),
            area=("area", "first"),
            tipo_sp=("tipo_sp", "first"),
            estado=("estado", "first"),
            tallos_proy=("tallos_proy", "first"),
            fecha_sp=("fecha_sp", "first"),
            harvest_start_pred=("harvest_start_pred", "min"),
            harvest_end_pred=("harvest_end_pred", "max"),
            n_harvest_days_pred=("n_harvest_days_pred", "max"),
            ml1_version=("ml1_version", "max"),
        )
    )
    head["days_sp_to_start_pred"] = (head["harvest_start_pred"] - head["fecha_sp"]).dt.days
    return head

def _features_asof(clima: pd.DataFrame, base: pd.DataFrame, as_of_date: pd.Timestamp) -> pd.DataFrame:
    """
    base: columns ciclo_id, bloque_base, fecha_sp
    returns 1 row per ciclo_id
    """
    base = base[["ciclo_id", "bloque_base", "fecha_sp"]].drop_duplicates().copy()
    base["bloque_base"] = _canon_str(base["bloque_base"])
    base["fecha_sp"] = _to_date(base["fecha_sp"])

    cl = clima.copy()
    cl["fecha"] = _to_date(cl["fecha"])
    cl["bloque_base"] = _canon_str(cl["bloque_base"])
    cl = cl.loc[cl["fecha"] <= as_of_date, :].copy()

```

```

m = base.merge(cl, on="bloque_base", how="left")
m = m.loc[(m["fecha"] >= m["fecha_sp"]) & (m["fecha"] <= as_of_date), :].copy()

if m.empty:
    out = base[["ciclo_id"]].copy()
    for c in [
        "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
        "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
        "solar_cum_sp", "solar_7d",
        "temp_avg_7d",
    ]:
        out[c] = 0.0
    return out

m["gdc_dia"] = pd.to_numeric(m["gdc_dia"], errors="coerce").fillna(0.0)
m["rainfall_mm_dia"] = pd.to_numeric(m["rainfall_mm_dia"], errors="coerce").fillna(0.0)
m["solar_energy_j_m2_dia"] = pd.to_numeric(m["solar_energy_j_m2_dia"], errors="coerce").fillna(0.0)
m["temp_avg_dia"] = pd.to_numeric(m["temp_avg_dia"], errors="coerce")
m["en_lluvia_dia"] = pd.to_numeric(m["en_lluvia_dia"], errors="coerce").fillna(0.0)

m = m.sort_values(["ciclo_id", "fecha"])

def _roll_sum(s: pd.Series, w: int) -> pd.Series:
    return s.rolling(window=w, min_periods=1).sum()

def _roll_mean(s: pd.Series, w: int) -> pd.Series:
    return s.rolling(window=w, min_periods=1).mean()

m["gdc_7d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 7))
m["gdc_14d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 14))
m["rain_7d"] = m.groupby("ciclo_id")["rainfall_mm_dia"].transform(lambda s: _roll_sum(s, 7))
m["solar_7d"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].transform(lambda s: _roll_sum(s, 7))
m["enlluvia_days_7d"] = m.groupby("ciclo_id")["en_lluvia_dia"].transform(lambda s: _roll_sum(s, 7))
m["temp_avg_7d"] = m.groupby("ciclo_id")["temp_avg_dia"].transform(lambda s: _roll_mean(s, 7))

m["gdc_cum_sp"] = m.groupby("ciclo_id")["gdc_dia"].cumsum()
m["rain_cum_sp"] = m.groupby("ciclo_id")["rainfall_mm_dia"].cumsum()
m["solar_cum_sp"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].cumsum()

last = m.groupby("ciclo_id", as_index=False).tail(1).copy()
last["days_from_sp"] = (as_of_date - last["fecha_sp"]).dt.days
last["days_from_sp"] = pd.to_numeric(last["days_from_sp"], errors="coerce").fillna(0).clip(lower=0)
last["gdc_per_day"] = _safe_div(last["gdc_cum_sp"], last["days_from_sp"].replace(0, np.nan))

feat_cols = [
    "ciclo_id",
    "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
    "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
    "solar_cum_sp", "solar_7d",
    "temp_avg_7d",
]
last = last[feat_cols].copy()

out = base[["ciclo_id"]].merge(last, on="ciclo_id", how="left")

for c in [
    "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
    "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
    "solar_cum_sp", "solar_7d",
]:
    out[c] = pd.to_numeric(out[c], errors="coerce").fillna(0.0)

out["temp_avg_7d"] = pd.to_numeric(out["temp_avg_7d"], errors="coerce")
med = out["temp_avg_7d"].median()
if pd.isna(med):
    med = 0.0
out["temp_avg_7d"] = out["temp_avg_7d"].fillna(med)

return out

def _load_latest_model() -> tuple[object, dict]:

```



```

metas = sorted(MODELS_DIR.glob("harvest_horizon_ml2_*_meta.json"))
if not metas:
    raise FileNotFoundError(f"No ML2 meta found in {MODELS_DIR}")
meta_path = metas[-1]
meta = json.loads(meta_path.read_text(encoding="utf-8"))
model_path = MODELS_DIR / f"harvest_horizon_ml2_{meta['run_id']}.pkl"
import joblib
model = joblib.load(model_path)
return model, meta

def _parse_args() -> argparse.Namespace:
    p = argparse.ArgumentParser()
    p.add_argument("--mode", choices=["prod", "backtest"], default="prod")
    p.add_argument("--asof", default=None, help="Override as_of_date (YYYY-MM-DD) for prod mode; default=today")
    return p.parse_args()

def _pick_soh_inputs(mode: str) -> tuple[Path, Path]:
    if mode == "backtest":
        return IN_SOH_FINAL_BT, IN_SOH_FACTOR_BT
    return IN_SOH_FINAL_PROD, IN_SOH_FACTOR_PROD

def main() -> None:
    args = _parse_args()
    model, meta = _load_latest_model()

    ciclo = read_parquet(IN_CICLO).copy()
    grid = read_parquet(IN_GRID_ML1).copy()
    clima = read_parquet(IN_CLIMA).copy()

    ciclo["bloque_base"] = _canon_str(ciclo["bloque_base"])
    ciclo["fecha_sp"] = _to_date(ciclo["fecha_sp"])
    ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])
    ciclo["estado"] = _canon_str(ciclo["estado"])

    grid["bloque_base"] = _canon_str(grid["bloque_base"])
    grid["variedad_canon"] = _canon_str(grid["variedad_canon"])

    head = _build_cycle_header(grid)
    df = ciclo.merge(head, on="ciclo_id", how="inner", suffixes=("", "_ml1"))

    # as_of_date
    if args.mode == "prod":
        if args.asof:
            as_of = pd.Timestamp(args.asof).normalize()
        else:
            as_of = pd.Timestamp(datetime.now()).normalize() - pd.Timedelta(days=1)
        df["as_of_date"] = as_of
    else:
        # backtest causal: usar inicio real - 1
        df = df.loc[df["fecha_inicio_cosecha"].notna(), :].copy()
        df["as_of_date"] = df["fecha_inicio_cosecha"] - pd.to_timedelta(1, unit="D")

    # --- SoH (ML2 Inicio): harvest_start_final + pred_error_start_days ---
    soh_final_path, soh_factor_path = _pick_soh_inputs(args.mode)

    if not soh_final_path.exists() or not soh_factor_path.exists():
        raise FileNotFoundError(
            f"Missing SoH inputs for mode={args.mode}. Need:\n {soh_final_path}\n {soh_factor_path}"
        )

    soh_final = read_parquet(soh_final_path).copy()
    soh_factor = read_parquet(soh_factor_path).copy()

    soh_final["harvest_start_pred"] = _to_date(soh_final["harvest_start_pred"])
    soh_final["harvest_start_final"] = _to_date(soh_final["harvest_start_final"])
    soh_factor["pred_error_start_days"] = pd.to_numeric(soh_factor["pred_error_start_days"], errors="coerce")

    soh_map = soh_final[["ciclo_id", "harvest_start_final"]].drop_duplicates("ciclo_id")

```

```

err_map = soh_factor[["ciclo_id", "pred_error_start_days"]].drop_duplicates("ciclo_id")

df = df.merge(soh_map, on="ciclo_id", how="left").merge(err_map, on="ciclo_id", how="left")
df["pred_error_start_days"] = pd.to_numeric(df["pred_error_start_days"], errors="coerce").fillna(0.0)

# fallback si algún ciclo no tiene harvest_start_final (debería ser raro)
df["harvest_start_final"] = df["harvest_start_final"].fillna(df["harvest_start_pred"])

# calendario
df["dow"] = df["as_of_date"].dt.dayofweek
df["month"] = df["as_of_date"].dt.month
df["weekofyear"] = df["as_of_date"].dt.isocalendar().week.astype(int)
df["days_from_sp"] = (df["as_of_date"] - df["fecha_sp"]).dt.days

# clima features por grupo (eficiente en backtest)
feats = []
for as_of_date, chunk in df.groupby("as_of_date"):
    base = chunk[["ciclo_id", "bloque_base", "fecha_sp"]]
    feats.append(_features_asof(clima, base, pd.Timestamp(as_of_date)))
feat = pd.concat(feats, ignore_index=True) if feats else pd.DataFrame(columns=["ciclo_id"])

df = df.merge(feat, on="ciclo_id", how="left")

# imputaciones numéricas como en meta
for c in meta.get("num_cols", []):
    if c in df.columns:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)

# cat
for c in meta.get("cat_cols", []):
    if c in df.columns:
        df[c] = df[c].astype(str).fillna("")

feature_cols = meta["num_cols"] + meta["cat_cols"]
X = df[feature_cols].copy()

pred_err = model.predict(X).astype(float)
lo, hi = meta["guardrails"]["clip_error_days"]
pred_err = np.clip(pred_err, lo, hi)

df["pred_error_horizon_days"] = pred_err

# Horizon final (min 7)
df["n_harvest_days_pred"] = pd.to_numeric(df["n_harvest_days_pred"], errors="coerce").fillna(0.0)
df["n_harvest_days_final"] = df["n_harvest_days_pred"] + df["pred_error_horizon_days"]
df["n_harvest_days_final"] = df["n_harvest_days_final"].clip(lower=meta["guardrails"]["min_horizon_days"])

# harvest_end_final derivado desde harvest_start_final
# Usamos "-1" para mantener consistencia si duración cuenta día 1 como inicio
df["harvest_end_final"] = df["harvest_start_final"] + pd.to_timedelta(df["n_harvest_days_final"] - 1, unit="D")

out_factor = df[[
    "ciclo_id", "bloque_base", "variedad", "variedad_canon", "estado",
    "fecha_sp", "fecha_inicio_cosecha",
    "as_of_date",
    "harvest_start_pred", "harvest_start_final",
    "n_harvest_days_pred",
    "pred_error_start_days",
    "pred_error_horizon_days",
    "n_harvest_days_final",
    "harvest_end_pred",
    "harvest_end_final",
    "mll_version",
]].copy()

out_factor["ml2_run_id"] = meta["run_id"]
out_factor["created_at"] = pd.Timestamp(datetime.now()).normalize()

out_final = out_factor[[
    "ciclo_id", "bloque_base", "variedad_canon", "fecha_sp",
    "as_of_date",

```

```

        "harvest_start_pred", "harvest_start_final",
        "n_harvest_days_pred", "n_harvest_days_final",
        "harvest_end_pred", "harvest_end_final",
        "ml1_version", "ml2_run_id", "created_at",
    ]].copy()

    if args.mode == "prod":
        OUT_FACTOR_PROD.parent.mkdir(parents=True, exist_ok=True)
        write_parquet(out_factor, OUT_FACTOR_PROD)
        write_parquet(out_final, OUT_FINAL_PROD)
        print(f"[OK] PROD factor: {OUT_FACTOR_PROD} rows={len(out_factor):,}")
        print(f"[OK] PROD final : {OUT_FINAL_PROD} rows={len(out_final):,}")
    else:
        EVAL_DIR.mkdir(parents=True, exist_ok=True)
        write_parquet(out_factor, OUT_FACTOR_BT)
        write_parquet(out_final, OUT_FINAL_BT)
        print(f"[OK] BACKTEST factor: {OUT_FACTOR_BT} rows={len(out_factor):,}")
        print(f"[OK] BACKTEST final : {OUT_FINAL_BT} rows={len(out_final):,}")

if __name__ == "__main__":
    main()

```

```

-----
[104/164] FILE: \src\models\ml2\apply_harvest_start_ml2.py
-----

```

```

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import argparse
import json
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"
MODELS_DIR = DATA_DIR / "models" / "ml2"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_CICLO = SILVER_DIR / "fact_ciclo_maestro.parquet"
IN_GRID_ML1 = GOLD_DIR / "universe_harvest_grid_ml1.parquet"
IN_CLIMA = SILVER_DIR / "dim_clima_bloque_dia.parquet"

OUT_FACTOR_PROD = GOLD_DIR / "factors" / "factor_ml2_harvest_start.parquet"
OUT_FINAL_PROD = GOLD_DIR / "pred_harvest_start_final_ml2.parquet"

OUT_FACTOR_BT = EVAL_DIR / "backtest_factor_ml2_harvest_start.parquet"
OUT_FINAL_BT = EVAL_DIR / "backtest_pred_harvest_start_final_ml2.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _safe_div(a: pd.Series, b: pd.Series) -> pd.Series:
    b = b.replace(0, np.nan)
    return (a / b).fillna(0.0)

```

```

def _build_cycle_header(grid: pd.DataFrame) -> pd.DataFrame:
    g = grid.copy()
    g["harvest_start_pred"] = _to_date(g["harvest_start_pred"])
    g["harvest_end_pred"] = _to_date(g["harvest_end_pred"])
    g["fecha_sp"] = _to_date(g["fecha_sp"])

    head = (
        g.groupby("ciclo_id", as_index=False)
        .agg(
            bloque_base=("bloque_base", "first"),
            variedad_canon=("variedad_canon", "first"),
            area=("area", "first"),
            tipo_sp=("tipo_sp", "first"),
            estado=("estado", "first"),
            tallos_proy=("tallos_proy", "first"),
            fecha_sp=("fecha_sp", "first"), # ? importante
            harvest_start_pred=("harvest_start_pred", "min"),
            harvest_end_pred=("harvest_end_pred", "max"),
            n_harvest_days_pred=("n_harvest_days_pred", "max"),
            mll_version=("mll_version", "max"),
        )
    )

    head["days_sp_to_start_pred"] = (head["harvest_start_pred"] - head["fecha_sp"]).dt.days
    return head


def _features_asof(clima: pd.DataFrame, ciclo_chunk: pd.DataFrame, as_of_date: pd.Timestamp) -> pd.DataFrame:
    """
    Devuelve features por ciclo_id usando clima entre [fecha_sp, as_of_date].
    IMPORTANT:
    - usa gdc_dia acumulado
    - si un ciclo no tiene clima en el rango, devuelve ceros (no desaparece)
    """
    base = ciclo_chunk[["ciclo_id", "bloque_base", "fecha_sp"]].drop_duplicates().copy()
    base["bloque_base"] = _canon_str(base["bloque_base"])
    base["fecha_sp"] = _to_date(base["fecha_sp"])

    cl = clima.copy()
    cl["fecha"] = _to_date(cl["fecha"])
    cl["bloque_base"] = _canon_str(cl["bloque_base"])
    cl = cl.loc[cl["fecha"] <= as_of_date, :].copy()

    m = base.merge(cl, on="bloque_base", how="left")
    m = m.loc[(m["fecha"] >= m["fecha_sp"]) & (m["fecha"] <= as_of_date), :].copy()

    # Si no hay filas de clima para ningún ciclo, devolvemos ceros
    if m.empty:
        out = base[["ciclo_id"]].copy()
        for c in [
            "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
            "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
            "solar_cum_sp", "solar_7d",
            "temp_avg_7d",
        ]:
            out[c] = 0.0
        return out

    # IMPORTANT: usar gdc_dia y acumular (ignorar gdc_base)
    m["gdc_dia"] = pd.to_numeric(m["gdc_dia"], errors="coerce").fillna(0.0)
    m["rainfall_mm_dia"] = pd.to_numeric(m["rainfall_mm_dia"], errors="coerce").fillna(0.0)
    m["solar_energy_j_m2_dia"] = pd.to_numeric(m["solar_energy_j_m2_dia"], errors="coerce").fillna(0.0)
    m["temp_avg_dia"] = pd.to_numeric(m["temp_avg_dia"], errors="coerce")
    m["en_lluvia_dia"] = pd.to_numeric(m["en_lluvia_dia"], errors="coerce").fillna(0.0)

    m = m.sort_values(["ciclo_id", "fecha"])

    def _roll_sum(s: pd.Series, w: int) -> pd.Series:
        return s.rolling(window=w, min_periods=1).sum()

```

```

def _roll_mean(s: pd.Series, w: int) -> pd.Series:
    return s.rolling(window=w, min_periods=1).mean()

m["gdc_7d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 7))
m["gdc_14d"] = m.groupby("ciclo_id")["gdc_dia"].transform(lambda s: _roll_sum(s, 14))
m["rain_7d"] = m.groupby("ciclo_id")["rainfall_mm_dia"].transform(lambda s: _roll_sum(s, 7))
m["solar_7d"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].transform(lambda s: _roll_sum(s, 7))
m["enlluvia_days_7d"] = m.groupby("ciclo_id")["en_lluvia_dia"].transform(lambda s: _roll_sum(s, 7))
m["temp_avg_7d"] = m.groupby("ciclo_id")["temp_avg_dia"].transform(lambda s: _roll_mean(s, 7))

m["gdc_cum_sp"] = m.groupby("ciclo_id")["gdc_dia"].cumsum()
m["rain_cum_sp"] = m.groupby("ciclo_id")["rainfall_mm_dia"].cumsum()
m["solar_cum_sp"] = m.groupby("ciclo_id")["solar_energy_j_m2_dia"].cumsum()

last = m.groupby("ciclo_id", as_index=False).tail(1).copy()
last["days_from_sp"] = (as_of_date - last["fecha_sp"]).dt.days
last["days_from_sp"] = pd.to_numeric(last["days_from_sp"], errors="coerce").fillna(0).clip(lower=0)
last["gdc_per_day"] = _safe_div(last["gdc_cum_sp"], last["days_from_sp"].replace(0, np.nan))

feat_cols = [
    "ciclo_id",
    "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
    "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
    "solar_cum_sp", "solar_7d",
    "temp_avg_7d",
]
last = last[feat_cols].copy()

# ? garantizar que todos los ciclos queden (si algunos no tuvieron filas)
out = base[["ciclo_id"]].merge(last, on="ciclo_id", how="left")

# imputaciones
for c in [
    "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
    "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
    "solar_cum_sp", "solar_7d",
]:
    out[c] = pd.to_numeric(out[c], errors="coerce").fillna(0.0)

out["temp_avg_7d"] = pd.to_numeric(out["temp_avg_7d"], errors="coerce")
# si todo es NaN -> 0
med = out["temp_avg_7d"].median()
if pd.isna(med):
    med = 0.0
out["temp_avg_7d"] = out["temp_avg_7d"].fillna(med)

return out

def _load_latest_model() -> tuple[object, dict]:
    metas = sorted(MODELS_DIR.glob("harvest_start_ml2_*_meta.json"))
    if not metas:
        raise FileNotFoundError(f"No ML2 meta found in {MODELS_DIR}")
    meta_path = metas[-1]
    meta = json.loads(meta_path.read_text(encoding="utf-8"))
    model_path = MODELS_DIR / f"harvest_start_ml2_{meta['run_id']}.pkl"
    import joblib
    model = joblib.load(model_path)
    return model, meta

def _parse_args() -> argparse.Namespace:
    p = argparse.ArgumentParser()
    p.add_argument("--mode", choices=["prod", "backtest"], default="prod")
    p.add_argument(
        "--asof",
        default=None,
        help="Override as_of_date (YYYY-MM-DD) for prod mode. If not set, uses today-1.",
    )
    return p.parse_args()

```

```

def main() -> None:
    args = _parse_args()
    model, meta = _load_latest_model()

    ciclo = read_parquet(IN_CICLO).copy()
    grid = read_parquet(IN_GRID_ML1).copy()
    clima = read_parquet(IN_CLIMA).copy()

    # canon
    ciclo["bloque_base"] = _canon_str(ciclo["bloque_base"])
    ciclo["fecha_sp"] = _to_date(ciclo["fecha_sp"])
    ciclo["fecha_inicio_cosecha"] = _to_date(ciclo["fecha_inicio_cosecha"])

    grid["bloque_base"] = _canon_str(grid["bloque_base"])
    grid["variedad_canon"] = _canon_str(grid["variedad_canon"])

    head = _build_cycle_header(grid)
    df = ciclo.merge(head, on="ciclo_id", how="inner", suffixes=("", "_ml1"))

    # -----
    # Define as_of_date
    # -----
    if args.mode == "prod":
        if args.asof:
            as_of = pd.Timestamp(args.asof).normalize()
        else:
            as_of = pd.Timestamp(datetime.now()).normalize() - pd.Timedelta(days=1)
        df["as_of_date"] = as_of

    else: # backtest
        # as_of = (fecha_inicio_cosecha - 1) por ciclo (si no hay real, se excluye del backtest)
        df = df.loc[df["fecha_inicio_cosecha"].notna(), :].copy()
        df["as_of_date"] = df["fecha_inicio_cosecha"] - pd.to_timedelta(1, unit="D")

    # features calendario (por fila)
    df["dow"] = df["as_of_date"].dt.dayofweek
    df["month"] = df["as_of_date"].dt.month
    df["weekofyear"] = df["as_of_date"].dt.isocalendar().week.astype(int)

    # -----
    # Build clima features as-of
    # En prod: un solo as_of.
    # En backtest: muchos as_of distintos -> procesar por grupo para eficiencia.
    # -----
    feats = []
    for as_of_date, chunk in df.groupby("as_of_date"):
        feats.append(_features_asof(clima, chunk[["ciclo_id", "bloque_base", "fecha_sp"]], pd.Timestamp(as_of_
date)))
    feat = pd.concat(feats, ignore_index=True) if feats else pd.DataFrame(columns=["ciclo_id"])

    df = df.merge(feat, on="ciclo_id", how="left")

    # imputaciones num según meta
    for c in meta.get("num_cols", []):
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)

    # temp (ya viene imputada en _features_asof pero mantenemos robustez)
    if "temp_avg_7d" in df.columns:
        df["temp_avg_7d"] = pd.to_numeric(df["temp_avg_7d"], errors="coerce")
        med = df["temp_avg_7d"].median()
        if pd.isna(med):
            med = 0.0
        df["temp_avg_7d"] = df["temp_avg_7d"].fillna(med)

    feature_cols = meta["num_cols"] + meta["cat_cols"]
    X = df[feature_cols].copy()

    pred_err = model.predict(X).astype(float)
    lo, hi = meta["guardrails"]["clip_error_days"]
    pred_err = np.clip(pred_err, lo, hi)

```

```

df["pred_error_start_days"] = pred_err
df["harvest_start_final"] = df["harvest_start_pred"] + pd.to_timedelta(df["pred_error_start_days"], unit="
D")

```

```

out_factor = df[[
    "ciclo_id", "bloque_base", "variedad", "variedad_canon", "estado",
    "fecha_sp", "fecha_inicio_cosecha",
    "as_of_date",
    "harvest_start_pred", "pred_error_start_days", "harvest_start_final",
    "m11_version",
]].copy()

```

```

out_factor["ml2_run_id"] = meta["run_id"]
out_factor["created_at"] = pd.Timestamp(datetime.now()).normalize()

```

```

out_final = out_factor[[
    "ciclo_id", "bloque_base", "variedad_canon", "fecha_sp",
    "as_of_date",
    "harvest_start_pred", "harvest_start_final",
    "m11_version", "ml2_run_id", "created_at",
]].copy()

```

```

# -----
# Save
# -----

```

```

if args.mode == "prod":
    OUT_FACTOR_PROD.mkdir(parents=True, exist_ok=True)
    write_parquet(out_factor, OUT_FACTOR_PROD)
    write_parquet(out_final, OUT_FINAL_PROD)
    print(f"[OK] PROD factor: {OUT_FACTOR_PROD} rows={len(out_factor):,}")
    print(f"[OK] PROD final : {OUT_FINAL_PROD} rows={len(out_final):,}")
else:
    EVAL_DIR.mkdir(parents=True, exist_ok=True)
    write_parquet(out_factor, OUT_FACTOR_BT)
    write_parquet(out_final, OUT_FINAL_BT)
    print(f"[OK] BACKTEST factor: {OUT_FACTOR_BT} rows={len(out_factor):,}")
    print(f"[OK] BACKTEST final : {OUT_FINAL_BT} rows={len(out_final):,}")

```

```

if __name__ == "__main__":
    main()

```

```

-----
[105/164] FILE: \src\models\ml2\apply_hidr_poscosecha_ml2.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import json
import numpy as np
import pandas as pd
from joblib import load

```

```

from src.common.io import read_parquet, write_parquet

```

```

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

```

```

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS = DATA / "models" / "ml2"

```

```

IN_UNIVERSE = GOLD / "pred_poscosecha_ml2_dh_grado_dia_bloque_destino_final.parquet"

```

```

OUT_FACTOR = EVAL / "backtest_factor_ml2_hidr_poscosecha.parquet"
OUT_FINAL = GOLD / "pred_poscosecha_ml2_hidr_grado_dia_bloque_destino_final.parquet"

```

```

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _as_of_date_today_minus_1() -> pd.Timestamp:
    return (pd.Timestamp.now().normalize() - pd.Timedelta(days=1)).normalize()

def _latest_model(prefix: str) -> tuple[Path, dict]:
    files = sorted(MODELS.glob(f"{prefix}_*.pkl"))
    if not files:
        raise FileNotFoundError(f"No hay modelos en {MODELS} con prefijo {prefix}_*.pkl")
    model_path = files[-1]
    meta_path = MODELS / (model_path.stem + "_meta.json")
    if not meta_path.exists():
        raise FileNotFoundError(f"No existe meta: {meta_path}")
    with open(meta_path, "r", encoding="utf-8") as f:
        meta = json.load(f)
    return model_path, meta

def _pick_first(df: pd.DataFrame, cands: list[str]) -> str | None:
    for c in cands:
        if c in df.columns:
            return c
    return None

def _resolve_fecha_post_pred(df: pd.DataFrame) -> str:
    c = _pick_first(df, ["fecha_post_pred_final", "fecha_post_pred_ml2", "fecha_post_pred_ml1", "fecha_post_pr
ed"])
    if c is None:
        raise KeyError("No encuentro fecha_post_pred. Espero una de: fecha_post_pred_final / fecha_post_pred_m
l2 / fecha_post_pred_ml1 / fecha_post_pred")
    return c

def _resolve_factor_hidr_ml1(df: pd.DataFrame) -> str:
    c = _pick_first(df, ["factor_hidr_ml1", "factor_hidr_pred_ml1", "factor_hidr"])
    if c is None:
        raise KeyError("No encuentro factor_hidr ML1 en universe. Espero: factor_hidr_ml1 / factor_hidr_pred_m
l1 / factor_hidr")
    return c

def main(mode: str = "backtest") -> None:
    as_of_date = _as_of_date_today_minus_1()
    created_at = pd.Timestamp.utcnow()

    model_path, meta = _latest_model("hidr_poscosecha_ml2")
    pipe = load(model_path)

    df = read_parquet(IN_UNIVERSE).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = {"fecha", "bloque_base", "variedad_canon", "grado", "destino"}
    miss = need - set(df.columns)
    if miss:
        raise KeyError(f"Universe sin columnas: {sorted(miss)}")

    fecha_post_col = _resolve_fecha_post_pred(df)
    hidr_ml1_col = _resolve_factor_hidr_ml1(df)

```



```

df["fecha"] = _to_date(df["fecha"])
df[fecha_post_col] = _to_date(df[fecha_post_col])
df["destino"] = _canon_str(df["destino"])
df["grado"] = _canon_int(df["grado"])
df["variedad_canon"] = _canon_str(df["variedad_canon"])
df["bloque_base"] = _canon_str(df["bloque_base"])

# filtro backtest <= as_of_date
df = df.loc[df["fecha"].notna() & (df["fecha"] <= as_of_date)].copy()

# features
df["dow"] = df[fecha_post_col].dt.dayofweek.astype("Int64")
df["month"] = df[fecha_post_col].dt.month.astype("Int64")
df["weekofyear"] = df[fecha_post_col].dt.isocalendar().week.astype("Int64")

# dh final (si existe)
if "dh_dias_final" not in df.columns:
    cdh = _pick_first(df, ["dh_dias_final", "dh_dias_ml2", "dh_dias_ml1", "dh_dias"])
    if cdh and cdh != "dh_dias_final":
        df = df.rename(columns={cdh: "dh_dias_final"})
if "dh_dias_final" not in df.columns:
    df["dh_dias_final"] = np.nan

df["factor_hidr_ml1"] = pd.to_numeric(df[hydr_ml1_col], errors="coerce")

X = df[["dow", "month", "weekofyear", "dh_dias_final", "destino", "grado", "variedad_canon"]].copy()
pred_log = pd.to_numeric(pd.Series(pipe.predict(X)), errors="coerce")

lo_log, hi_log = meta.get("apply_clip_log", [-1.2, 1.2])
pred_log_clip = pred_log.clip(lower=float(lo_log), upper=float(hi_log))

# factor multiplicativo ML2
df["factor_hidr_ml2_raw"] = np.exp(pred_log)
df["factor_hidr_ml2"] = np.exp(pred_log_clip)

lo_f, hi_f = meta.get("final_factor_clip", [0.60, 3.00])
df["factor_hidr_final"] = (
    df["factor_hidr_ml1"].fillna(1.0).astype(float) * df["factor_hidr_ml2"].astype(float)
).clip(lower=float(lo_f), upper=float(hi_f))

df["ml2_hidr_model_file"] = model_path.name
df["as_of_date"] = as_of_date
df["created_at"] = created_at

# factor output (ligero)
out_factor = df[[
    "fecha", "bloque_base", "variedad_canon", "grado", "destino",
    fecha_post_col, "factor_hidr_ml1", "factor_hidr_ml2_raw", "factor_hidr_ml2", "factor_hidr_final",
    "ml2_hidr_model_file", "as_of_date", "created_at"
]].copy().rename(columns={fecha_post_col: "fecha_post_pred_used"})

EVAL.mkdir(parents=True, exist_ok=True)
write_parquet(out_factor, OUT_FACTOR)
GOLD.mkdir(parents=True, exist_ok=True)
write_parquet(df, OUT_FINAL)

print(f"[OK] BACKTEST factor: {OUT_FACTOR} rows={len(out_factor):,}")
print(f"[OK] BACKTEST final : {OUT_FINAL} rows={len(df):,}")
print(f"      model={model_path.name} as_of_date={as_of_date.date()} final_clip={lo_f}->{hi_f}")

if __name__ == "__main__":
    import argparse

    ap = argparse.ArgumentParser()
    ap.add_argument("--mode", default="backtest", choices=["backtest"])
    args = ap.parse_args()

    main(mode=args.mode)

```

```
-----  
[106/164] FILE: \src\models\ml2\apply_peso_tallo_ml2.py  
-----
```

```
from __future__ import annotations
```

```
from pathlib import Path  
from datetime import datetime  
import argparse  
import json  
import numpy as np  
import pandas as pd
```

```
from src.common.io import read_parquet, write_parquet
```

```
def _project_root() -> Path:  
    return Path(__file__).resolve().parents[3]
```

```
ROOT = _project_root()  
DATA = ROOT / "data"  
GOLD = DATA / "gold"  
SILVER = DATA / "silver"  
MODELS = DATA / "models" / "ml2"  
EVAL = DATA / "eval" / "ml2"
```

```
# Dataset (ya consolidado con ML1 + real + clima)  
IN_DS = GOLD / "ml2_datasets" / "ds_peso_tallo_ml2_v1.parquet"
```

```
# Outputs PROD  
OUT_FACTOR_PROD = GOLD / "factors" / "factor_ml2_peso_tallo_grado_dia.parquet"  
OUT_FINAL_PROD = GOLD / "pred_peso_tallo_grado_dia_ml2_final.parquet"
```

```
# Outputs BACKTEST  
OUT_FACTOR_BT = EVAL / "backtest_factor_ml2_peso_tallo_grado_dia.parquet"  
OUT_FINAL_BT = EVAL / "backtest_pred_peso_tallo_grado_dia_ml2_final.parquet"
```

```
def _canon_str(s: pd.Series) -> pd.Series:  
    return s.astype(str).str.upper().str.strip()
```

```
def _load_latest_model() -> tuple[object, dict]:  
    metas = sorted(MODELS.glob("peso_tallo_ml2_*_meta.json"))  
    if not metas:  
        raise FileNotFoundError(f"No ML2 meta found in {MODELS}")  
    meta_path = metas[-1]  
    meta = json.loads(meta_path.read_text(encoding="utf-8"))  
    model_path = MODELS / f"peso_tallo_ml2_{meta['run_id']}.pkl"  
    import joblib  
    model = joblib.load(model_path)  
    return model, meta
```

```
def _parse_args() -> argparse.Namespace:  
    p = argparse.ArgumentParser()  
    p.add_argument("--mode", choices=["prod", "backtest"], default="backtest")  
    return p.parse_args()
```

```
def main() -> None:  
    args = _parse_args()  
    model, meta = _load_latest_model()  
  
    df = read_parquet(IN_DS).copy()  
  
    # Canon  
    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()  
    for c in ["bloque_base", "variedad_canon", "grado", "tipo_sp", "estado", "area"]:  
        if c in df.columns:  
            df[c] = _canon_str(df[c])
```

```

# Feature set from meta
for c in meta.get("num_cols", []):
    if c in df.columns:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)
for c in meta.get("cat_cols", []):
    if c in df.columns:
        df[c] = df[c].astype(str).fillna("")

feature_cols = meta["num_cols"] + meta["cat_cols"]
X = df[feature_cols].copy()

pred_log = model.predict(X).astype(float)
lo, hi = meta["guardrails"]["clip_log_error"]
pred_log = np.clip(pred_log, lo, hi)

df["pred_log_error_peso"] = pred_log
df["pred_ratio_peso"] = np.exp(df["pred_log_error_peso"])

# Final
df["peso_tallo_mll_g"] = pd.to_numeric(df["peso_tallo_mll_g"], errors="coerce")
df["peso_tallo_final_g"] = df["peso_tallo_mll_g"] * df["pred_ratio_peso"]

# Versioning
df["ml2_run_id"] = meta["run_id"]
df["created_at"] = pd.Timestamp(datetime.now()).normalize()

# Outputs: factor y final
cols_factor = [
    "ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado",
    "peso_tallo_mll_g", "peso_tallo_real_g",
    "pred_log_error_peso", "pred_ratio_peso", "peso_tallo_final_g",
    "ml2_run_id", "created_at",
]
cols_factor = [c for c in cols_factor if c in df.columns]
out_factor = df[cols_factor].copy()

cols_final = [
    "ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado",
    "peso_tallo_mll_g", "peso_tallo_final_g",
    "ml2_run_id", "created_at",
]
cols_final = [c for c in cols_final if c in df.columns]
out_final = df[cols_final].copy()

if args.mode == "prod":
    OUT_FACTOR_PROD.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(out_factor, OUT_FACTOR_PROD)
    write_parquet(out_final, OUT_FINAL_PROD)
    print(f"[OK] PROD factor: {OUT_FACTOR_PROD} rows={len(out_factor):,}")
    print(f"[OK] PROD final : {OUT_FINAL_PROD} rows={len(out_final):,}")
else:
    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(out_factor, OUT_FACTOR_BT)
    write_parquet(out_final, OUT_FINAL_BT)
    print(f"[OK] BACKTEST factor: {OUT_FACTOR_BT} rows={len(out_factor):,}")
    print(f"[OK] BACKTEST final : {OUT_FINAL_BT} rows={len(out_final):,}")

if __name__ == "__main__":
    main()

-----
[107/164] FILE: \src\models\ml2\apply_share_grado_ml2.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import argparse
import json
import numpy as np

```

```

import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS = DATA / "models" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_share_grado_ml2_v1.parquet"

# Tallos totales ya corregidos por ML2 curva diaria (backtest)
IN_TALLOS_TOTAL_BT = EVAL / "backtest_pred_tallos_dia_ml2_final.parquet"

# Outputs
OUT_FACTOR_BT = EVAL / "backtest_factor_ml2_share_grado.parquet"
OUT_FINAL_BT = EVAL / "backtest_pred_tallos_grado_dia_ml2_final.parquet"

OUT_FACTOR_PROD = GOLD / "factors" / "factor_ml2_share_grado.parquet"
OUT_FINAL_PROD = GOLD / "pred_tallos_grado_dia_ml2_final.parquet"

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _load_latest_model() -> tuple[object, dict]:
    metas = sorted(MODELS.glob("share_grado_ml2_*_meta.json"))
    if not metas:
        raise FileNotFoundError(f"No ML2 meta found in {MODELS}")
    meta_path = metas[-1]
    meta = json.loads(meta_path.read_text(encoding="utf-8"))
    model_path = MODELS / f"share_grado_ml2_{meta['run_id']}.pkl"
    import joblib
    model = joblib.load(model_path)
    return model, meta

def _parse_args() -> argparse.Namespace:
    p = argparse.ArgumentParser()
    p.add_argument("--mode", choices=["prod", "backtest"], default="backtest")
    return p.parse_args()

def main() -> None:
    args = _parse_args()
    model, meta = _load_latest_model()

    df = read_parquet(IN_DS).copy()
    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    for c in ["bloque_base", "variedad_canon", "grado", "tipo_sp", "estado", "area"]:
        if c in df.columns:
            df[c] = _canon_str(df[c])

    # features from meta
    for c in meta.get("num_cols", []):
        if c in df.columns:
            df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)
    for c in meta.get("cat_cols", []):
        if c in df.columns:
            df[c] = df[c].astype(str).fillna("")

    X = df[meta["num_cols"] + meta["cat_cols"]].copy()

    pred_log = model.predict(X).astype(float)
    lo, hi = meta["guardrails"]["clip_log_error"]

```

```

pred_log = np.clip(pred_log, lo, hi)

df["pred_log_error_share"] = pred_log
df["pred_ratio_share"] = np.exp(df["pred_log_error_share"])

df["share_ml1"] = pd.to_numeric(df["share_grado_ml1"], errors="coerce").fillna(0.0)
df["share_raw"] = df["share_ml1"] * df["pred_ratio_share"]

# --- RENORMALIZACIÓN (clave) por ciclo_id + fecha ---
grp = ["ciclo_id", "fecha"]
denom = df.groupby(grp)["share_raw"].transform("sum")
df["share_final"] = np.where(denom > 0, df["share_raw"] / denom, df["share_ml1"])

# Tallos totales (backtest usa salida de ML2 curva)
if args.mode == "backtest":
    tallos_total = read_parquet(IN_TALLOS_TOTAL_BT).copy()
    tallos_total["fecha"] = pd.to_datetime(tallos_total["fecha"], errors="coerce").dt.normalize()
    if "variedad_canon" in tallos_total.columns:
        tallos_total["variedad_canon"] = _canon_str(tallos_total["variedad_canon"])
    tallos_total["bloque_base"] = _canon_str(tallos_total["bloque_base"])
    # busquemos columna de tallos total
    if "tallos_pred_final" in tallos_total.columns:
        tot_col = "tallos_pred_final"
    elif "tallos_final" in tallos_total.columns:
        tot_col = "tallos_final"
    elif "tallos_pred_ml2" in tallos_total.columns:
        tot_col = "tallos_pred_ml2"
    else:
        # fallback: toma primera que parezca tallos
        cand = [c for c in tallos_total.columns if "tallos" in c and "pred" in c]
        if not cand:
            raise KeyError("No encuentro columna de tallos total en backtest_pred_tallos_dia_ml2_final.parquet")
        tot_col = cand[0]

    base_cols = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
    opt_cols = ["area", "tipo_sp", "estado"]

    cols_keep = [c for c in base_cols if c in tallos_total.columns]
    cols_keep += [c for c in opt_cols if c in tallos_total.columns]
    cols_keep += [tot_col]

    tallos_total = tallos_total[cols_keep].copy()
    tallos_total = tallos_total.rename(columns={tot_col: "tallos_total_ml2"})

    df = df.merge(tallos_total, on=["ciclo_id", "fecha", "bloque_base", "variedad_canon"], how="left")
else:
    # PROD: usa tallos ML1 día (si aún no tienes tallos_curve_ml2 prod)
    # Si ya tienes pred_tallos_dia_ml2_final en gold, lo cambiamos luego.
    df["tallos_total_ml2"] = df["tallos_pred_ml1_grado_dia"].groupby([df["ciclo_id"], df["fecha"]]).transform("sum")

df["tallos_final_grado_dia"] = df["tallos_total_ml2"] * df["share_final"]

df["ml2_run_id"] = meta["run_id"]
df["created_at"] = pd.Timestamp(datetime.now()).normalize()

# Factor output
out_factor = df[[
    "ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado",
    "share_ml1", "share_raw", "share_final",
    "pred_log_error_share", "pred_ratio_share",
    "ml2_run_id", "created_at"
]].copy()

# Final output
out_final = df[[
    "ciclo_id", "fecha", "bloque_base", "variedad_canon", "grado",
    "tallos_total_ml2", "share_final", "tallos_final_grado_dia",
    "ml2_run_id", "created_at"
]].copy()

```

```

if args.mode == "backtest":
    EVAL.mkdir(parents=True, exist_ok=True)
    write_parquet(out_factor, OUT_FACTOR_BT)
    write_parquet(out_final, OUT_FINAL_BT)
    print(f"[OK] BACKTEST factor: {OUT_FACTOR_BT} rows={len(out_factor):,}")
    print(f"[OK] BACKTEST final : {OUT_FINAL_BT} rows={len(out_final):,}")
else:
    OUT_FACTOR_PROD.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(out_factor, OUT_FACTOR_PROD)
    write_parquet(out_final, OUT_FINAL_PROD)
    print(f"[OK] PROD factor: {OUT_FACTOR_PROD} rows={len(out_factor):,}")
    print(f"[OK] PROD final : {OUT_FINAL_PROD} rows={len(out_final):,}")

if __name__ == "__main__":
    main()

-----
[108/164] FILE: \src\models\ml2\apply_tallos_curve_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import argparse
import json
import numpy as np
import pandas as pd

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
SILVER_DIR = DATA_DIR / "silver"
GOLD_DIR = DATA_DIR / "gold"
MODELS_DIR = DATA_DIR / "models" / "ml2"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

# Inputs
IN_GRID_ML2 = GOLD_DIR / "universe_harvest_grid_ml2.parquet"
IN_PRED_ML1 = GOLD_DIR / "pred_tallos_grado_dia_ml1_full.parquet"
IN_REAL = SILVER_DIR / "fact_cosecha_real_grado_dia.parquet"
IN_CLIMA = SILVER_DIR / "dim_clima_bloque_dia.parquet"

# Outputs (PROD)
OUT_FACTOR_PROD = GOLD_DIR / "factors" / "factor_ml2_tallos_curve_dia.parquet"
OUT_FINAL_PROD = GOLD_DIR / "pred_tallos_dia_ml2_final.parquet"

# Outputs (BACKTEST)
OUT_FACTOR_BT = EVAL_DIR / "backtest_factor_ml2_tallos_curve_dia.parquet"
OUT_FINAL_BT = EVAL_DIR / "backtest_pred_tallos_dia_ml2_final.parquet"

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def _load_latest_model() -> tuple[object, dict]:
    metas = sorted(MODELS_DIR.glob("tallos_curve_ml2_*_meta.json"))
    if not metas:
        raise FileNotFoundError(f"No ML2 meta found in {MODELS_DIR}")

```

```

meta_path = metas[-1]
meta = json.loads(meta_path.read_text(encoding="utf-8"))
model_path = MODELS_DIR / f"tallos_curve_ml2_{meta['run_id']}.pkl"
import joblib
model = joblib.load(model_path)
return model, meta

def _parse_args() -> argparse.Namespace:
    p = argparse.ArgumentParser()
    p.add_argument("--mode", choices=["prod", "backtest"], default="backtest")
    return p.parse_args()

def _resolve_block_base(df: pd.DataFrame) -> pd.DataFrame:
    if "bloque_base" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_base"])
        return df
    if "bloque_padre" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_padre"])
        return df
    if "bloque" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque"])
        return df
    raise KeyError("No encuentro columna de bloque: bloque_base / bloque_padre / bloque")

def main() -> None:
    args = _parse_args()
    model, meta = _load_latest_model()

    grid = read_parquet(IN_GRID_ML2).copy()
    pred = read_parquet(IN_PRED_ML1).copy()
    clima = read_parquet(IN_CLIMA).copy()

    # --- GRID ---
    grid["fecha"] = _to_date(grid["fecha"])
    grid["bloque_base"] = _canon_str(grid["bloque_base"])
    if "variedad_canon" in grid.columns:
        grid["variedad_canon"] = _canon_str(grid["variedad_canon"])
    if "tipo_sp" in grid.columns:
        grid["tipo_sp"] = _canon_str(grid["tipo_sp"])
    if "estado" in grid.columns:
        grid["estado"] = _canon_str(grid["estado"])

    base_cols = [
        "ciclo_id", "fecha", "bloque_base", "variedad_canon", "area", "tipo_sp", "estado",
        "rel_pos_final", "day_in_harvest_final", "n_harvest_days_final",
        "harvest_start_final", "harvest_end_final",
        "dow", "month", "weekofyear",
        "mll_version",
    ]
    base_cols = [c for c in base_cols if c in grid.columns]
    df = grid[base_cols].drop_duplicates(["ciclo_id", "fecha"]).copy()

    # --- PRED ML1 tallos día (sum grades) ---
    pred["fecha"] = _to_date(pred["fecha"])
    pred["bloque_base"] = _canon_str(pred["bloque_base"])
    pred["variedad_canon"] = _canon_str(pred["variedad_canon"])

    # Canon predicted col
    if "tallos_pred_mll_grado_dia" in pred.columns:
        pred_col = "tallos_pred_mll_grado_dia"
    elif "tallos_pred_mll_dia" in pred.columns:
        pred_col = "tallos_pred_mll_dia"
    else:
        raise KeyError("No encuentro tallos_pred_mll_grado_dia ni tallos_pred_mll_dia en pred ML1.")

    pred_agg = (
        pred.groupby(["fecha", "bloque_base", "variedad_canon"], as_index=False)
        .agg(tallos_pred_mll_dia=(pred_col, "sum"))
    )

```

```

df = df.merge(pred_agg, on=["fecha", "bloque_base", "variedad_canon"], how="left")
df["tallos_pred_ml1_dia"] = pd.to_numeric(df["tallos_pred_ml1_dia"], errors="coerce").fillna(0.0)

# --- REAL (solo en backtest; para KPIs posteriores) ---
if args.mode == "backtest":
    real = read_parquet(IN_REAL).copy()
    real["fecha"] = _to_date(real["fecha"])
    real = _resolve_block_base(real)
    real_agg = (
        real.groupby(["fecha", "bloque_base"], as_index=False)
        .agg(tallos_real_dia=("tallos_real", "sum"))
    )
    df = df.merge(real_agg, on=["fecha", "bloque_base"], how="left")
    df["tallos_real_dia"] = pd.to_numeric(df["tallos_real_dia"], errors="coerce").fillna(0.0)
else:
    df["tallos_real_dia"] = 0.0 # placeholder

# --- CLIMA diario ---
clima["fecha"] = _to_date(clima["fecha"])
clima["bloque_base"] = _canon_str(clima["bloque_base"])
clima_cols = [
    "fecha", "bloque_base",
    "gdc_dia", "gdc_base",
    "rainfall_mm_dia", "horas_lluvia", "en_lluvia_dia",
    "temp_avg_dia", "solar_energy_j_m2_dia",
    "wind_speed_avg_dia", "wind_run_dia",
]
clima_cols = [c for c in clima_cols if c in clima.columns]
clima0 = clima[clima_cols].copy()
df = df.merge(clima0, on=["fecha", "bloque_base"], how="left")

# Evitar duplicadas por seguridad pyarrow
df = df.loc[:, ~df.columns.duplicated()].copy()

# --- Features cleaning as meta ---
for c in meta.get("num_cols", []):
    if c in df.columns:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)

for c in meta.get("cat_cols", []):
    if c in df.columns:
        df[c] = df[c].astype(str).fillna("")

feature_cols = meta["num_cols"] + meta["cat_cols"]
X = df[feature_cols].copy()

pred_log = model.predict(X).astype(float)

lo, hi = meta["guardrails"]["clip_log_error"]
pred_log = np.clip(pred_log, lo, hi)

df["pred_log_error"] = pred_log
df["pred_ratio"] = np.exp(df["pred_log_error"])

# Final prediction
df["tallos_final_ml2_dia"] = df["tallos_pred_ml1_dia"] * df["pred_ratio"]

# Versioning
df["ml2_run_id"] = meta["run_id"]
df["created_at"] = pd.Timestamp(datetime.now()).normalize()

cols_factor = [
    "ciclo_id", "fecha", "bloque_base", "variedad_canon",
    "tallos_pred_ml1_dia", "tallos_real_dia",
    "pred_log_error", "pred_ratio", "tallos_final_ml2_dia",
    "ml2_run_id", "created_at",
]
if "ml1_version" in df.columns:
    cols_factor.insert(-2, "ml1_version")

out_factor = df[cols_factor].copy()

```



```

cols_final = [
    "ciclo_id", "fecha", "bloque_base", "variedad_canon",
    "tallos_pred_ml1_dia", "tallos_final_ml2_dia",
    "ml2_run_id", "created_at",
]
if "ml1_version" in df.columns:
    cols_final.insert(-2, "ml1_version")

out_final = df[cols_final].copy()

if "ml1_version" not in df.columns:
    print("[WARN] ml1_version not found in universe grid; outputs will omit ml1_version.")

if args.mode == "prod":
    OUT_FACTOR_PROD.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(out_factor, OUT_FACTOR_PROD)
    write_parquet(out_final, OUT_FINAL_PROD)
    print(f"[OK] PROD factor: {OUT_FACTOR_PROD} rows={len(out_factor):,}")
    print(f"[OK] PROD final : {OUT_FINAL_PROD} rows={len(out_final):,}")
else:
    EVAL_DIR.mkdir(parents=True, exist_ok=True)
    write_parquet(out_factor, OUT_FACTOR_BT)
    write_parquet(out_final, OUT_FINAL_BT)
    print(f"[OK] BACKTEST factor: {OUT_FACTOR_BT} rows={len(out_factor):,}")
    print(f"[OK] BACKTEST final : {OUT_FINAL_BT} rows={len(out_final):,}")

if __name__ == "__main__":
    main()

-----
[109/164] FILE: \src\models\ml2\train_ajuste_poscosecha_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import hashlib

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import HistGradientBoostingRegressor

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
MODELS = DATA / "models" / "ml2"
EVAL = DATA / "eval" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_ajuste_poscosecha_ml2_v1.parquet"

def _hash8(s: str) -> str:
    return hashlib.shal(s.encode("utf-8")).hexdigest()[0:8]

def _safe_ohc() -> OneHotEncoder:
    # sklearn compatibility: sparse_output vs sparse

```

```

try:
    return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    return OneHotEncoder(handle_unknown="ignore", sparse=False)

def _wmae(y_true: np.ndarray, y_pred: np.ndarray, w: np.ndarray) -> float:
    y_true = np.asarray(y_true, dtype="float64")
    y_pred = np.asarray(y_pred, dtype="float64")
    w = np.asarray(w, dtype="float64")
    m = np.isfinite(y_true) & np.isfinite(y_pred) & np.isfinite(w) & (w >= 0)
    if not np.any(m):
        return float("nan")
    denom = float(np.sum(w[m]))
    if denom <= 0:
        return float("nan")
    return float(np.sum(np.abs(y_true[m] - y_pred[m]) * w[m]) / denom)

def main() -> None:
    df = read_parquet(IN_DS).copy()
    df.columns = [str(c).strip() for c in df.columns]

    # solo filas con real para entrenamiento
    m = df["factor_ajuste_real"].notna() & df["log_ratio_ajuste"].notna()
    d = df.loc[m].copy()

    if d.empty:
        raise ValueError("Dataset no tiene filas con real (factor_ajuste_real). No se puede entrenar.")

    # features
    num_cols = ["dow", "month", "weekofyear"]
    cat_cols = ["destino", "grado"]

    for c in num_cols:
        if c not in d.columns:
            d[c] = pd.NA
    for c in cat_cols:
        if c not in d.columns:
            d[c] = "UNKNOWN"

    X = d[num_cols + cat_cols].copy()
    y = pd.to_numeric(d["log_ratio_ajuste"], errors="coerce").astype("float64").values
    w = pd.to_numeric(d.get("w", 1.0), errors="coerce").fillna(0.0).astype("float64").values

    # split temporal simple: 80/20 por fecha_post_pred_used
    d["fecha_post_pred_used"] = pd.to_datetime(d["fecha_post_pred_used"], errors="coerce").dt.normalize()
    order = d["fecha_post_pred_used"].fillna(pd.Timestamp("1970-01-01")).values
    idx = np.argsort(order)

    cut = int(0.80 * len(d))
    tr_idx = idx[:cut]
    va_idx = idx[cut:]

    X_tr, X_va = X.iloc[tr_idx], X.iloc[va_idx]
    y_tr, y_va = y[tr_idx], y[va_idx]
    w_tr, w_va = w[tr_idx], w[va_idx]

    pre = ColumnTransformer(
        transformers=[
            ("num", "passthrough", num_cols),
            ("cat", "_safe_ohe()", cat_cols),
        ],
        remainder="drop",
    )

    model = HistGradientBoostingRegressor(
        loss="squared_error",
        max_depth=6,
        learning_rate=0.08,
        max_iter=250,
        random_state=42,
    )

```

```

)

pipe = Pipeline([("pre", pre), ("model", model)])
pipe.fit(X_tr, y_tr, model__sample_weight=w_tr)

pred_tr = pipe.predict(X_tr)
pred_va = pipe.predict(X_va)

mae_tr = _wmae(y_tr, pred_tr, w_tr)
mae_va = _wmae(y_va, pred_va, w_va)

# save
MODELS.mkdir(parents=True, exist_ok=True)
EVAL.mkdir(parents=True, exist_ok=True)

ts = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
tag = _hash8(f"{ts}|ajuste_poscosecha_ml2|{mae_va:.6f}")
model_name = f"ajuste_poscosecha_ml2_{ts}_{tag}.pkl"
meta_name = f"ajuste_poscosecha_ml2_{ts}_{tag}_meta.json"

model_path = MODELS / model_name
meta_path = MODELS / meta_name

dump(pipe, model_path)

meta = {
    "model_file": model_name,
    "created_at_utc": datetime.utcnow().isoformat(),
    "target": "log_ratio_ajuste = log(real/ml1) clipped",
    "clip_range_target": [-1.2, 1.2],
    "clip_range_apply": [0.50, 2.00],
    "features_num": num_cols,
    "features_cat": cat_cols,
    "n_train": int(len(tr_idx)),
    "n_val": int(len(va_idx)),
    "mae_log_train_w": float(mae_tr),
    "mae_log_val_w": float(mae_va),
}

with open(meta_path, "w", encoding="utf-8") as f:
    json.dump(meta, f, ensure_ascii=False, indent=2)

out_eval = pd.DataFrame([
    {
        "model_file": model_name,
        "mae_log_train_w": mae_tr,
        "mae_log_val_w": mae_va,
        "n_train": int(len(tr_idx)),
        "n_val": int(len(va_idx)),
        "created_at": pd.Timestamp.utcnow(),
    }
])
eval_path = EVAL / "ml2_ajuste_poscosecha_train_val.parquet"
write_parquet(out_eval, eval_path)

print(f"[OK] Model saved: {model_path}")
print(f"[OK] Meta saved: {meta_path}")
print(f"[OK] Eval saved: {eval_path}")
print(f"      MAE log train={mae_tr:.4f} val={mae_va:.4f}")

if __name__ == "__main__":
    main()

-----
[110/164] FILE: \src\models\ml2\train_desp_poscosecha_ml2.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import hashlib
import json

```

```

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import HistGradientBoostingRegressor

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS = DATA / "models" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_desp_poscosecha_ml2_v1.parquet"
OUT_EVAL = EVAL / "ml2_desp_poscosecha_train_val.parquet"

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino", "grado"]
TARGET_COL = "log_ratio_desp_clipped"
WEIGHT_COL = "tallos_w"

def _mae(a: np.ndarray, b: np.ndarray, w: np.ndarray | None = None) -> float:
    a = np.asarray(a, dtype=float)
    b = np.asarray(b, dtype=float)
    m = np.isfinite(a) & np.isfinite(b)
    if not np.any(m):
        return float("nan")
    err = np.abs(a[m] - b[m])
    if w is None:
        return float(np.mean(err))
    ww = np.asarray(w, dtype=float)[m]
    denom = float(np.sum(ww))
    if denom <= 0:
        return float(np.mean(err))
    return float(np.sum(err * ww) / denom)

def _fingerprint(df: pd.DataFrame) -> str:
    # hash liviano para versionado
    h = hashlib.sha256()
    h.update(str(df.shape).encode("utf-8"))
    h.update(str(df[TARGET_COL].dropna().describe().to_dict()).encode("utf-8"))
    return h.hexdigest()[:8]

def main() -> None:
    df = read_parquet(IN_DS).copy()
    df.columns = [str(c).strip() for c in df.columns]

    need = set(NUM_COLS + CAT_COLS + [TARGET_COL, WEIGHT_COL, "has_real", "fecha_post_pred_used"])
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"Dataset sin columnas: {sorted(miss)}")

    d = df.loc[df["has_real"].astype(bool)].copy()
    d["fecha_post_pred_used"] = pd.to_datetime(d["fecha_post_pred_used"], errors="coerce").dt.normalize()

    # Split temporal: últimos 30 días (por fecha_post) a validación
    max_date = d["fecha_post_pred_used"].max()
    if pd.isna(max_date):

```

```

        raise ValueError("No hay fecha_post_pred_used válida en dataset con real.")

cut = max_date - pd.Timedelta(days=30)

tr = d.loc[d["fecha_post_pred_used"] < cut].copy()
va = d.loc[d["fecha_post_pred_used"] >= cut].copy()

X_tr = tr[NUM_COLS + CAT_COLS].copy()
y_tr = pd.to_numeric(tr[TARGET_COL], errors="coerce").values
w_tr = pd.to_numeric(tr[WEIGHT_COL], errors="coerce").fillna(0.0).values

X_va = va[NUM_COLS + CAT_COLS].copy()
y_va = pd.to_numeric(va[TARGET_COL], errors="coerce").values
w_va = pd.to_numeric(va[WEIGHT_COL], errors="coerce").fillna(0.0).values

# Preprocess: OHE denso (evita error sparse->dense)
pre = ColumnTransformer(
    transformers=[
        ("cat", OneHotEncoder(handle_unknown="ignore", sparse_output=False), CAT_COLS),
        ("num", "passthrough", NUM_COLS),
    ],
    remainder="drop",
)

model = HistGradientBoostingRegressor(
    loss="squared_error",
    learning_rate=0.05,
    max_depth=6,
    max_iter=400,
    random_state=42,
)

pipe = Pipeline([("prep", pre), ("model", model)])

pipe.fit(X_tr, y_tr, model__sample_weight=w_tr)

pred_tr = pipe.predict(X_tr)
pred_va = pipe.predict(X_va)

mae_tr = _mae(y_tr, pred_tr, w_tr)
mae_va = _mae(y_va, pred_va, w_va)

# Persist
MODELS.mkdir(parents=True, exist_ok=True)
EVAL.mkdir(parents=True, exist_ok=True)

ts = datetime.now().strftime("%Y%m%d_%H%M%S")
fp = _fingerprint(d)
model_name = f"desp_poscosecha_ml2_{ts}_{fp}.pkl"
meta_name = f"desp_poscosecha_ml2_{ts}_{fp}_meta.json"

dump(pipe, MODELS / model_name)

meta = {
    "model_file": model_name,
    "dataset": str(IN_DS),
    "target": TARGET_COL,
    "clip_target": 1.2,
    "clip_factor_final": [0.05, 1.00],
    "split_cut_date": str(cut.date()),
    "mae_train": mae_tr,
    "mae_val": mae_va,
    "n_train": int(len(tr)),
    "n_val": int(len(va)),
    "created_at": ts,
}

with open(MODELS / meta_name, "w", encoding="utf-8") as f:
    json.dump(meta, f, ensure_ascii=False, indent=2)

out_eval = pd.DataFrame(
    [

```

```

        {
            "split": "train",
            "n": int(len(tr)),
            "mae": mae_tr,
            "cut_date": str(cut.date()),
            "created_at": pd.Timestamp.now().normalize(),
        },
        {
            "split": "val",
            "n": int(len(va)),
            "mae": mae_va,
            "cut_date": str(cut.date()),
            "created_at": pd.Timestamp.now().normalize(),
        },
    ],
)
write_parquet(out_eval, OUT_EVAL)

print(f"[OK] Model saved: {MODELS / model_name}")
print(f"[OK] Meta saved: {MODELS / meta_name}")
print(f"[OK] Eval saved: {OUT_EVAL}")
print(f"      MAE log train={mae_tr:.4f}  val={mae_va:.4f}")

if __name__ == "__main__":
    main()

-----
[111/164] FILE: \src\models\ml2\train_dh_poscosecha_ml2.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import hashlib

import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS = DATA / "models" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_dh_poscosecha_ml2_v1.parquet"

NUM_COLS = ["dow", "month", "weekofyear"]
CAT_COLS = ["destino", "grado"]

TARGET = "err_dh_days_clipped"
WEIGHT_COL = "tallos_real"

def _run_id(prefix: str = "dh_poscosecha_ml2") -> str:

```

```

ts = datetime.now().strftime("%Y%m%d_%H%M%S")
h = hashlib.md5(ts.encode("utf-8")).hexdigest()[:8]
return f"{prefix}_{ts}_{h}"

def _make_ohe_dense() -> OneHotEncoder:
    """
    sklearn >=1.2 usa sparse_output; sklearn <1.2 usa sparse.
    Forzamos salida densa para que HistGradientBoostingRegressor no falle.
    """
    try:
        return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        return OneHotEncoder(handle_unknown="ignore", sparse=False)

def main() -> None:
    df = read_parquet(IN_DS).copy()
    df.columns = [str(c).strip() for c in df.columns]

    # Train solo donde hay real
    m = df["dh_real"].notna() & df["dh_ml1"].notna() & df[TARGET].notna()
    tr = df.loc[m].copy()
    if tr.empty:
        raise ValueError("No hay filas con real para entrenar DH ML2.")

    tr["fecha"] = pd.to_datetime(tr["fecha"], errors="coerce").dt.normalize()
    tr = tr.sort_values("fecha")

    # Split temporal: últimas 8 semanas val
    cut = tr["fecha"].max() - pd.Timedelta(days=56)
    is_val = tr["fecha"] >= cut

    train_df = tr.loc[~is_val].copy()
    val_df = tr.loc[is_val].copy()

    # fallback si val es muy pequeño
    if len(val_df) < 1000:
        n = len(tr)
        n_val = max(int(0.2 * n), 1000 if n >= 5000 else int(0.2 * n))
        val_df = tr.tail(n_val).copy()
        train_df = tr.iloc[: max(n - n_val, 1)].copy()

    # asegurar features
    for c in NUM_COLS:
        if c not in tr.columns:
            train_df[c] = pd.NA
            val_df[c] = pd.NA
    for c in CAT_COLS:
        if c not in tr.columns:
            train_df[c] = "UNKNOWN"
            val_df[c] = "UNKNOWN"

    X_train = train_df[NUM_COLS + CAT_COLS]
    y_train = pd.to_numeric(train_df[TARGET], errors="coerce")

    w_train = pd.to_numeric(train_df.get(WEIGHT_COL), errors="coerce").fillna(0.0).values.astype(float)
    w_train = np.where(w_train <= 0, 1.0, w_train)

    X_val = val_df[NUM_COLS + CAT_COLS]
    y_val = pd.to_numeric(val_df[TARGET], errors="coerce")

    w_val = pd.to_numeric(val_df.get(WEIGHT_COL), errors="coerce").fillna(0.0).values.astype(float)
    w_val = np.where(w_val <= 0, 1.0, w_val)

    pre = ColumnTransformer(
        transformers=[
            ("num", Pipeline(steps=[
                ("imputer", SimpleImputer(strategy="median")),
            ]), NUM_COLS),
            ("cat", Pipeline(steps=[
                ("imputer", SimpleImputer(strategy="most_frequent")),
            ]), CAT_COLS)
        ]
    )

```

```

        ("ohe", _make_ohe_dense()), # <-- FIX: denso
    ]), CAT_COLS),
],
remainder="drop",
verbose_feature_names_out=False,
)

model = HistGradientBoostingRegressor(
    loss="absolute_error",
    max_depth=6,
    learning_rate=0.08,
    max_iter=300,
    random_state=42,
)

pipe = Pipeline(steps=[("pre", pre), ("model", model)])

pipe.fit(X_train, y_train, model__sample_weight=w_train)

pred_tr = pipe.predict(X_train)
pred_va = pipe.predict(X_val)

mae_tr = mean_absolute_error(y_train, pred_tr, sample_weight=w_train)
mae_va = mean_absolute_error(y_val, pred_va, sample_weight=w_val)

run_id = _run_id()

MODELS.mkdir(parents=True, exist_ok=True)
EVAL.mkdir(parents=True, exist_ok=True)

out_model = MODELS / f"{run_id}.pkl"
out_meta = MODELS / f"{run_id}_meta.json"
out_eval = EVAL / "ml2_dh_poscosecha_train_val.parquet"

dump(pipe, out_model)

clip_err = float(pd.to_numeric(df.get("clip_err_days", pd.Series([5.0])).iloc[0], errors="coerce") or 5.0)

meta = {
    "run_id": run_id,
    "created_at_utc": str(pd.Timestamp.utcnow()),
    "dataset": str(IN_DS),
    "target": TARGET,
    "clip_err_days": clip_err,
    "features_num": NUM_COLS,
    "features_cat": CAT_COLS,
    "split": {"type": "time", "val_window_days": 56},
    "metrics": {"mae_train_days": float(mae_tr), "mae_val_days": float(mae_va)},
    "note": "OHE forced dense for HistGradientBoostingRegressor compatibility",
}

with open(out_meta, "w", encoding="utf-8") as f:
    json.dump(meta, f, ensure_ascii=False, indent=2)

ev = pd.DataFrame([
    {
        "run_id": run_id,
        "n_train": int(len(train_df)),
        "n_val": int(len(val_df)),
        "mae_train_days": float(mae_tr),
        "mae_val_days": float(mae_va),
        "created_at": pd.Timestamp(datetime.now()).normalize(),
    }
])
write_parquet(ev, out_eval)

print(f"[OK] Model saved: {out_model}")
print(f"[OK] Meta saved: {out_meta}")
print(f"[OK] Eval saved: {out_eval}")
print(f"      MAE days train={mae_tr:.4f}   val={mae_va:.4f}")

if __name__ == "__main__":
    main()

```



```

-----
[112/164] FILE: \src\models\ml2\train_harvest_horizon_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import uuid

import numpy as np
import pandas as pd

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
GOLD_DIR = DATA_DIR / "gold"
MODELS_DIR = DATA_DIR / "models" / "ml2"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_DS = GOLD_DIR / "ml2_datasets" / "ds_harvest_horizon_ml2_v2.parquet"

# Guardrails
CLIP_ERROR_LO = -14
CLIP_ERROR_HI = 21

# Split temporal
VAL_QUANTILE = 0.80 # últimos 20% as_of_date como validación

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def main() -> None:
    df = read_parquet(IN_DS).copy()

    # Canon
    if "variedad_canon" in df.columns:
        df["variedad_canon"] = _canon_str(df["variedad_canon"])
    if "bloque_base" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_base"])
    if "tipo_sp" in df.columns:
        df["tipo_sp"] = _canon_str(df["tipo_sp"])
    if "estado" in df.columns:
        df["estado"] = _canon_str(df["estado"])

    df["as_of_date"] = pd.to_datetime(df["as_of_date"], errors="coerce").dt.normalize()

    # Target
    y = pd.to_numeric(df["error_horizon_days"], errors="coerce")
    m = y.notna() & df["as_of_date"].notna()
    df = df.loc[m, :].copy()
    y = y.loc[m].astype(float)

    # Features: num + cat
    num_cols = [
        "tallos_proy",

```

```

    "days_sp_to_start_pred",
    "n_harvest_days_pred",
    "days_from_sp",
    "pred_error_start_days",
    "dow",
    "month",
    "weekofyear",
    "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
    "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
    "solar_cum_sp", "solar_7d",
    "temp_avg_7d",
]
cat_cols = [
    "bloque_base",
    "variedad_canon",
    "area",
    "tipo_sp",
    "estado",
]

# asegurar columnas existentes
num_cols = [c for c in num_cols if c in df.columns]
cat_cols = [c for c in cat_cols if c in df.columns]

for c in num_cols:
    df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)
for c in cat_cols:
    df[c] = df[c].astype(str).fillna("")

X = df[num_cols + cat_cols].copy()

# Split temporal por as_of_date
cutoff = df["as_of_date"].quantile(VAL_QUANTILE)
is_val = df["as_of_date"] >= cutoff

Xtr, ytr = X.loc[~is_val], y.loc[~is_val]
Xva, yva = X.loc[is_val], y.loc[is_val]

# OHE denso (compatible sklearn vieja)
try:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

pre = ColumnTransformer(
    transformers=[
        ("num", "passthrough", num_cols),
        ("cat", ohe, cat_cols),
    ],
    remainder="drop",
)

model = HistGradientBoostingRegressor(
    loss="absolute_error",
    max_depth=6,
    learning_rate=0.05,
    max_iter=300,
    random_state=42,
)

pipe = Pipeline([("pre", pre), ("model", model)])

pipe.fit(Xtr, ytr)

# Metrics
pred_tr = pipe.predict(Xtr)
pred_va = pipe.predict(Xva)

mae_tr = float(mean_absolute_error(ytr, pred_tr))
mae_va = float(mean_absolute_error(yva, pred_va))

# Guardrail metrics (solo diagnóstico; el clip se aplica en APPLY)

```

```

pred_va_clip = np.clip(pred_va, CLIP_ERROR_LO, CLIP_ERROR_HI)
pct_clip_va = float(np.mean((pred_va <= CLIP_ERROR_LO) | (pred_va >= CLIP_ERROR_HI))) if len(pred_va) else
np.nan
mae_va_clip = float(mean_absolute_error(yva, pred_va_clip)) if len(pred_va) else np.nan

run_id = datetime.now().strftime("%Y%m%d_%H%M%S") + "_" + uuid.uuid4().hex[:8]

MODELS_DIR.mkdir(parents=True, exist_ok=True)
EVAL_DIR.mkdir(parents=True, exist_ok=True)

# Save model
import joblib
model_path = MODELS_DIR / f"harvest_horizon_ml2_{run_id}.pkl"
joblib.dump(pipe, model_path)

# Save meta
meta = {
    "run_id": run_id,
    "created_at": datetime.now().isoformat(timespec="seconds"),
    "dataset": str(IN_DS),
    "target": "error_horizon_days = n_harvest_days_real - n_harvest_days_pred_ml1",
    "guardrails": {"clip_error_days": [CLIP_ERROR_LO, CLIP_ERROR_HI], "min_horizon_days": 7},
    "split": {"type": "temporal_quantile", "val_quantile": VAL_QUANTILE, "cutoff_as_of_date": str(cutoff.d
ate())},
    "num_cols": num_cols,
    "cat_cols": cat_cols,
    "metrics": {
        "mae_train": mae_tr,
        "mae_val": mae_va,
        "mae_val_clipped": mae_va_clip,
        "pct_clip_val": pct_clip_va,
        "n_train": int(len(ytr)),
        "n_val": int(len(yva)),
    },
}
meta_path = MODELS_DIR / f"harvest_horizon_ml2_{run_id}_meta.json"
meta_path.write_text(json.dumps(meta, indent=2), encoding="utf-8")

# Save eval row
eval_row = pd.DataFrame([
    "run_id": run_id,
    "cutoff_as_of_date": pd.to_datetime(cutoff).normalize(),
    "n_train": int(len(ytr)),
    "n_val": int(len(yva)),
    "mae_train": mae_tr,
    "mae_val": mae_va,
    "mae_val_clipped": mae_va_clip,
    "pct_clip_val": pct_clip_va,
    "created_at": pd.Timestamp(datetime.now()).normalize(),
])
eval_path = EVAL_DIR / "ml2_harvest_horizon_train_cv.parquet"
# append-safe: si existe, concatenar
if eval_path.exists():
    old = read_parquet(eval_path)
    eval_row = pd.concat([old, eval_row], ignore_index=True)
write_parquet(eval_row, eval_path)

print(f"[OK] Model saved: {model_path}")
print(f"[OK] Meta saved: {meta_path}")
print(f"[OK] Eval saved: {eval_path}")
print(f"MAE train={mae_tr:.4f} val={mae_va:.4f} val_clipped={mae_va_clip:.4f} pct_clip_val={pct_cl
ip_va:.3%}")

if __name__ == "__main__":
    main()

```

```

-----
[113/164] FILE: \src\models\ml2\train_harvest_start_ml2.py
-----
# src/models/ml2/train_harvest_start_ml2.py

```

```

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import uuid
import numpy as np
import pandas as pd

from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_absolute_error
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
GOLD_DIR = DATA_DIR / "gold"
MODELS_DIR = DATA_DIR / "models" / "ml2"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

DS_PATH = GOLD_DIR / "ml2_datasets" / "ds_harvest_start_ml2_v2.parquet"
OUT_KPIS = EVAL_DIR / "ml2_harvest_start_train_cv.parquet"

TARGET = "error_start_days"

NUM_COLS = [
    "days_sp_to_start_pred",
    "n_harvest_days_pred",
    "tallos_proy",
    "dow", "month", "weekofyear",
    "gdc_cum_sp", "gdc_7d", "gdc_14d", "gdc_per_day",
    "rain_cum_sp", "rain_7d", "enlluvia_days_7d",
    "solar_cum_sp", "solar_7d",
    "temp_avg_7d",
]
CAT_COLS = [
    "bloque_base",
    "variedad_canon",
    "tipo_sp",
    "area",
    "estado",
]
CLIP_LO, CLIP_HI = -21, 21

def main() -> None:
    df = read_parquet(DS_PATH).copy()

    num_cols = [c for c in NUM_COLS if c in df.columns]
    cat_cols = [c for c in CAT_COLS if c in df.columns]

    df = df.sort_values("as_of_date").reset_index(drop=True)

    X = df[num_cols + cat_cols]
    y = df[TARGET].astype(float)

    # Preprocess
    try:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        # sklearn older
        ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

```

```

pre = ColumnTransformer(
    transformers=[
        ("num", "passthrough", num_cols),
        ("cat", ohe, cat_cols),
    ],
    remainder="drop",
)

model = HistGradientBoostingRegressor(
    loss="absolute_error",
    max_depth=6,
    learning_rate=0.05,
    max_iter=400,
    random_state=42,
)

pipe = Pipeline(steps=[("pre", pre), ("model", model)])

tscv = TimeSeriesSplit(n_splits=5)
maes = []
for fold, (tr, te) in enumerate(tscv.split(X), start=1):
    Xtr, Xte = X.iloc[tr], X.iloc[te]
    ytr, yte = y.iloc[tr], y.iloc[te]
    pipe.fit(Xtr, ytr)
    pred = pipe.predict(Xte)
    mae = float(mean_absolute_error(yte, pred))
    maes.append({"fold": fold, "mae_days": mae, "n_test": int(len(te))})

pipe.fit(X, y)

run_id = datetime.now().strftime("%Y%m%d_%H%M%S") + "_" + uuid.uuid4().hex[:8]
MODELS_DIR.mkdir(parents=True, exist_ok=True)

model_path = MODELS_DIR / f"harvest_start_ml2_{run_id}.pkl"
meta_path = MODELS_DIR / f"harvest_start_ml2_{run_id}_meta.json"

import joblib
joblib.dump(pipe, model_path)

meta = {
    "run_id": run_id,
    "trained_at": datetime.now().isoformat(),
    "dataset": str(DS_PATH),
    "target": TARGET,
    "num_cols": num_cols,
    "cat_cols": cat_cols,
    "cv": maes,
    "cv_mae_mean": float(np.mean([m["mae_days"] for m in maes])) if maes else None,
    "guardrails": {"clip_error_days": [CLIP_LO, CLIP_HI]},
}
meta_path.write_text(json.dumps(meta, indent=2), encoding="utf-8")

# Guardar CV
EVAL_DIR.mkdir(parents=True, exist_ok=True)
kpi_df = pd.DataFrame(maes)
kpi_df["run_id"] = run_id
kpi_df["created_at"] = pd.Timestamp(datetime.now()).normalize()
write_parquet(kpi_df, OUT_KPIS)

print(f"[OK] Model saved: {model_path}")
print(f"[OK] Meta saved : {meta_path}")
print(f"[OK] CV KPIs saved: {OUT_KPIS}")

if __name__ == "__main__":
    main()

```

```

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import numpy as np
import pandas as pd
from joblib import dump

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
EVAL = DATA / "eval" / "ml2"
MODELS = DATA / "models" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_hidr_poscosecha_ml2_v1.parquet"

def _ohe_dense() -> OneHotEncoder:
    # compat sklearn
    try:
        return OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        return OneHotEncoder(handle_unknown="ignore", sparse=False)

def main() -> None:
    df = read_parquet(IN_DS).copy()
    df.columns = [str(c).strip() for c in df.columns]

    # solo filas con real
    m = df.get("is_in_real", pd.Series(False, index=df.index)).astype(bool)
    d = df.loc[m].copy()
    if d.empty:
        raise ValueError("DS no tiene filas con real (is_in_real=1). No se puede entrenar.")

    # features
    num_cols = ["dow", "month", "weekofyear", "dh_dias_final"]
    # dh_dias_final podría no existir en dataset; intenta alias
    if "dh_dias_final" not in d.columns:
        for c in ["dh_dias_ml2", "dh_dias_ml1", "dh_dias"]:
            if c in d.columns:
                d = d.rename(columns={c: "dh_dias_final"})
                break
    if "dh_dias_final" not in d.columns:
        d["dh_dias_final"] = np.nan

    cat_cols = ["destino", "grado", "variedad_canon"]

    for c in num_cols:
        if c not in d.columns:
            d[c] = np.nan
    for c in cat_cols:
        if c not in d.columns:
            d[c] = "UNKNOWN"

    X = d[num_cols + cat_cols].copy()
    y = pd.to_numeric(d["log_error_hidr_clipped"], errors="coerce")

    # pesos por tallos (si no hay, 1)

```

```

w = pd.to_numeric(d.get("tallos_w"), errors="coerce").fillna(1.0).clip(lower=0.0)

# split simple por tiempo (más estable que random)
# tomamos las últimas ~20% fechas_post_pred como "val" si existe, si no random
split_col = None
for c in ["fecha_post_pred_final", "fecha_post_pred_ml2", "fecha_post_pred_ml1", "fecha_post_pred"]:
    if c in d.columns:
        split_col = c
        break

if split_col is not None:
    t = pd.to_datetime(d[split_col], errors="coerce").dt.normalize()
    q = t.quantile(0.80)
    train_idx = t <= q
else:
    rng = np.random.default_rng(42)
    train_idx = rng.random(len(d)) <= 0.80

X_train, y_train, w_train = X.loc[train_idx], y.loc[train_idx], w.loc[train_idx]
X_val, y_val, w_val = X.loc[~train_idx], y.loc[~train_idx], w.loc[~train_idx]

pre = ColumnTransformer(
    transformers=[
        ("num", "passthrough", num_cols),
        ("cat", _ohe_dense(), cat_cols),
    ],
    remainder="drop",
)

model = HistGradientBoostingRegressor(
    loss="squared_error",
    max_depth=6,
    learning_rate=0.06,
    max_iter=350,
    min_samples_leaf=40,
    l2_regularization=0.0,
    random_state=42,
)

pipe = Pipeline([("pre", pre), ("model", model)])

pipe.fit(X_train, y_train, model__sample_weight=w_train)

# eval en log space
pred_tr = pipe.predict(X_train)
pred_va = pipe.predict(X_val)

def _mae(y_true, y_pred, w=None) -> float:
    a = pd.to_numeric(pd.Series(y_true), errors="coerce")
    b = pd.to_numeric(pd.Series(y_pred), errors="coerce")
    m = a.notna() & b.notna()
    if not m.any():
        return float("nan")

    err = (a[m] - b[m]).abs()

    if w is None:
        return float(err.mean())

    ww = pd.to_numeric(pd.Series(w), errors="coerce").fillna(0.0)
    ww = ww[m].clip(lower=0.0)
    denom = float(ww.sum())
    if denom <= 0:
        return float(err.mean())
    return float((err * ww).sum() / denom)

mae_tr = _mae(y_train, pred_tr, w_train)
mae_va = _mae(y_val, pred_va, w_val)

# artefactos
ts = datetime.now().strftime("%Y%m%d_%H%M%S")
uid = f"{np.random.default_rng().integers(0, 16**8):08x}"

```

```

model_name = f"hidr_poscosecha_ml2_{ts}_{uid}.pkl"
meta_name = f"hidr_poscosecha_ml2_{ts}_{uid}_meta.json"

MODELS.mkdir(parents=True, exist_ok=True)
EVAL.mkdir(parents=True, exist_ok=True)

dump(pipe, MODELS / model_name)

meta = {
    "model_file": model_name,
    "created_at_utc": pd.Timestamp.utcnow().isoformat(),
    "target": "log_error_hidr_clipped",
    "apply_clip_log": [-1.2, 1.2],
    "final_factor_clip": [0.60, 3.00],
    "features_num": num_cols,
    "features_cat": cat_cols,
    "mae_log_train_w": mae_tr,
    "mae_log_val_w": mae_va,
    "n_train": int(len(X_train)),
    "n_val": int(len(X_val)),
}
with open(MODELS / meta_name, "w", encoding="utf-8") as f:
    json.dump(meta, f, ensure_ascii=False, indent=2)

# guardar tabla simple train/val
out = pd.DataFrame([
    "mae_log_train_w": mae_tr,
    "mae_log_val_w": mae_va,
    "n_train": int(len(X_train)),
    "n_val": int(len(X_val)),
    "model_file": model_name,
    "meta_file": meta_name,
    "created_at": pd.Timestamp.now().normalize(),
])
write_parquet(out, EVAL / "ml2_hidr_poscosecha_train_val.parquet")

print(f"[OK] Model saved: {MODELS / model_name}")
print(f"[OK] Meta saved: {MODELS / meta_name}")
print(f"[OK] Eval saved: {EVAL / 'ml2_hidr_poscosecha_train_val.parquet'}")
print(f"      MAE log train(w)={mae_tr:.4f}   val(w)={mae_va:.4f}")

```

```

if __name__ == "__main__":
    main()

```

```

-----
[115/164] FILE: \src\models\ml2\train_peso_tallo_ml2.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import uuid

import numpy as np
import pandas as pd

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

```



```

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
MODELS = DATA / "models" / "ml2"
EVAL = DATA / "eval" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_peso_tallo_ml2_v1.parquet"

CLIP_LO, CLIP_HI = -0.8, 0.8
VAL_QUANTILE = 0.80

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def main() -> None:
    df = read_parquet(IN_DS).copy()

    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    for c in ["bloque_base", "variedad_canon", "grado", "tipo_sp", "estado", "area"]:
        if c in df.columns:
            df[c] = _canon_str(df[c])

    y = pd.to_numeric(df["log_error_peso"], errors="coerce")
    m = y.notna() & df["fecha"].notna()
    df = df.loc[m].copy()
    y = y.loc[m].astype(float).clip(CLIP_LO, CLIP_HI)

    # Features (num + cat)
    num_cols = [
        "rel_pos_final", "day_in_harvest_final", "n_harvest_days_final",
        "peso_tallo_ml1_g", # escala base
        # clima diario
        "gdc_dia", "rainfall_mm_dia", "en_lluvia_dia",
        "temp_avg_dia", "solar_energy_j_m2_dia",
        "wind_speed_avg_dia", "wind_run_dia",
        # calendario
        "dow", "month", "weekofyear",
    ]
    cat_cols = [
        "bloque_base", "variedad_canon", "grado",
        "area", "tipo_sp", "estado",
    ]

    num_cols = [c for c in num_cols if c in df.columns]
    cat_cols = [c for c in cat_cols if c in df.columns]

    for c in num_cols:
        df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)
    for c in cat_cols:
        df[c] = df[c].astype(str).fillna("")

    X = df[num_cols + cat_cols].copy()

    cutoff = df["fecha"].quantile(VAL_QUANTILE)
    is_val = df["fecha"] >= cutoff

    Xtr, ytr = X.loc[~is_val], y.loc[~is_val]
    Xva, yva = X.loc[is_val], y.loc[is_val]

    try:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
    except TypeError:
        ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

    pre = ColumnTransformer(
        transformers=[
            ("num", "passthrough", num_cols),
            ("cat", ohe, cat_cols),
        ],
        remainder="drop",

```

```

)

model = HistGradientBoostingRegressor(
    loss="absolute_error",
    max_depth=6,
    learning_rate=0.05,
    max_iter=350,
    random_state=42,
)

pipe = Pipeline([("pre", pre), ("model", model)])
pipe.fit(Xtr, ytr)

pred_tr = pipe.predict(Xtr)
pred_va = pipe.predict(Xva)

mae_tr = float(mean_absolute_error(ytr, pred_tr))
mae_va = float(mean_absolute_error(yva, pred_va))

pred_va_clip = np.clip(pred_va, CLIP_LO, CLIP_HI)
mae_va_clip = float(mean_absolute_error(yva, pred_va_clip))
pct_clip_va = float(np.mean((pred_va <= CLIP_LO) | (pred_va >= CLIP_HI))) if len(pred_va) else np.nan

run_id = datetime.now().strftime("%Y%m%d_%H%M%S") + "_" + uuid.uuid4().hex[:8]

MODELS.mkdir(parents=True, exist_ok=True)
EVAL.mkdir(parents=True, exist_ok=True)

import joblib
model_path = MODELS / f"peso_tallo_ml2_{run_id}.pkl"
joblib.dump(pipe, model_path)

meta = {
    "run_id": run_id,
    "created_at": datetime.now().isoformat(timespec="seconds"),
    "dataset": str(IN_DS),
    "target": "log_error_peso = log((peso_tallo_real_g+eps)/(peso_tallo_ml1_g+eps))",
    "guardrails": {"clip_log_error": [CLIP_LO, CLIP_HI]},
    "split": {"type": "temporal_quantile", "val_quantile": VAL_QUANTILE, "cutoff_fecha": str(pd.to_datetime(cutoff).date())},
    "num_cols": num_cols,
    "cat_cols": cat_cols,
    "metrics": {
        "mae_train_log": mae_tr,
        "mae_val_log": mae_va,
        "mae_val_log_clipped": mae_va_clip,
        "pct_clip_val": pct_clip_va,
        "n_train": int(len(ytr)),
        "n_val": int(len(yva)),
    },
},
}

meta_path = MODELS / f"peso_tallo_ml2_{run_id}_meta.json"
meta_path.write_text(json.dumps(meta, indent=2), encoding="utf-8")

eval_row = pd.DataFrame([
    "run_id": run_id,
    "cutoff_fecha": pd.to_datetime(cutoff).normalize(),
    "n_train": int(len(ytr)),
    "n_val": int(len(yva)),
    "mae_train_log": mae_tr,
    "mae_val_log": mae_va,
    "mae_val_log_clipped": mae_va_clip,
    "pct_clip_val": pct_clip_va,
    "created_at": pd.Timestamp(datetime.now()).normalize(),
])

eval_path = EVAL / "ml2_peso_tallo_train_cv.parquet"
if eval_path.exists():
    old = read_parquet(eval_path)
    eval_row = pd.concat([old, eval_row], ignore_index=True)
write_parquet(eval_row, eval_path)

print(f"[OK] Model saved: {model_path}")

```

```

    print(f"[OK] Meta   saved: {meta_path}")
    print(f"[OK] Eval   saved: {eval_path}")
    print(f"        MAE log train={mae_tr:.4f}   val={mae_va:.4f}   val_clipped={mae_va_clip:.4f}   pct_clip_val={pct_clip_val:.3%}")

if __name__ == "__main__":
    main()

-----
[116/164] FILE: \src\models\ml2\train_share_grado_ml2.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import json
import uuid

import numpy as np
import pandas as pd

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA = ROOT / "data"
GOLD = DATA / "gold"
MODELS = DATA / "models" / "ml2"
EVAL = DATA / "eval" / "ml2"

IN_DS = GOLD / "ml2_datasets" / "ds_share_grado_ml2_v1.parquet"

CLIP_LO, CLIP_HI = -1.2, 1.2
VAL_QUANTILE = 0.80

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def main() -> None:
    df = read_parquet(IN_DS).copy()

    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    for c in ["bloque_base", "variedad_canon", "grado", "tipo_sp", "estado", "area"]:
        if c in df.columns:
            df[c] = _canon_str(df[c])

    y = pd.to_numeric(df["log_error_share"], errors="coerce")
    m = y.notna() & df["fecha"].notna()
    df = df.loc[m].copy()
    y = y.loc[m].astype(float).clip(CLIP_LO, CLIP_HI)

    # Features
    num_cols = [
        "rel_pos_final", "day_in_harvest_final", "n_harvest_days_final",
        # contexto de volumen (importante para shares)
        "tallos_pred_ml1_grado_dia",
        "share_grado_ml1",
        # clima

```

```

    "gdc_dia", "rainfall_mm_dia", "en_lluvia_dia",
    "temp_avg_dia", "solar_energy_j_m2_dia",
    "wind_speed_avg_dia", "wind_run_dia",
    # calendario
    "dow", "month", "weekofyear",
]
cat_cols = [
    "grado",
    "variedad_canon",
    "area", "tipo_sp", "estado",
    # bloque_base puede overfit fuerte; lo dejamos pero si quieres lo quitamos después
    "bloque_base",
]

num_cols = [c for c in num_cols if c in df.columns]
cat_cols = [c for c in cat_cols if c in df.columns]

for c in num_cols:
    df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)
for c in cat_cols:
    df[c] = df[c].astype(str).fillna("")

X = df[num_cols + cat_cols].copy()

cutoff = df["fecha"].quantile(VAL_QUANTILE)
is_val = df["fecha"] >= cutoff

Xtr, ytr = X.loc[~is_val], y.loc[~is_val]
Xva, yva = X.loc[is_val], y.loc[is_val]

try:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

pre = ColumnTransformer(
    transformers=[
        ("num", "passthrough", num_cols),
        ("cat", ohe, cat_cols),
    ],
    remainder="drop",
)

model = HistGradientBoostingRegressor(
    loss="absolute_error",
    max_depth=6,
    learning_rate=0.05,
    max_iter=350,
    random_state=42,
)

pipe = Pipeline([("pre", pre), ("model", model)])
pipe.fit(Xtr, ytr)

pred_tr = pipe.predict(Xtr)
pred_va = pipe.predict(Xva)

mae_tr = float(mean_absolute_error(ytr, pred_tr))
mae_va = float(mean_absolute_error(yva, pred_va))

pred_va_clip = np.clip(pred_va, CLIP_LO, CLIP_HI)
mae_va_clip = float(mean_absolute_error(yva, pred_va_clip))
pct_clip_va = float(np.mean((pred_va <= CLIP_LO) | (pred_va >= CLIP_HI))) if len(pred_va) else np.nan

run_id = datetime.now().strftime("%Y%m%d-%H%M%S") + "_" + uuid.uuid4().hex[:8]

MODELS.mkdir(parents=True, exist_ok=True)
EVAL.mkdir(parents=True, exist_ok=True)

import joblib
model_path = MODELS / f"share_grado_ml2_{run_id}.pkl"
joblib.dump(pipe, model_path)

```

```

meta = {
    "run_id": run_id,
    "created_at": datetime.now().isoformat(timespec="seconds"),
    "dataset": str(IN_DS),
    "target": "log_error_share = log((share_real+eps)/(share_mll+eps))",
    "guardrails": {"clip_log_error": [CLIP_LO, CLIP_HI]},
    "split": {
        "type": "temporal_quantile",
        "val_quantile": VAL_QUANTILE,
        "cutoff_fecha": str(pd.to_datetime(cutoff).date()),
    },
    "num_cols": num_cols,
    "cat_cols": cat_cols,
    "metrics": {
        "mae_train_log": mae_tr,
        "mae_val_log": mae_va,
        "mae_val_log_clipped": mae_va_clip,
        "pct_clip_val": pct_clip_va,
        "n_train": int(len(ytr)),
        "n_val": int(len(yva)),
    },
}
}
meta_path = MODELS / f"share_grado_ml2_{run_id}_meta.json"
meta_path.write_text(json.dumps(meta, indent=2), encoding="utf-8")

eval_row = pd.DataFrame([
    "run_id": run_id,
    "cutoff_fecha": pd.to_datetime(cutoff).normalize(),
    "n_train": int(len(ytr)),
    "n_val": int(len(yva)),
    "mae_train_log": mae_tr,
    "mae_val_log": mae_va,
    "mae_val_log_clipped": mae_va_clip,
    "pct_clip_val": pct_clip_va,
    "created_at": pd.Timestamp(datetime.now()).normalize(),
])
eval_path = EVAL / "ml2_share_grado_train_cv.parquet"
if eval_path.exists():
    old = read_parquet(eval_path)
    eval_row = pd.concat([old, eval_row], ignore_index=True)
write_parquet(eval_row, eval_path)

print(f"[OK] Model saved: {model_path}")
print(f"[OK] Meta saved: {meta_path}")
print(f"[OK] Eval saved: {eval_path}")
print(f"    MAE log train={mae_tr:.4f}    val={mae_va:.4f}    val_clipped={mae_va_clip:.4f}    pct_clip_val={pct_clip_va:.3%}")

```

```

if __name__ == "__main__":
    main()

```

```

-----
[117/164] FILE: \src\models\ml2\train_tallos_curve_ml2.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import json
import uuid

```

```

import numpy as np
import pandas as pd

```

```

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.metrics import mean_absolute_error

```

```

from src.common.io import read_parquet, write_parquet

def _project_root() -> Path:
    return Path(__file__).resolve().parents[3]

ROOT = _project_root()
DATA_DIR = ROOT / "data"
GOLD_DIR = DATA_DIR / "gold"
MODELS_DIR = DATA_DIR / "models" / "ml2"
EVAL_DIR = DATA_DIR / "eval" / "ml2"

IN_DS = GOLD_DIR / "ml2_datasets" / "ds_tallos_curve_ml2_v2.parquet"

# Guardrails (mismo rango del dataset builder)
CLIP_LOG_ERR_LO = -1.5
CLIP_LOG_ERR_HI = 1.5

VAL_QUANTILE = 0.80 # último 20% fechas para validación

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

def main() -> None:
    df = read_parquet(IN_DS).copy()

    # Canon + fecha
    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    if "bloque_base" in df.columns:
        df["bloque_base"] = _canon_str(df["bloque_base"])
    if "variedad_canon" in df.columns:
        df["variedad_canon"] = _canon_str(df["variedad_canon"])
    if "tipo_sp" in df.columns:
        df["tipo_sp"] = _canon_str(df["tipo_sp"])
    if "estado" in df.columns:
        df["estado"] = _canon_str(df["estado"])

    # Target
    y = pd.to_numeric(df["log_error"], errors="coerce")
    m = y.notna() & df["fecha"].notna()
    df = df.loc[m, :].copy()
    y = y.loc[m].astype(float).clip(CLIP_LOG_ERR_LO, CLIP_LOG_ERR_HI)

    # Features (num + cat)
    num_cols = [
        "rel_pos_final",
        "day_in_harvest_final",
        "n_harvest_days_final",
        "tallos_pred_ml1_dia",
        "tallos_real_dia",
        # clima día
        "gdc_dia",
        "rainfall_mm_dia",
        "en_lluvia_dia",
        "temp_avg_dia",
        "solar_energy_j_m2_dia",
        "wind_speed_avg_dia",
        "wind_run_dia",
        # calendario
        "dow", "month", "weekofyear",
    ]
    cat_cols = [
        "bloque_base",
        "variedad_canon",
        "area",
        "tipo_sp",
        "estado",
    ]

```

```

num_cols = [c for c in num_cols if c in df.columns]
cat_cols = [c for c in cat_cols if c in df.columns]

# Limpieza
for c in num_cols:
    df[c] = pd.to_numeric(df[c], errors="coerce").fillna(0.0)
for c in cat_cols:
    df[c] = df[c].astype(str).fillna("")

X = df[num_cols + cat_cols].copy()

# Split temporal por fecha
cutoff = df["fecha"].quantile(VAL_QUANTILE)
is_val = df["fecha"] >= cutoff

Xtr, ytr = X.loc[~is_val], y.loc[~is_val]
Xva, yva = X.loc[is_val], y.loc[is_val]

# OHE denso
try:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

pre = ColumnTransformer(
    transformers=[
        ("num", "passthrough", num_cols),
        ("cat", ohe, cat_cols),
    ],
    remainder="drop",
)

model = HistGradientBoostingRegressor(
    loss="absolute_error",
    max_depth=6,
    learning_rate=0.05,
    max_iter=350,
    random_state=42,
)

pipe = Pipeline([("pre", pre), ("model", model)])
pipe.fit(Xtr, ytr)

# Métricas sobre log_error
pred_tr = pipe.predict(Xtr)
pred_va = pipe.predict(Xva)

mae_tr = float(mean_absolute_error(ytr, pred_tr))
mae_va = float(mean_absolute_error(yva, pred_va))

# Diagnóstico: % clipping si se aplica (aún no lo aplicamos aquí)
pred_va_clip = np.clip(pred_va, CLIP_LOG_ERR_LO, CLIP_LOG_ERR_HI)
pct_clip_va = float(np.mean((pred_va <= CLIP_LOG_ERR_LO) | (pred_va >= CLIP_LOG_ERR_HI))) if len(pred_va)
else np.nan
mae_va_clip = float(mean_absolute_error(yva, pred_va_clip)) if len(pred_va) else np.nan

run_id = datetime.now().strftime("%Y%m%d_%H%M%S") + "_" + uuid.uuid4().hex[:8]

MODELS_DIR.mkdir(parents=True, exist_ok=True)
EVAL_DIR.mkdir(parents=True, exist_ok=True)

# Save model
import joblib
model_path = MODELS_DIR / f"tallos_curve_ml2_{run_id}.pkl"
joblib.dump(pipe, model_path)

# Save meta
meta = {
    "run_id": run_id,
    "created_at": datetime.now().isoformat(timespec="seconds"),
    "dataset": str(IN_DS),

```

```

        "target": "log_error = log((tallos_real_dia+eps)/(tallos_pred_mll_dia+eps))",
        "guardrails": {"clip_log_error": [CLIP_LOG_ERR_LO, CLIP_LOG_ERR_HI]},
        "split": {"type": "temporal_quantile", "val_quantile": VAL_QUANTILE, "cutoff_fecha": str(pd.to_datetime(cutoff).date())},
        "num_cols": num_cols,
        "cat_cols": cat_cols,
        "metrics": {
            "mae_train_log": mae_tr,
            "mae_val_log": mae_va,
            "mae_val_log_clipped": mae_va_clip,
            "pct_clip_val": pct_clip_va,
            "n_train": int(len(ytr)),
            "n_val": int(len(yva)),
        },
    },
}
meta_path = MODELS_DIR / f"tallos_curve_ml2_{run_id}_meta.json"
meta_path.write_text(json.dumps(meta, indent=2), encoding="utf-8")

# Save eval row
eval_row = pd.DataFrame([
    "run_id": run_id,
    "cutoff_fecha": pd.to_datetime(cutoff).normalize(),
    "n_train": int(len(ytr)),
    "n_val": int(len(yva)),
    "mae_train_log": mae_tr,
    "mae_val_log": mae_va,
    "mae_val_log_clipped": mae_va_clip,
    "pct_clip_val": pct_clip_va,
    "created_at": pd.Timestamp(datetime.now()).normalize(),
])
eval_path = EVAL_DIR / "ml2_tallos_curve_train_cv.parquet"
if eval_path.exists():
    old = read_parquet(eval_path)
    eval_row = pd.concat([old, eval_row], ignore_index=True)
write_parquet(eval_row, eval_path)

print(f"[OK] Model saved: {model_path}")
print(f"[OK] Meta saved: {meta_path}")
print(f"[OK] Eval saved: {eval_path}")
print(f"      MAE log train={mae_tr:.4f}  val={mae_va:.4f}  val_clipped={mae_va_clip:.4f}  pct_clip_val={pct_clip_va:.3%}")

```

```

if __name__ == "__main__":
    main()

```

```

-----
[118/164] FILE: \src\ops\io.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
import yaml

```

```

def load_settings(path: str = "config/settings.yaml") -> dict:
    p = Path(path)
    if not p.exists():
        raise FileNotFoundError(f"settings not found: {p.resolve()}")
    with p.open("r", encoding="utf-8") as f:
        return yaml.safe_load(f) or {}

```

```

-----
[119/164] FILE: \src\ops\registry.py
-----

```

```

from __future__ import annotations

```

```

from ops.runner import Step

```



```

def build_registry() -> list[Step]:
    """
    Registry lineal (orden de ejecución) que asegura que los .py se corran sin fallar
    por dependencias faltantes.

    Nota importante:
    - Si un step escribe SIEMPRE sobre el mismo output que otros steps,
      aquí lo dejamos con outputs_rel=[] para que NO se ?salte? por existencia previa.
    """

    steps: list[Step] = []

    # =====
    # BRONZE
    # =====
    steps += [
        Step(
            name="balanza_cosecha_raw",
            layer="bronze",
            script_relpath="src/bronze/build_balanza_cosecha_raw.py",
            outputs_rel=["bronze/balanza_cosecha_raw.parquet"],
        ),
        Step(
            name="balanza_lc_raw",
            layer="bronze",
            script_relpath="src/bronze/build_balanza_lc_raw.py",
            outputs_rel=["bronze/balanza_lc_raw.parquet"],
        ),
        Step(
            name="balanza_mermas_sources",
            layer="bronze",
            script_relpath="src/bronze/build_balanza_mermas_sources.py",
            outputs_rel=[
                "bronze/balanza_2_raw.parquet",
                "bronze/balanza_2a_raw.parquet",
            ],
        ),
        Step(
            name="fenograma_sources",
            layer="bronze",
            script_relpath="src/bronze/build_fenograma_sources.py",
            outputs_rel=[
                "bronze/balanza_bloque_fecha_raw.parquet",
                "bronze/fenograma_xlsm_raw.parquet",
                "bronze/indices_clo_raw.parquet",
                "bronze/indices_xl_raw.parquet",
            ],
        ),
        Step(
            name="ghu_maestro_horas",
            layer="bronze",
            script_relpath="src/bronze/build_ghu_maestro_horas.py",
            outputs_rel=["bronze/ghu_maestro_horas.parquet"],
        ),
        Step(
            name="personal_sources",
            layer="bronze",
            script_relpath="src/bronze/build_personal_sources.py",
            outputs_rel=["bronze/personal_raw.parquet"],
        ),
        Step(
            name="ventas_sources",
            layer="bronze",
            script_relpath="src/bronze/build_ventas_sources.py",
            outputs_rel=[
                "bronze/ventas_2025_raw.parquet",
                "bronze/ventas_2026_raw.parquet",
            ],
        ),
        Step(
            name="weather_hour_main",
            layer="bronze",

```

```

        script_relpath="src/bronze/build_weather_hour_main.py",
        outputs_rel=["bronze/weather_hour_main.parquet"],
    ),
    Step(
        name="weather_hour_a4",
        layer="bronze",
        script_relpath="src/bronze/build_weather_hour_a4.py",
        outputs_rel=["bronze/weather_hour_a4.parquet"],
    ),
]

# =====
# SILVER (base)
# =====
steps += [
    Step(
        name="weather_hour_estado",
        layer="silver",
        script_relpath="src/silver/build_weather_hour_estado.py",
        outputs_rel=["silver/weather_hour_estado.parquet"],
    ),
    Step(
        name="weather_hour_wide",
        layer="silver",
        script_relpath="src/silver/build_weather_hour_wide.py",
        outputs_rel=["silver/weather_hour_wide.parquet"],
    ),
    Step(
        name="ciclo_maestro_from_sources",
        layer="silver",
        script_relpath="src/silver/build_ciclo_maestro_from_sources.py",
        outputs_rel=[
            "silver/fact_ciclo_maestro.parquet",
            "silver/grid_ciclo_fecha.parquet",
        ],
    ),
    Step(
        name="hidratacion_real_from_balanza2",
        layer="silver",
        script_relpath="src/silver/build_hidratacion_real_from_balanza2.py",
        outputs_rel=[
            "silver/fact_hidratacion_real_post_grado_destino.parquet",
            "silver/dim_hidratacion_dh_grado_destino.parquet",
            "silver/dim_hidratacion_fecha_post_grado_destino.parquet",
        ],
    ),
    Step(
        name="fact_cosecha_real_grado_dia",
        layer="silver",
        script_relpath="src/silver/build_fact_cosecha_real_grado_dia.py",
        outputs_rel=["silver/fact_cosecha_real_grado_dia.parquet"],
    ),
    Step(
        name="fact_peso_tallo_real_grado_dia",
        layer="silver",
        script_relpath="src/silver/build_fact_peso_tallo_real_grado_dia.py",
        outputs_rel=["silver/fact_peso_tallo_real_grado_dia.parquet"],
    ),
    Step(
        name="fact_cosecha_uph_hora_clima",
        layer="silver",
        script_relpath="src/silver/build_fact_cosecha_uph_hora_clima.py",
        outputs_rel=["silver/fact_cosecha_uph_hora_clima.parquet"],
    ),
    Step(
        name="dim_dist_grado_baseline",
        layer="silver",
        script_relpath="src/silver/build_dim_dist_grado_baseline.py",
        outputs_rel=["silver/dim_dist_grado_baseline.parquet"],
    ),
    Step(
        name="dim_peso_tallo_baseline",

```

```

        layer="silver",
        script_relpath="src/silver/build_dim_peso_tallo_baseline.py",
        outputs_rel=["silver/dim_peso_tallo_baseline.parquet"],
    ),
    Step(
        name="dim_peso_tallo_promedio_dia",
        layer="silver",
        script_relpath="src/silver/build_dim_peso_tallo_promedio_dia.py",
        outputs_rel=["silver/dim_peso_tallo_promedio_dia.parquet"],
    ),
    Step(
        name="dim_factor_uph_cosecha_clima",
        layer="silver",
        script_relpath="src/silver/build_dim_factor_uph_cosecha_clima.py",
        outputs_rel=["silver/dim_factor_uph_cosecha_clima.parquet"],
    ),
    Step(
        name="dim_mermas_ajuste_fecha_post",
        layer="silver",
        script_relpath="src/silver/build_dim_mermas_ajuste_fecha_post.py",
        outputs_rel=["silver/dim_mermas_ajuste_fecha_post_destino.parquet"],
    ),
    Step(
        name="dim_dh_baseline_grado_destino",
        layer="silver",
        script_relpath="src/silver/build_dim_dh_baseline_grado_destino.py",
        outputs_rel=["silver/dim_dh_baseline_grado_destino.parquet"],
    ),
    Step(
        name="dim_hidratacion_fecha_cosecha_grado_destino",
        layer="silver",
        script_relpath="src/silver/build_dim_hidratacion_fecha_cosecha_grado_destino.py",
        outputs_rel=["silver/dim_hidratacion_fecha_cosecha_grado_destino.parquet"],
    ),
    Step(
        name="dim_hidratacion_baseline_grado_destino",
        layer="silver",
        script_relpath="src/silver/build_dim_hidratacion_baseline_grado_destino.py",
        outputs_rel=["silver/dim_hidratacion_baseline_grado_destino.parquet"],
    ),
    Step(
        name="dim_variedad_canon",
        layer="silver",
        script_relpath="src/silver/build_dim_variedad_canon.py",
        outputs_rel=["silver/dim_variedad_canon.parquet"],
    ),
    Step(
        name="dim_clima_bloque_dia",
        layer="silver",
        script_relpath="src/silver/build_dim_clima_bloque_dia.py",
        outputs_rel=["silver/dim_clima_bloque_dia.parquet"],
    ),
    Step(
        name="dim_estado_termico_cultivo_bloque_fecha",
        layer="silver",
        script_relpath="src/silver/build_dim_estado_termico_cultivo_bloque_fecha.py",
        outputs_rel=["silver/dim_estado_termico_cultivo_bloque_fecha.parquet"],
    ),
    Step(
        name="dim_mix_proceso_semana_from_ventas",
        layer="silver",
        script_relpath="src/silver/build_dim_mix_proceso_semana_from_ventas.py",
        outputs_rel=["silver/dim_mix_proceso_semana.parquet"],
    ),
    Step(
        name="fact_capacidad_proceso_hist",
        layer="silver",
        script_relpath="src/silver/build_fact_capacidad_proceso_hist.py",
        outputs_rel=[
            "silver/fact_capacidad_proceso_hist.parquet",
            "silver/dim_capacidad_baseline_proceso.parquet",
        ],
    ),

```

```

    ),
    Step(
        name="dim_baseline_capacidad_tallos_h_persona",
        layer="silver",
        script_relpath="src/silver/build_dim_baseline_capacidad_tallos_h_persona.py",
        outputs_rel=["silver/dim_baseline_capacidad_tallos_h_persona.parquet"],
    ),
    Step(
        name="dim_capacidad_baseline_tallos_proceso",
        layer="silver",
        script_relpath="src/silver/build_dim_capacidad_baseline_tallos_proceso.py",
        outputs_rel=[
            "silver/dim_capacidad_baseline_tallos_proceso.parquet",
            "silver/fact_capacidad_tallos_proceso_hist.parquet",
        ],
    ),
    Step(
        name="milestones_windows",
        layer="silver",
        script_relpath="src/silver/build_milestones_windows.py",
        outputs_rel=[
            "silver/fact_milestones_ciclo.parquet",
            "silver/milestone_window_ciclo.parquet",
        ],
    ),
]

# =====
# PREDS
# =====
steps += [
    Step(
        name="pred_milestones_baseline",
        layer="preds",
        script_relpath="src/preds/build_pred_milestones_baseline.py",
        outputs_rel=["preds/pred_milestones_ciclo.parquet"],
    ),
    Step(
        name="milestones_final",
        layer="preds",
        script_relpath="src/silver/build_milestones_final.py",
        outputs_rel=["silver/milestones_ciclo_final.parquet"],
    ),
    Step(
        name="dim_mediana_etapas_tipo_sp_variedad_area",
        layer="preds",
        script_relpath="src/silver/build_dim_mediana_etapas_tipo_sp_variedad_area.py",
        outputs_rel=["silver/dim_mediana_etapas_tipo_sp_variedad_area.parquet"],
    ),
    Step(
        name="milestone_window_ciclo_final_with_inference",
        layer="preds",
        script_relpath="src/silver/build_milestone_window_ciclo_final_with_inference.py",
        outputs_rel=["silver/milestone_window_ciclo_final.parquet"],
    ),
    Step(
        name="dim_cosecha_progress_bloque_fecha",
        layer="preds",
        script_relpath="src/silver/build_dim_cosecha_progress_bloque_fecha.py",
        outputs_rel=["silver/dim_cosecha_progress_bloque_fecha.parquet"],
    ),
    Step(
        name="pred_oferta_dia",
        layer="preds",
        script_relpath="src/preds/build_pred_oferta_dia.py",
        outputs_rel=["preds/pred_oferta_dia.parquet"],
    ),
    Step(
        name="pred_oferta_grado",
        layer="preds",
        script_relpath="src/preds/build_pred_oferta_grado.py",
        outputs_rel=["preds/pred_oferta_grado.parquet"],
    ),

```

```

    ),
    Step(
        name="pred_peso_grado",
        layer="preds",
        script_relpath="src/preds/build_pred_peso_grado.py",
        outputs_rel=["preds/pred_peso_grado.parquet"],
    ),
    Step(
        name="pred_peso_hidratado_grado",
        layer="preds",
        script_relpath="src/preds/build_pred_peso_hidratado_grado.py",
        outputs_rel=["preds/pred_peso_hidratado_grado.parquet"],
    ),
    Step(
        name="pred_peso_final_ajustado_grado",
        layer="preds",
        script_relpath="src/preds/build_pred_peso_final_ajustado_grado.py",
        outputs_rel=["preds/pred_peso_final_ajustado_grado.parquet"],
    ),
    Step(
        name="pred_cajas_from_peso_final",
        layer="preds",
        script_relpath="src/preds/build_pred_cajas_from_peso_final.py",
        outputs_rel=[
            "preds/pred_cajas_grado.parquet",
            "preds/pred_cajas_dia.parquet",
        ],
    ),
    Step(
        name="pred_tallos_cosecha_dia",
        layer="preds",
        script_relpath="src/preds/build_pred_tallos_cosecha_dia.py",
        outputs_rel=["preds/pred_tallos_cosecha_dia.parquet"],
    ),
    Step(
        name="capacidad_cosecha_dia",
        layer="preds",
        script_relpath="src/preds/build_capacidad_cosecha_dia.py",
        outputs_rel=["preds/capacidad_cosecha_dia.parquet"],
    ),
    Step(
        name="pred_horas_poscosecha",
        layer="preds",
        script_relpath="src/preds/build_pred_horas_poscosecha.py",
        outputs_rel=["preds/pred_horas_poscosecha_dia.parquet"],
    ),
    Step(
        name="pred_plan_horas_dia",
        layer="preds",
        script_relpath="src/preds/build_pred_plan_horas_dia.py",
        outputs_rel=["preds/pred_plan_horas_dia.parquet"],
    ),
    Step(
        name="patch_ciclo_maestro_from_pred_tallos",
        layer="preds",
        script_relpath="src/silver/build_patch_ciclo_maestro_from_pred_tallos.py",
        outputs_rel=["silver/fact_ciclo_maestro_patch_pred_tallos_report.parquet"],
    ),
]

# =====
# FEATURES (macro)
# =====
steps += [
    Step(
        name="features_ciclo_fecha",
        layer="features",
        script_relpath="src/features/build_features_ciclo_fecha.py",
        outputs_rel=["features/features_ciclo_fecha.parquet"],
    ),
]

```

```

# =====
# MODELS (ML1)
# =====
steps += [
    Step(
        name="features_harvest_window_ml1",
        layer="models",
        script_relpath="src/features/build_features_harvest_window_ml1.py",
        outputs_rel=["features/features_harvest_window_ml1.parquet"],
    ),
    Step(
        name="apply_harvest_window_ml1",
        layer="models",
        script_relpath="src/models/ml1/apply_harvest_window_ml1.py",
        outputs_rel=["gold/pred_harvest_window_ml1.parquet"],
    ),
    Step(
        name="universe_harvest_grid_ml1",
        layer="models",
        script_relpath="src/gold/build_universe_harvest_grid_ml1.py",
        outputs_rel=["gold/universe_harvest_grid_ml1.parquet"],
    ),
    Step(
        name="features_curva_cosecha_bloque_dia",
        layer="models",
        script_relpath="src/features/build_features_curva_cosecha_bloque_dia.py",
        outputs_rel=["features/features_curva_cosecha_bloque_dia.parquet"],
    ),
    Step(
        name="cap_tallos_real_dia",
        layer="models",
        script_relpath="src/features/build_cap_tallos_real_dia.py",
        outputs_rel=["gold/dim_cap_tallos_real_dia.parquet"],
    ),
    Step(
        name="targets_curva_beta_params",
        layer="models",
        script_relpath="src/features/build_targets_curva_beta_params.py",
        outputs_rel=["features/trainset_curva_beta_params.parquet"],
    ),
    Step(
        name="targets_curva_beta_multiplier_dia",
        layer="models",
        script_relpath="src/features/build_targets_curva_beta_multiplier_dia.py",
        outputs_rel=["features/trainset_curva_beta_multiplier_dia.parquet"],
    ),
    Step(
        name="features_cosecha_bloque_fecha",
        layer="models",
        script_relpath="src/features/build_features_cosecha_bloque_fecha.py",
        outputs_rel=["features/features_cosecha_bloque_fecha.parquet"],
    ),
    Step(
        name="apply_dist_grado_ml1",
        layer="models",
        script_relpath="src/models/ml1/apply_dist_grado.py",
        outputs_rel=["gold/pred_dist_grado_ml1.parquet"],
    ),
    Step(
        name="features_peso_tallo_grado_bloque_dia",
        layer="models",
        script_relpath="src/features/build_features_peso_tallo_grado_bloque_dia.py",
        outputs_rel=["features/features_peso_tallo_grado_bloque_dia.parquet"],
    ),
    Step(
        name="apply_peso_tallo_grado_ml1",
        layer="models",
        script_relpath="src/models/ml1/apply_peso_tallo_grado.py",
        outputs_rel=["gold/pred_peso_tallo_grado_ml1.parquet"],
    ),
    Step(
        name="apply_curva_beta_params",

```

```

        layer="models",
        script_relpath="src/models/ml1/apply_curva_beta_params.py",
        outputs_rel=[], # no skip
    ),
    # (los otros de curva los dejaste comentados, ok)
]

# =====
# GOLD (ML1 finales)
# =====
steps += [
    Step(
        name="pred_tallos_grado_dia_ml1",
        layer="gold",
        script_relpath="src/gold/build_pred_tallos_grado_dia_ml1.py",
        outputs_rel=["gold/pred_tallos_grado_dia_ml1.parquet"],
    ),
    Step(
        name="pred_tallos_grado_dia_ml1_full",
        layer="gold",
        script_relpath="src/gold/build_pred_tallos_grado_dia_ml1_full.py",
        outputs_rel=["gold/pred_tallos_grado_dia_ml1_full.parquet"],
    ),
    Step(
        name="pred_kg_cajas_grado_dia_ml1_full",
        layer="gold",
        script_relpath="src/gold/build_pred_kg_cajas_grado_dia_ml1_full.py",
        outputs_rel=[
            "gold/pred_kg_grado_dia_ml1_full.parquet",
            "gold/pred_kg_grado_dia_ml1_agg.parquet",
            "gold/pred_cajas_grado_dia_ml1_full.parquet",
            "gold/pred_cajas_grado_dia_ml1_agg.parquet",
            "gold/pred_cajas_dia_ml1_full.parquet",
        ],
    ),
    Step(
        name="pred_cajas_postcosecha_seed_mix_grado_dia",
        layer="gold",
        script_relpath="src/gold/build_pred_cajas_postcosecha_seed_mix_grado_dia.py",
        outputs_rel=[
            "gold/pred_poscosecha_seed_grado_dia_bloque_destino.parquet",
            "gold/pred_poscosecha_seed_dia_destino.parquet",
            "gold/pred_poscosecha_seed_dia_total.parquet",
        ],
    ),
    Step(
        name="apply_dh_poscosecha_ml1",
        layer="gold",
        script_relpath="src/models/ml1/apply_dh_poscosecha_ml1.py",
        outputs_rel=["gold/pred_poscosecha_ml1_dh_grado_dia_bloque_destino.parquet"],
    ),
    Step(
        name="apply_hidr_poscosecha_ml1",
        layer="gold",
        script_relpath="src/models/ml1/apply_hidr_poscosecha_ml1.py",
        outputs_rel=["gold/pred_poscosecha_ml1_hidr_grado_dia_bloque_destino.parquet"],
    ),
    Step(
        name="apply_desp_ajuste_poscosecha_ml1",
        layer="gold",
        script_relpath="src/models/ml1/apply_desp_ajuste_poscosecha_ml1.py",
        outputs_rel=["gold/pred_poscosecha_ml1_full_grado_dia_bloque_destino.parquet"],
    ),
    Step(
        name="pred_poscosecha_ml1_views",
        layer="gold",
        script_relpath="src/gold/build_pred_poscosecha_ml1_views.py",
        outputs_rel=[
            "gold/pred_poscosecha_ml1_dia_bloque_destino.parquet",
            "gold/pred_poscosecha_ml1_dia_destino.parquet",
            "gold/pred_poscosecha_ml1_dia_total.parquet",
        ],
    ),

```

```

    ),
    Step(
        name="view_planificacion_campo_tallos",
        layer="gold",
        script_relpath="src/gold/build_view_planificacion_campo_tallos.py",
        outputs_rel=[
            "gold/view_planificacion_campo_tallos_dia.parquet",
            "gold/view_planificacion_campo_tallos_semana.parquet",
            "gold/view_planificacion_campo_tallos_semana_area.parquet",
            "gold/view_planificacion_campo_tallos_semana_bloque.parquet",
        ],
    ),
]

# =====
# ML2 (correcciones + outputs ML2)
# =====
steps += [
    # ---- harvest corrections ----
    Step(
        name="apply_harvest_start_ml2",
        layer="ml2",
        script_relpath="src/models/ml2/apply_harvest_start_ml2.py",
        outputs_rel=[], # recomendado: corre siempre
        args=["--mode", "prod"], # si tu script soporta prod
    ),
    Step(
        name="apply_harvest_horizon_ml2",
        layer="ml2",
        script_relpath="src/models/ml2/apply_harvest_horizon_ml2.py",
        outputs_rel=[],
        args=["--mode", "prod"],
    ),

    # ---- universe ML2 (si tienes este script) ----
    Step(
        name="universe_harvest_grid_ml2",
        layer="ml2",
        script_relpath="src/gold/build_universe_harvest_grid_ml2.py",
        outputs_rel=["gold/universe_harvest_grid_ml2.parquet"],
    ),

    # ---- tallos curve (si lo usas) ----
    Step(
        name="apply_tallos_curve_ml2",
        layer="ml2",
        script_relpath="src/models/ml2/apply_tallos_curve_ml2.py",
        outputs_rel=[],
        args=["--mode", "prod"],
    ),

    # ---- share grado ----
    Step(
        name="build_ds_share_grado_ml2_v1",
        layer="ml2",
        script_relpath="src/gold/build_ds_share_grado_ml2_v1.py",
        outputs_rel=[], # si quieres skip, pon aquí el output real del ds
    ),
    Step(
        name="apply_share_grado_ml2",
        layer="ml2",
        script_relpath="src/models/ml2/apply_share_grado_ml2.py",
        outputs_rel=[], # recomendado: corre siempre
        args=["--mode", "prod"],
    ),

    # ---- peso tallo ----
    Step(
        name="build_ds_peso_tallo_ml2_v1",
        layer="ml2",
        script_relpath="src/gold/build_ds_peso_tallo_ml2_v1.py",
        outputs_rel=[],
    ),

```



```

),
Step(
    name="apply_peso_tallo_ml2",
    layer="ml2",
    script_relpath="src/models/ml2/apply_peso_tallo_ml2.py",
    outputs_rel=[],
    args=["--mode", "prod"],
),

# ---- kg/cajas usando outputs ML2 ----
Step(
    name="pred_kg_cajas_grado_dia_ml2_full",
    layer="ml2",
    script_relpath="src/gold/build_pred_kg_cajas_grado_dia_ml2_full.py",
    outputs_rel=[
        "gold/pred_kg_grado_dia_ml2_full.parquet",
        "gold/pred_kg_grado_dia_ml2_agg.parquet",
        "gold/pred_cajas_grado_dia_ml2_full.parquet",
        "gold/pred_cajas_grado_dia_ml2_agg.parquet",
        "gold/pred_cajas_dia_ml2_full.parquet",
    ],
),

# ---- POSCOSECHA ML2 (recomendado: ML1-on-ML2) ----
Step(
    name="apply_dh_poscosecha_ml1_on_ml2_seed",
    layer="ml2",
    script_relpath="src/gold/apply_dh_poscosecha_ml1_on_ml2_seed.py",
    outputs_rel=["gold/pred_poscosecha_ml2_dh_grado_dia_bloque_destino.parquet"],
),
Step(
    name="apply_hidr_poscosecha_ml1_on_ml2_dh",
    layer="ml2",
    script_relpath="src/gold/apply_hidr_poscosecha_ml1_on_ml2_dh.py",
    outputs_rel=["gold/pred_poscosecha_ml2_hidr_grado_dia_bloque_destino.parquet"],
),
Step(
    name="apply_desp_ajuste_poscosecha_ml1_on_ml2_hidr",
    layer="ml2",
    script_relpath="src/gold/apply_desp_ajuste_poscosecha_ml1_on_ml2_hidr.py",
    outputs_rel=["gold/pred_poscosecha_ml2_full_grado_dia_bloque_destino.parquet"],
),
Step(
    name="pred_poscosecha_ml2_views_from_full",
    layer="ml2",
    script_relpath="src/gold/build_pred_poscosecha_ml2_views_from_full.py",
    outputs_rel=[
        "gold/pred_poscosecha_ml2_dia_bloque_destino.parquet",
        "gold/pred_poscosecha_ml2_dia_destino.parquet",
        "gold/pred_poscosecha_ml2_dia_total.parquet",
    ],
),

# (Opcional) Backtest-only (NO usar con prod si tu script no soporta prod)
# Step(
#     name="apply_dh_poscosecha_ml2",
#     layer="ml2",
#     script_relpath="src/models/ml2/apply_dh_poscosecha_ml2.py",
#     outputs_rel=[],
#     args=["--mode", "backtest"],
# ),
# Step(
#     name="apply_hidr_poscosecha_ml2",
#     layer="ml2",
#     script_relpath="src/models/ml2/apply_hidr_poscosecha_ml2.py",
#     outputs_rel=[],
#     args=["--mode", "backtest"],
# ),

]

# =====
# AUDIT

```

```
# =====
steps += [
    Step(
        name="audit_universe_harvest_grid_mll",
        layer="audit",
        script_relpath="src/audit/audit_universe_harvest_grid_mll.py",
        outputs_rel=["audit/audit_universe_harvest_grid_mll_checks.parquet"],
    ),
    Step(
        name="audit_dist_grado_mll",
        layer="audit",
        script_relpath="src/audit/audit_dist_grado_mll.py",
        outputs_rel=["audit/audit_dist_grado_mll_checks.parquet"],
    ),
    Step(
        name="audit_poscosecha_seed",
        layer="audit",
        script_relpath="src/audit/audit_poscosecha_seed.py",
        outputs_rel=["audit/audit_poscosecha_seed_checks.parquet"],
    ),
    Step(
        name="audit_mll_curva_clipping_impact",
        layer="audit",
        script_relpath="src/audit/audit_mll_curva_clipping_impact.py",
        outputs_rel=[
            "audit/audit_mll_curva_clipping_by_cycle.parquet",
            "audit/audit_mll_curva_clipping_by_relpos.parquet",
            "audit/audit_mll_curva_clipping_report.parquet",
        ],
    ),
    Step(
        name="audit_harvest_windows_real",
        layer="audit",
        script_relpath="src/audit/audit_harvest_windows_real.py",
        outputs_rel=[],
    ),
    Step(
        name="audit_mass_balance_pred_oferta_vs_gold",
        layer="audit",
        script_relpath="src/audit/audit_mass_balance_pred_oferta_vs_gold.py",
        outputs_rel=[],
    ),
    Step(
        name="audit_mismatch_tallos_grado_vs_dia",
        layer="audit",
        script_relpath="src/audit/audit_mismatch_tallos_grado_vs_dia.py",
        outputs_rel=[],
    ),
]

return steps
```

```
-----
[120/164] FILE: \src\ops\runner.py
-----
from __future__ import annotations
```

```
import os
import subprocess
import time
from dataclasses import dataclass, field
from pathlib import Path
from typing import Optional
```

```
@dataclass(frozen=True)
class Step:
```

```
    """
```

```
    Step ejecutable del pipeline.
```

```
    - outputs_rel: lista de outputs relativos a artifacts_dir (data/...)
```

```

    Si es [] => nunca se hace skip por existencia (corre siempre)
- args: argumentos CLI para el script (ej: ["--mode","prod"])
- extra_env: variables de entorno extra por step
- timeout_seconds: si un step se cuelga, se corta
"""
name: str
layer: str
script_relpath: str
outputs_rel: list[str]

args: list[str] = field(default_factory=list)
extra_env: dict[str, str] = field(default_factory=dict)
timeout_seconds: Optional[int] = None

@dataclass
class RunResult:
    step: Step
    ok: bool
    seconds: float
    returncode: int
    stdout_path: Path
    stderr_path: Path

def _outputs_exist(outputs: list[Path]) -> bool:
    return bool(outputs) and all(p.exists() for p in outputs)

def _fmt_cmd(cmd: list[str]) -> str:
    def q(s: str) -> str:
        return f'"{s}"' if (" " in s or "\t" in s) else s
    return " ".join(q(x) for x in cmd)

def run_step(
    step: Step,
    python_exe: str,
    repo_root: Path,
    artifacts_dir: Path,
    log_dir: Path,
    force: bool,
    dry_run: bool,
    extra_env: Optional[dict[str, str]] = None,
) -> RunResult:
    script_path = (repo_root / step.script_relpath).resolve()
    outputs_abs = [(artifacts_dir / o).resolve() for o in step.outputs_rel]

    log_dir.mkdir(parents=True, exist_ok=True)
    stdout_path = log_dir / f"{step.layer}__{step.name}.out.log"
    stderr_path = log_dir / f"{step.layer}__{step.name}.err.log"

    if not script_path.exists():
        stderr_path.write_text(f"Script not found: {script_path}\n", encoding="utf-8")
        return RunResult(step, False, 0.0, 2, stdout_path, stderr_path)

    if (not force) and _outputs_exist(outputs_abs):
        return RunResult(step, True, 0.0, 0, stdout_path, stderr_path)

    cmd = [python_exe, str(script_path), *step.args]

    env = dict(os.environ)

    # env global
    if extra_env:
        env.update(extra_env)

    # env por step
    if step.extra_env:
        env.update(step.extra_env)

    # ? CLAVE: incluir repo_root y repo_root/src en PYTHONPATH

```

```

root_path = str(repo_root.resolve())
src_path = str((repo_root / "src").resolve())

# Mantener PYTHONPATH previo si existe
prev = env.get("PYTHONPATH", "")
parts = [root_path, src_path]
if prev:
    parts.append(prev)
env["PYTHONPATH"] = os.pathsep.join(parts)

header = []
header.append(f"[{time.strftime('%Y-%m-%d %H:%M:%S')}] STEP {step.layer}:{step.name}")
header.append(f"script={script_path}")
header.append(f"cmd={_fmt_cmd(cmd)}")
header.append(f"cwd={repo_root}")
header.append(f"PYTHONPATH={env['PYTHONPATH']}")
if step.timeout_seconds:
    header.append(f"timeout_seconds={step.timeout_seconds}")
if step.outputs_rel:
    header.append("outputs:")
    for o in outputs_abs:
        header.append(f" - {o}")
else:
    header.append("outputs: (none) => no-skip mode")
header_txt = "\n".join(header) + "\n\n"

if dry_run:
    stdout_path.write_text("[DRY RUN]\n" + header_txt, encoding="utf-8")
    stderr_path.write_text("", encoding="utf-8")
    return RunResult(step, True, 0.0, 0, stdout_path, stderr_path)

t0 = time.time()
rc = 0

try:
    with stdout_path.open("w", encoding="utf-8") as out, stderr_path.open("w", encoding="utf-8") as err:
        out.write(header_txt)
        out.flush()

        p = subprocess.run(
            cmd,
            cwd=str(repo_root),
            stdout=out,
            stderr=err,
            env=env,
            timeout=step.timeout_seconds,
        )
        rc = p.returncode

except subprocess.TimeoutExpired:
    rc = 124
    with stderr_path.open("a", encoding="utf-8") as err:
        err.write(f"\n[ERROR] TimeoutExpired after {step.timeout_seconds}s\n")

except Exception as e:
    rc = 3
    with stderr_path.open("a", encoding="utf-8") as err:
        err.write(f"\n[ERROR] Exception running step: {e}\n")

seconds = time.time() - t0
ok = (rc == 0)
return RunResult(step, ok, seconds, rc, stdout_path, stderr_path)

```

```

-----
[121/164] FILE: \src\ops\validators.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
import pandas as pd

```

```

def validate_exists(outputs: list[Path]) -> None:
    missing = [p for p in outputs if not p.exists()]
    if missing:
        raise FileNotFoundError(f"Missing outputs: {missing}")

def validate_parquet_nonempty(outputs: list[Path]) -> None:
    for p in outputs:
        if p.suffix.lower() != ".parquet":
            continue
        df = pd.read_parquet(p)
        if len(df) == 0:
            raise ValueError(f"Parquet has 0 rows: {p}")

-----
[122/164] FILE: \src\preds\build_capacidad_cosecha_dia.py
-----
# src/preds/build_capacidad_cosecha_dia.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd
import yaml

from common.io import write_parquet

# -----
# Config / helpers
# -----
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce")

def _norm_date(s: pd.Series) -> pd.Series:
    return _to_dt(s).dt.normalize()

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def _warn(msg: str) -> None:
    print(f"[WARN] {msg}")

def _pick_col(df: pd.DataFrame, candidates: list[str], required: bool = True) -> str | None:
    cols = list(df.columns)
    cols_l = {c.lower(): c for c in cols}
    for cand in candidates:
        if cand in cols:
            return cand
        cl = cand.lower()
        if cl in cols_l:
            return cols_l[cl]
    if required:
        raise ValueError(f"No encontré columna. Candidatos={candidates}. Disponibles={cols}")
    return None

```

```

def _require_cols(df: pd.DataFrame, cols: list[str], name: str) -> None:
    missing = [c for c in cols if c not in df.columns]
    if missing:
        raise ValueError(f"{name}: faltan columnas {missing}. Disponibles={list(df.columns)}")

def _norm_station_from_area(area: pd.Series) -> pd.Series:
    """
    Regla negocio:
    - A-4 / A4 / SJP / SAN JUAN => A4
    - todo lo demás => MAIN
    """
    a = area.astype(str).str.upper().str.strip()
    is_a4 = a.isin(["A-4", "A4", "SJP", "SAN JUAN"])
    return pd.Series(np.where(is_a4, "A4", "MAIN"), index=area.index)

def _ensure_area_from_inputs(pred: pd.DataFrame, pg: pd.DataFrame | None) -> pd.DataFrame:
    """
    Retorna pred con columna canónica area_trabajada.
    Estrategia:
    1) si pred trae area/area_trabajada => usarla
    2) si no trae, y existe pred_peso_grado con área => intentar mapear por (fecha,bloque,variedad)
    3) else => ALL (con warning)
    """
    pred = pred.copy()

    col_area = _pick_col(pred, ["area_trabajada", "area", "Area"], required=False)
    if col_area is not None:
        pred["area_trabajada"] = pred[col_area].astype(str).str.upper().str.strip()
        return pred

    # intentar mapear con pred_peso_grado si existe y si pred trae bloque
    col_bloque = _pick_col(pred, ["bloque", "Bloque", "bloque_norm", "bloque_padre"], required=False)
    if col_bloque is not None and pg is not None and len(pg):
        pred["_bloque_key"] = pred[col_bloque].astype(str).str.extract(r"^(\\d+)", expand=False)
        pred["_bloque_key"] = pd.to_numeric(pred["_bloque_key"], errors="coerce").astype("Int64")

    pg = pg.copy()
    if "fecha" in pg.columns:
        pg["fecha"] = _norm_date(pg["fecha"])

    col_pg_area = _pick_col(pg, ["area", "area_trabajada", "Area"], required=False)
    col_pg_bloque = _pick_col(pg, ["bloque", "Bloque", "bloque_padre"], required=False)
    col_pg_var = _pick_col(pg, ["variedad_std", "variedad", "variedad_estandar"], required=False)

    if col_pg_area and col_pg_bloque and col_pg_var:
        pg["_bloque_key"] = pg[col_pg_bloque].astype(str).str.extract(r"^(\\d+)", expand=False)
        pg["_bloque_key"] = pd.to_numeric(pg["_bloque_key"], errors="coerce").astype("Int64")

        pg["area_trabajada"] = pg[col_pg_area].astype(str).str.upper().str.strip()
        pg["variedad"] = (
            pg[col_pg_var].astype(str).str.upper().str.strip()
            .replace({"XLENCE": "XL", "CLOUD": "CLO"})
        )

    # mapping por (fecha, bloque, variedad) -> área (modo)
    m = (
        pg.dropna(subset=["fecha", "_bloque_key", "area_trabajada", "variedad"])
        .groupby(["fecha", "_bloque_key", "variedad"], dropna=False)["area_trabajada"]
        .agg(lambda s: s.mode().iat[0] if len(s.mode()) else s.iloc[0])
        .reset_index()
    )

    pred = pred.merge(
        m,
        left_on=["fecha", "_bloque_key", "variedad"],
        right_on=["fecha", "_bloque_key", "variedad"],
        how="left",
    )

```

```

miss = int(pred["area_trabajada"].isna().sum())
if miss > 0:
    _warn(f"No pude mapear área para {miss} filas (se asigna 'ALL').")
    pred["area_trabajada"] = pred["area_trabajada"].fillna("ALL")

pred = pred.drop(columns=["_bloque_key"])
return pred

_warn("pred_tallos_cosecha_dia no trae área y no pude mapearla: se asigna area_trabajada='ALL'.")
pred["area_trabajada"] = "ALL"
return pred

def _validate_unique(df: pd.DataFrame, keys: list[str], name: str) -> None:
    dup = int(df.duplicated(subset=keys).sum())
    if dup > 0:
        # muestra ejemplos
        ex = df.loc[df.duplicated(subset=keys, keep=False), keys].head(20)
        raise ValueError(
            f"{name}: llaves no únicas (dup={dup}) para keys={keys}. Ejemplos:\n{ex.to_string(index=False)}"
        )

# -----
# Main
# -----
def main() -> None:
    cfg = load_settings()

    silver_dir = Path(cfg["paths"]["silver"])
    preds_dir = Path(cfg["paths"]["preds"])
    preds_dir.mkdir(parents=True, exist_ok=True)

    # Inputs
    path_pred_tallos = preds_dir / "pred_tallos_cosecha_dia.parquet"
    path_factor_uph = silver_dir / "dim_factor_uph_cosecha_clima.parquet"
    path_base_uph = silver_dir / "dim_baseline_uph_cosecha.parquet"
    path_weather = silver_dir / "weather_hour_wide.parquet"
    path_pg = preds_dir / "pred_peso_grado.parquet"

    for p in [path_pred_tallos, path_factor_uph, path_base_uph, path_weather]:
        if not p.exists():
            raise FileNotFoundError(f"No existe input requerido: {p}")

    pred = pd.read_parquet(path_pred_tallos)
    f = pd.read_parquet(path_factor_uph)
    b = pd.read_parquet(path_base_uph)
    w = pd.read_parquet(path_weather)
    pg = pd.read_parquet(path_pg) if path_pg.exists() else None

    pred.columns = [str(c).strip() for c in pred.columns]
    f.columns = [str(c).strip() for c in f.columns]
    b.columns = [str(c).strip() for c in b.columns]
    w.columns = [str(c).strip() for c in w.columns]
    if pg is not None:
        pg.columns = [str(c).strip() for c in pg.columns]

    # =====
    # 1) pred_tallos: normalizar + asegurar área + SELLAR GRANO
    # =====
    col_fecha = _pick_col(pred, ["fecha", "Fecha", "dt", "date"])
    pred["fecha"] = _norm_date(pred[col_fecha])

    col_station = _pick_col(pred, ["station", "estacion"], required=False)
    if col_station is None:
        col_area_tmp = _pick_col(pred, ["area_trabajada", "area", "Area"], required=False)
        if col_area_tmp is not None:
            pred["station"] = _norm_station_from_area(pred[col_area_tmp])
        else:
            raise ValueError("pred_tallos_cosecha_dia: falta 'station' y no hay 'area' para inferirlo.")
    else:
        pred["station"] = pred[col_station].astype(str).str.upper().str.strip()

```

```

col_var = _pick_col(pred, ["variedad", "variedad_std", "variedad_estandar"])
pred["variedad"] = (
    pred[col_var].astype(str).str.upper().str.strip()
    .replace({"XLENCE": "XL", "CLOUD": "CLO"})
)

col_tallos = _pick_col(pred, ["tallos_pred", "tallos_proy", "tallos", "tallos_pred_dia", "tallos_pred_tot
1"])
pred["tallos_proy"] = _to_num(pred[col_tallos]).fillna(0.0)

pred = pred[pred["fecha"].notna()].copy()
pred = pred[pred["station"].isin(["MAIN", "A4"])] .copy()
pred = pred[pred["variedad"].isin(["XL", "CLO"])] .copy()

# área
pred = _ensure_area_from_inputs(pred, pg)
pred["area_trabajada"] = pred["area_trabajada"].astype(str).str.upper().str.strip()

# SELLAR grano (evita cualquier duplicado upstream)
pred = (
    pred.groupby(["fecha", "station", "area_trabajada", "variedad"], dropna=False)
    .agg(tallos_proy=("tallos_proy", "sum"))
    .reset_index()
)
_validate_unique(pred, ["fecha", "station", "area_trabajada", "variedad"], "pred_tallos_cosecha_dia (sella
do)")

# =====
# 2) Baseline UPH (ZCSP equiv) por station+variedad
# =====
if "uph_base_mediana" not in b.columns:
    raise ValueError("dim_baseline_uph_cosecha: falta uph_base_mediana.")
b2 = b.copy()
b2["station"] = b2["station"].astype(str).str.upper().str.strip()
b2["variedad"] = b2["variedad"].astype(str).str.upper().str.strip()
b2["uph_base_zcsp_equiv"] = _to_num(b2["uph_base_mediana"])

base_keep = ["station", "variedad", "uph_base_zcsp_equiv"]
if "n_base" in b2.columns:
    base_keep.append("n_base")
else:
    b2["n_base"] = np.nan
    base_keep.append("n_base")

_validate_unique(b2[base_keep], ["station", "variedad"], "dim_baseline_uph_cosecha")

# =====
# 3) Clima por hora -> factor_uph por hora (pelado=1) -> factor diario por jornada
# =====
# 3.1 weather_hour_wide mínimo
col_w_dt = _pick_col(w, ["dt_hora", "fecha", "datetime", "timestamp"])
# si viene "fecha" con hora, también sirve
w["dt_hora"] = _to_dt(w[col_w_dt]).dt.floor("h")
w = w[w["dt_hora"].notna()].copy()
w["fecha"] = w["dt_hora"].dt.normalize()
w["hora_n"] = w["dt_hora"].dt.hour.astype(int)

col_w_station = _pick_col(w, ["station", "estacion", "station_code"])
w["station"] = w[col_w_station].astype(str).str.upper().str.strip().replace({"A-4": "A4", "SJP": "A4"})
w = w[w["station"].isin(["MAIN", "A4"])] .copy()

col_estado = _pick_col(w, ["Estado_Kardex", "estado_kardex"])
col_lluvia = _pick_col(w, ["En_Lluvia", "en_lluvia"])
w["Estado_Kardex"] = w[col_estado].astype(str).str.upper().str.strip().replace({"HÚMEDO": "HUMEDO"})
w["En_Lluvia"] = _to_num(w[col_lluvia]).fillna(0).astype(int)
w["estado_final"] = np.where(w["En_Lluvia"].eq(1), "LLUVIA", w["Estado_Kardex"])
w["estado_final"] = w["estado_final"].astype(str).str.upper().str.strip().replace({"HÚMEDO": "HUMEDO"})

# reducir a llaves únicas por hora+station
w2 = (
    w[["dt_hora", "fecha", "hora_n", "station", "estado_final"]]

```



```

        .dropna(subset=["dt_hora", "station"])
        .drop_duplicates(subset=["dt_hora", "station"], keep="last")
        .copy()
    )
    _validate_unique(w2, ["dt_hora", "station"], "weather_hour_wide (w2)")

    # 3.2 dim_factor_uph_cosecha_clima: usar pelado=1 como estándar (ZCSP)
    _require_cols(f, ["station", "variedad", "pelado", "estado_final", "factor_uph"], "dim_factor_uph_cosecha_
clima")

    f2 = f.copy()
    f2["station"] = f2["station"].astype(str).str.upper().str.strip()
    f2["variedad"] = f2["variedad"].astype(str).str.upper().str.strip().replace({"XLENCE": "XL", "CLOUD": "CLO
"})
    f2["estado_final"] = f2["estado_final"].astype(str).str.upper().str.strip().replace({"HÚMEDO": "HUMEDO"})
    f2["pelado"] = _to_num(f2["pelado"]).fillna(0).astype(int)
    f2["factor_uph"] = _to_num(f2["factor_uph"])

    keep_cols = ["station", "variedad", "estado_final", "factor_uph"]
    if "n_obs" in f2.columns:
        keep_cols.append("n_obs")

    f_p1 = f2[f2["pelado"] == 1][keep_cols].copy()
    f_p0 = f2[f2["pelado"] == 0][keep_cols].copy()

    # --- Resolver duplicados en dim_factor: colapsar a 1 fila por (station,variedad,estado_final)
    def _collapse_factor(df_in: pd.DataFrame, name: str) -> pd.DataFrame:
        dfc = df_in.copy()

        # Si existe n_obs (lo usual), usamos una mediana ponderada aproximada repitiendo por bins no es eficie
nte,
        # así que usamos una aproximación robusta: promedio ponderado si hay pesos, si no mediana simple.
        # Para planificación, promedio ponderado por n_obs suele ser estable.
        if "n_obs" in dfc.columns:
            dfc["n_obs"] = _to_num(dfc["n_obs"]).fillna(0.0)
            g = (
                dfc.groupby(["station", "variedad", "estado_final"], dropna=False)
                .apply(lambda x: pd.Series({
                    "factor_uph": float(
                        np.average(
                            _to_num(x["factor_uph"]).fillna(1.0).to_numpy(),
                            weights=x["n_obs"].to_numpy()
                        )
                    ) if x["n_obs"].sum() > 0 else float(_to_num(x["factor_uph"]).median())
                }))
                .reset_index()
            )
        else:
            g = (
                dfc.groupby(["station", "variedad", "estado_final"], dropna=False)
                .agg(factor_uph=("factor_uph", "median"))
                .reset_index()
            )

        # clip de seguridad (misma filosofía que en dim_factor)
        g["factor_uph"] = _to_num(g["factor_uph"]).fillna(1.0).clip(lower=0.30, upper=1.20)

        # validación final: ahora sí debe ser único
        _validate_unique(g, ["station", "variedad", "estado_final"], f"{name} (colapsado)")
        return g

    # colapsar duplicados (LLUVIA y otros)
    f_p1 = _collapse_factor(f_p1, "dim_factor_uph pelado=1")
    f_p0 = _collapse_factor(f_p0, "dim_factor_uph pelado=0")

    # 3.3 Expandir a factor por hora y variedad:
    # Para cada (dt_hora, station) se replica por variedad usando join con f_p1.
    variedades = pd.DataFrame({"variedad": ["XL", "CLO"]})
    wh = w2.merge(variedades, how="cross") # (hora,station) x variedad

```

```

# Join factor pelado=1; fallback a pelado=0; fallback final 1.0
wh = wh.merge(
    f_p1.rename(columns={"factor_uph": "factor_p1"}),
    on=["station", "variedad", "estado_final"],
    how="left",
).merge(
    f_p0.rename(columns={"factor_uph": "factor_p0"}),
    on=["station", "variedad", "estado_final"],
    how="left",
)

wh["factor_uph"] = _to_num(wh["factor_p1"]).fillna(_to_num(wh["factor_p0"])).fillna(1.0)
wh = wh.drop(columns=["factor_p1", "factor_p0"])

# 3.4 Agregar a día por horas de jornada
horas_turno = float(cfg.get("cosecha", {}).get("horas_turno_cosecha", 9.5))
# horas jornada: por defecto 6..14 (9 horas). Ajusta en settings si quieres.
shift_hours = cfg.get("cosecha", {}).get("shift_hours", list(range(6, 15))) # 6..14
shift_hours = [int(x) for x in shift_hours]

wh_j = wh[wh["hora_n"].isin(shift_hours)].copy()

factor_dia = (
    wh_j.groupby(["fecha", "station", "variedad"], dropna=False)
        .agg(
            factor_uph_dia=("factor_uph", "median"),
            n_horas=("factor_uph", "size"),
        )
        .reset_index()
)
_validate_unique(factor_dia, ["fecha", "station", "variedad", "factor_uph_dia"])

# =====
# 4) Join final: pred (día-área-variedad) + factor_dia + baseline
# =====
m = pred.merge(factor_dia, on=["fecha", "station", "variedad"], how="left")
miss_factor = int(m["factor_uph_dia"].isna().sum())
if miss_factor > 0:
    _warn(f"Faltan {miss_factor} factores diarios (se imputan a 1.0). Revisa cobertura weather_hour_wide.")
)

m["factor_uph_dia"] = _to_num(m["factor_uph_dia"]).fillna(1.0)

m = m.merge(
    b2[base_keep],
    on=["station", "variedad"],
    how="left",
)
miss_base = int(m["uph_base_zcsp_equiv"].isna().sum())
if miss_base > 0:
    raise ValueError(f"Faltan {miss_base} baselines por station/variedad. Revisa dim_baseline_uph_cosecha.")
)

# UPH efectiva
m["uph_eff"] = m["uph_base_zcsp_equiv"] * m["factor_uph_dia"]

# Horas requeridas y personas (jornada semilla)
m["horas_req"] = np.where(m["uph_eff"] > 0, m["tallos_proy"] / m["uph_eff"], np.nan)
m["personas_req"] = np.where(m["horas_req"].notna(), m["horas_req"] / horas_turno, np.nan)

# Si por cualquier razón alguien metió area_trabajada adicional, sellamos grano final:
out = m[
    "fecha", "station", "area_trabajada", "variedad",
    "tallos_proy",
    "uph_base_zcsp_equiv",
    "factor_uph_dia",
    "uph_eff",
    "horas_req",
    "personas_req",
    "n_base",
    "n_horas",
].copy()

```

```

out["n_horas"] = _to_num(out["n_horas"]).fillna(0).astype(int)
out["created_at"] = datetime.now().isoformat(timespec="seconds")

# Validación final de unicidad
_validate_unique(out, ["fecha", "station", "area_trabajada", "variedad"], "capacidad_cosecha_dia (output)"
)

# Guardar
out_path = preds_dir / "capacidad_cosecha_dia.parquet"
write_parquet(out, out_path)

_info(f"OK: capacidad_cosecha_dia={len(out)} filas -> {out_path}")
_info(f"Rango fechas: {out['fecha'].min()} -> {out['fecha'].max()}")
_info(f"Stations: {out['station'].value_counts(dropna=False).to_dict()}")
_info(f"Áreas (top): {out['area_trabajada'].value_counts(dropna=False).head(10).to_dict()}")
_info(f"Variedades: {out['variedad'].value_counts(dropna=False).to_dict()}")

# Extra: check rápido de multiplicación típica (tu prueba del 9)
vc = out.groupby(["fecha", "station", "area_trabajada", "variedad"]).size().value_counts().head(10)
_info(f"Check duplicados (size por llave) -> {vc.to_dict()}")

if __name__ == "__main__":
    main()

-----
[123/164] FILE: \src\preds\build_pred_cajas_from_peso_final.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import read_parquet, write_parquet

# -----
# Config / helpers
# -----
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def _warn(msg: str) -> None:
    print(f"[WARN] {msg}")

def _require_cols(df: pd.DataFrame, cols: list[str], name: str) -> None:
    missing = [c for c in cols if c not in df.columns]
    if missing:
        raise ValueError(f"{name}: faltan columnas {missing}. Disponibles={list(df.columns)}")

def _validate_unique(df: pd.DataFrame, keys: list[str], name: str) -> None:
    dup = int(df.duplicated(subset=keys).sum())

```

```

if dup > 0:
    ex = df.loc[df.duplicated(subset=keys, keep=False), keys].head(30)
    raise ValueError(
        f"{name}: llaves no únicas dup={dup} para keys={keys}. Ejemplos:\n{ex.to_string(index=False)}"
    )

# -----
# Main
# -----
def main() -> None:
    cfg = load_settings()

    preds_dir = Path(cfg.get("paths", {})).get("preds", "data/preds")
    preds_dir.mkdir(parents=True, exist_ok=True)

    in_path = preds_dir / "pred_peso_final_ajustado_grado.parquet"
    if not in_path.exists():
        raise FileNotFoundError(f"No existe: {in_path}. Ejecuta build_pred_peso_final_ajustado_grado primero.")
)

df = read_parquet(in_path).copy()
df.columns = [str(c).strip() for c in df.columns]

# --- Requeridos mínimos para este pred ---
# Nota: el resto de columnas las pasamos "si existen", pero no deben romper.
required_min = ["peso_final_g", "fecha_post", "grado"]
_require_cols(df, required_min, "pred_peso_final_ajustado_grado")

# Normalización
df["fecha_post"] = _norm_date(df["fecha_post"])
df["grado"] = _to_num(df["grado"]).astype("Int64")

# OJO: NO llenar NaN con 0; si no hay peso es problema upstream.
df["peso_final_g"] = _to_num(df["peso_final_g"])

# filtros mínimos
n0 = len(df)
df = df[df["fecha_post"].notna()].copy()
df = df[df["grado"].notna()].copy()

miss_peso = int(df["peso_final_g"].isna().sum())
if miss_peso > 0:
    _warn(f"Hay {miss_peso} filas sin peso_final_g (NaN). Se excluyen para cálculo de cajas.")
    df = df[df["peso_final_g"].notna()].copy()

# sanity: pesos no negativos
neg = int((df["peso_final_g"] < 0).sum())
if neg > 0:
    _warn(f"Hay {neg} filas con peso_final_g negativo. Se excluyen.")
    df = df[df["peso_final_g"] >= 0].copy()

_info(f"Input: {n0} filas; después filtros mínimos: {len(df)} filas")

# Conversión a kg y cajas (10 kg/caja)
df["peso_final_kg"] = df["peso_final_g"] / 1000.0
df["cajas_pred_grado"] = df["peso_final_kg"] / 10.0 # 10 kg / caja

df["created_at"] = datetime.now().isoformat(timespec="seconds")

# -----
# Output 1: pred_cajas_grado (detalle)
# -----
# Columnas ?deseadas? (si faltan, no revienta; solo se omiten)
desired_cols = [
    "ciclo_id",
    "fecha", "fecha_post", "dh_dias",
    "destino",
    "bloque", "bloque_padre",
    "variedad", "variedad_std",
    "tipo_sp", "area", "estado",
    "stage",

```

```

    "grado",
    "tallos_pred_grado",
    "peso_final_g",
    "peso_final_kg",
    "cajas_pred_grado",
    "created_at",
]
out_grado_cols = [c for c in desired_cols if c in df.columns]

out_grado = df[out_grado_cols].copy()

# Validación grano (elige el set de llaves más estable según tu upstream)
# Preferimos: ciclo_id + fecha_post + bloque_padre + destino + grado
keys_candidates = [
    ["ciclo_id", "fecha_post", "bloque_padre", "destino", "grado"],
    ["fecha_post", "bloque_padre", "destino", "grado"],
    ["fecha_post", "destino", "grado"],
]
chosen_keys = None
for k in keys_candidates:
    if all(col in out_grado.columns for col in k):
        chosen_keys = k
        break
if chosen_keys is None:
    _warn("No pude validar unicidad de pred_cajas_grado: faltan columnas de llaves. (No es fatal, pero revísalo).")
else:
    _validate_unique(out_grado, chosen_keys, "pred_cajas_grado")

out_grado_path = preds_dir / "pred_cajas_grado.parquet"
write_parquet(out_grado, out_grado_path)

# -----
# Output 2: pred_cajas_dia (agregado)
# -----
group_keys = ["ciclo_id", "fecha_post", "bloque", "bloque_padre", "variedad", "variedad_std", "destino"]
group_keys = [k for k in group_keys if k in out_grado.columns]

if "fecha_post" not in group_keys:
    raise ValueError("No puedo construir pred_cajas_dia: falta 'fecha_post' en el dataset.")

daily = (
    out_grado.groupby(group_keys, dropna=False)
    .agg(
        peso_final_g=("peso_final_g", "sum"),
        peso_final_kg=("peso_final_kg", "sum"),
        cajas_pred=("cajas_pred_grado", "sum"),
    )
    .reset_index()
)
daily["created_at"] = datetime.now().isoformat(timespec="seconds")

_validate_unique(daily, group_keys, "pred_cajas_dia")

out_daily_path = preds_dir / "pred_cajas_dia.parquet"
write_parquet(daily, out_daily_path)

# -----
# Logs
# -----
_info(f"OK: pred_cajas_grado={len(out_grado)} -> {out_grado_path}")
_info(f"OK: pred_cajas_dia={len(daily)} -> {out_daily_path}")

if "cajas_pred_grado" in out_grado.columns and len(out_grado):
    _info("cajas_pred_grado describe:\n" + out_grado["cajas_pred_grado"].describe().to_string())
if len(daily):
    _info("cajas_pred (daily) describe:\n" + daily["cajas_pred"].describe().to_string())

if __name__ == "__main__":
    main()

```

```
-----  
[124/164] FILE: \src\preds\build_pred_horas_poscosecha.py  
-----
```

```
from __future__ import annotations
```

```
from pathlib import Path  
from datetime import datetime  
import pandas as pd  
import numpy as np  
import yaml
```

```
from common.io import read_parquet, write_parquet
```

```
# -----
```

```
# Config / helpers
```

```
# -----
```

```
def load_settings() -> dict:  
    with open("config/settings.yaml", "r", encoding="utf-8") as f:  
        return yaml.safe_load(f)
```

```
def _norm_date(s: pd.Series) -> pd.Series:  
    return pd.to_datetime(s, errors="coerce").dt.normalize()
```

```
def _week_id(fecha: pd.Series) -> pd.Series:  
    # Tu lógica: +2 días y semana Sunday-first (%U) => YYWW  
    fa = _norm_date(fecha) + pd.to_timedelta(2, unit="D")  
    yy = fa.dt.year.astype(str).str[-2:]  
    ww = fa.dt.strftime("%U").astype(int)  
    return yy + ww.astype(str).str.zfill(2)
```

```
def safe_div(a, b):  
    a = np.asarray(a, dtype="float64")  
    b = np.asarray(b, dtype="float64")  
    return np.where(b == 0, np.nan, a / b)
```

```
def _info(msg: str) -> None:  
    print(f"[INFO] {msg}")
```

```
def _warn(msg: str) -> None:  
    print(f"[WARN] {msg}")
```

```
def _pick_pred_file(preds_dir: Path) -> Path:  
    """  
    Escoge el mejor candidato existente como input de demanda.  
    (No depende de features; solo de preds ya construidos).  
    """  
    candidates = [  
        preds_dir / "pred_peso_final_ajustado_grado.parquet",  
        preds_dir / "pred_peso_final_grado_dia.parquet",  
        preds_dir / "pred_peso_grado.parquet",  
        preds_dir / "pred_oferta_grado.parquet",  
        preds_dir / "pred_oferta_dia.parquet",  
    ]  
    for p in candidates:  
        if p.exists():  
            return p  
    raise FileNotFoundError(  
        "No encuentro parquet de pred en data/preds. Busqué:\n- "  
        + "\n- ".join(str(x) for x in candidates)  
    )
```

```
def _ensure_peso_final_g(df: pd.DataFrame) -> pd.DataFrame:  
    """
```

```

Genera/estandariza 'peso_final_g' en el dataframe de pred.
Reglas:
1) si existe peso_final_g => listo
2) si existe peso_final_kg => peso_final_g = *1000
3) si existe peso_hidratado_g + factor_desp + ajuste => peso_final_g = peso_hidratado_g * factor_desp /
ajuste
4) fallback a peso_pred_g / peso_real_g (si existe)
"""
df = df.copy()
cols = {c.lower(): c for c in df.columns}

def has(name: str) -> bool:
    return name.lower() in cols

if has("peso_final_g"):
    df["peso_final_g"] = pd.to_numeric(df[cols["peso_final_g"]], errors="coerce").fillna(0.0)
    return df

if has("peso_final_kg"):
    x = pd.to_numeric(df[cols["peso_final_kg"]], errors="coerce").fillna(0.0)
    df["peso_final_g"] = x * 1000.0
    return df

if has("peso_hidratado_g") and has("factor_desp") and has("ajuste"):
    ph = pd.to_numeric(df[cols["peso_hidratado_g"]], errors="coerce").fillna(0.0)
    fd = pd.to_numeric(df[cols["factor_desp"]], errors="coerce").fillna(1.0)
    aj = pd.to_numeric(df[cols["ajuste"]], errors="coerce").replace(0, np.nan)
    df["peso_final_g"] = (ph * fd) / aj
    df["peso_final_g"] = df["peso_final_g"].fillna(0.0)
    return df

if has("peso_pred_g"):
    df["peso_final_g"] = pd.to_numeric(df[cols["peso_pred_g"]], errors="coerce").fillna(0.0)
    return df

if has("peso_real_g"):
    df["peso_final_g"] = pd.to_numeric(df[cols["peso_real_g"]], errors="coerce").fillna(0.0)
    return df

raise ValueError("Pred: no pude derivar peso_final_g. Columnas disponibles: " + ", ".join(df.columns))

def _ensure_fecha_post(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    cols = {c.lower(): c for c in df.columns}
    if "post_start" in cols:
        df["fecha_post"] = _norm_date(df[cols["post_start"]])
        return df
    if "fecha_post" in cols:
        df["fecha_post"] = _norm_date(df[cols["fecha_post"]])
        return df
    if "fecha" in cols:
        df["fecha_post"] = _norm_date(df[cols["fecha"]])
        return df
    raise ValueError("Pred: no encuentro post_start/fecha_post/fecha para derivar fecha_post")

def _pick_first_existing(paths: list[Path]) -> Path | None:
    for p in paths:
        if p.exists():
            return p
    return None

def _get_baseline_kg_h_blanco(silver_dir: Path) -> tuple[float | None, str]:
    """
    Busca baseline kg/h persona para BLANCO en silver, con compatibilidad de nombres.
    Retorna (valor, source_name). Si no existe, (None, 'missing').
    """
    candidates = [
        silver_dir / "dim_baseline_capacidad_kg_h_persona.parquet",
        silver_dir / "dim_capacidad_baseline_proceso.parquet",
    ]

```

```

        silver_dir / "dim_capacidad_baseline_kg_h_persona.parquet",
        silver_dir / "dim_baseline_capacidad_proceso_kg_h_persona.parquet",
    ]
    p = _pick_first_existing(candidates)
    if p is None:
        return None, "missing"

    df = read_parquet(p).copy()
    df.columns = df.columns.str.strip()
    if "proceso" not in df.columns:
        raise ValueError(f"{p.name}: falta columna 'proceso'.")

    df["proceso"] = df["proceso"].astype(str).str.upper().str.strip()

    kg_col = None
    for c in ["kg_h_persona_mediana", "kg_h_persona", "kg_h_mediana", "kg_h"]:
        if c in df.columns:
            kg_col = c
            break
    if kg_col is None:
        raise ValueError(f"{p.name}: no encuentro columna kg/h persona. Columnas={list(df.columns)}")

    s = df.loc[df["proceso"].eq("BLANCO"), kg_col]
    if s.empty:
        raise ValueError(f"{p.name}: no hay fila proceso=BLANCO")

    return float(pd.to_numeric(s, errors="coerce").iloc[0]), p.name

def _get_baseline_tallos_h(silver_dir: Path, proceso: str) -> float:
    """
    Devuelve baseline de tallos/h/persona para un proceso.
    - Compatible con varios nombres de archivo (porque tu pipeline ya genera otros nombres).
    - Si no encuentra el proceso, hace fallback: OTROS -> mediana global.
    """
    candidates = [
        silver_dir / "dim_baseline_capacidad_tallos_h_persona.parquet",
        silver_dir / "dim_capacidad_baseline_tallos_proceso.parquet", # <-- el que tú generas
        silver_dir / "dim_capacidad_baseline_tallos_h_persona.parquet",
    ]
    p = _pick_first_existing(candidates)
    if p is None:
        raise FileNotFoundError(
            "No existe baseline tallos/h en silver. Busqué:\n- " + "\n- ".join(str(x) for x in candidates)
        )

    df = read_parquet(p).copy()
    df.columns = df.columns.str.strip()

    if "proceso" not in df.columns:
        raise ValueError(f"{p.name}: falta columna 'proceso'. Columnas={list(df.columns)}")

    df["proceso"] = df["proceso"].astype(str).str.upper().str.strip()

    # localizar columna tallos/h
    col = None
    for c in ["tallos_h_persona_mediana", "tallos_h_persona", "tallos_h_mediana", "tallos_h"]:
        if c in df.columns:
            col = c
            break
    if col is None:
        raise ValueError(f"{p.name}: no encuentro columna tallos/h. Columnas={list(df.columns)}")

    df[col] = pd.to_numeric(df[col], errors="coerce")

    proc = proceso.upper().strip()
    s = df.loc[df["proceso"].eq(proc), col].dropna()
    if not s.empty:
        return float(s.iloc[0])

    # fallback 1: OTROS
    s2 = df.loc[df["proceso"].eq("OTROS"), col].dropna()

```



```

if not s2.empty:
    _warn(f"{p.name}: no hay baseline para {proc}. Fallback: OTROS.")
    return float(s2.iloc[0])

# fallback 2: mediana global
gmed = float(np.nanmedian(df[col].values))
if np.isnan(gmed) or gmed <= 0:
    raise ValueError(f"{p.name}: no hay baseline válido ni para {proc} ni global (todo NaN/<=0).")
_warn(f"{p.name}: no hay baseline para {proc}. Fallback: mediana global={gmed:.3f}.")
return gmed

# -----
# Main
# -----
def main() -> None:
    cfg = load_settings()
    silver_dir = Path(cfg["paths"]["silver"])
    preds_dir = Path(cfg["paths"]["preds"])
    preds_dir.mkdir(parents=True, exist_ok=True)

    # -----
    # 1) Input demanda (desde PREDS)
    # -----
    pred_path = _pick_pred_file(preds_dir)
    pred = read_parquet(pred_path).copy()
    pred.columns = pred.columns.str.strip()

    pred = _ensure_fecha_post(pred)
    pred = _ensure_peso_final_g(pred)

    _info(
        f"Input pred: {pred_path.name} filas={len(pred)} "
        f"rango fecha_post={pred['fecha_post'].min()} -> {pred['fecha_post'].max()}"
    )

    # Agregar demanda a día
    dem = (pred.groupby(["fecha_post"], dropna=False)
           .agg(peso_final_g_total=("peso_final_g", "sum"))
           .reset_index())
    dem = dem[dem["fecha_post"].notna()].copy()
    dem["kg_total"] = dem["peso_final_g_total"] / 1000.0
    dem["cajas_total"] = dem["kg_total"] / 10.0
    dem["Semana_Ventas"] = _week_id(dem["fecha_post"])

    # -----
    # 2) Mix semanal (SILVER)
    # -----
    mix_path = silver_dir / "dim_mix_proceso_semana.parquet"
    if not mix_path.exists():
        raise FileNotFoundError(f"No existe: {mix_path} (requerido)")

    mix = read_parquet(mix_path).copy()
    mix.columns = mix.columns.str.strip()

    if "Semana_Ventas" not in mix.columns and "semana_ventas" in mix.columns:
        mix = mix.rename(columns={"semana_ventas": "Semana_Ventas"})

    for c in ["W_Blanco", "W_Arcoiris", "W_Tinturado"]:
        if c not in mix.columns:
            raise ValueError(f"Mix: falta columna {c} en {mix_path}")

    seed = {
        "W_Blanco": float(pd.to_numeric(mix["W_Blanco"], errors="coerce").median()),
        "W_Arcoiris": float(pd.to_numeric(mix["W_Arcoiris"], errors="coerce").median()),
        "W_Tinturado": float(pd.to_numeric(mix["W_Tinturado"], errors="coerce").median()),
    }

    dem = dem.merge(
        mix[["Semana_Ventas", "W_Blanco", "W_Arcoiris", "W_Tinturado"]],
        on="Semana_Ventas",

```

```

    how="left",
)

dem["W_Blanco"] = pd.to_numeric(dem["W_Blanco"], errors="coerce").fillna(seed["W_Blanco"])
dem["W_Arcoiris"] = pd.to_numeric(dem["W_Arcoiris"], errors="coerce").fillna(seed["W_Arcoiris"])
dem["W_Tinturado"] = pd.to_numeric(dem["W_Tinturado"], errors="coerce").fillna(seed["W_Tinturado"])

s = dem["W_Blanco"] + dem["W_Arcoiris"] + dem["W_Tinturado"]
dem["W_Blanco"] = safe_div(dem["W_Blanco"], s)
dem["W_Arcoiris"] = safe_div(dem["W_Arcoiris"], s)
dem["W_Tinturado"] = safe_div(dem["W_Tinturado"], s)

dem["kg_blanco"] = dem["kg_total"] * dem["W_Blanco"]
dem["kg_arcoiris"] = dem["kg_total"] * dem["W_Arcoiris"]
dem["kg_tinturado"] = dem["kg_total"] * dem["W_Tinturado"]

# -----
# 3) Peso promedio (SILVER)
# -----
peso_prom_path = silver_dir / "dim_peso_tallo_promedio_dia.parquet"
if not peso_prom_path.exists():
    raise FileNotFoundError(f"No existe: {peso_prom_path} (requerido)")

peso_prom = read_parquet(peso_prom_path).copy()
peso_prom.columns = peso_prom.columns.str.strip()

if "fecha" in peso_prom.columns:
    peso_prom["fecha_post"] = _norm_date(peso_prom["fecha"])
elif "fecha_post" in peso_prom.columns:
    peso_prom["fecha_post"] = _norm_date(peso_prom["fecha_post"])
else:
    raise ValueError("dim_peso_tallo_promedio_dia: falta fecha/fecha_post")

if "peso_tallo_prom_g" not in peso_prom.columns:
    raise ValueError("dim_peso_tallo_promedio_dia: falta peso_tallo_prom_g")

peso_seed_g = float(pd.to_numeric(peso_prom["peso_tallo_prom_g"], errors="coerce").median())
peso_prom = peso_prom[["fecha_post", "peso_tallo_prom_g"]].drop_duplicates()

out = dem.merge(peso_prom, on="fecha_post", how="left")
out["peso_tallo_prom_g"] = pd.to_numeric(out["peso_tallo_prom_g"], errors="coerce").fillna(peso_seed_g)

# -----
# 4) Capacidades baseline (SILVER)
# -----
# BLANCO: intentar kg/h directo, si no existe usar fallback desde tallos/h
kg_h_blanco, kg_h_src = _get_baseline_kg_h_blanco(silver_dir)
if kg_h_blanco is None:
    warn("No existe baseline kg/h para BLANCO en silver. Fallback: BLANCO kg/h = tallos_h(BLANCO)*peso_prom_g/1000.")
    tallos_h_blanco = _get_baseline_tallos_h(silver_dir, "BLANCO")
    out["kg_h_persona_blanco"] = (tallos_h_blanco * out["peso_tallo_prom_g"]) / 1000.0
    out["kg_h_blanco_source"] = "fallback_from_tallos_h"
else:
    out["kg_h_persona_blanco"] = float(kg_h_blanco)
    out["kg_h_blanco_source"] = kg_h_src

tallos_h_tint = _get_baseline_tallos_h(silver_dir, "TINTURADO")
tallos_h_arc = _get_baseline_tallos_h(silver_dir, "ARCOIRIS")

out["kg_h_persona_tinturado"] = (tallos_h_tint * out["peso_tallo_prom_g"]) / 1000.0
out["kg_h_persona_arcoiris"] = (tallos_h_arc * out["peso_tallo_prom_g"]) / 1000.0

# -----
# 5) Horas y personas
# -----
out["horas_req_blanco"] = safe_div(out["kg_blanco"], out["kg_h_persona_blanco"])
out["horas_req_tinturado"] = safe_div(out["kg_tinturado"], out["kg_h_persona_tinturado"])
out["horas_req_arcoiris"] = safe_div(out["kg_arcoiris"], out["kg_h_persona_arcoiris"])
out["horas_req_total"] = out["horas_req_blanco"] + out["horas_req_tinturado"] + out["horas_req_arcoiris"]

horas_turno = float(cfg.get("pipeline", {}).get("horas_turno_poscosecha", 8.0))

```

```

out["personas_req_blanco"] = safe_div(out["horas_req_blanco"], horas_turno)
out["personas_req_tinturado"] = safe_div(out["horas_req_tinturado"], horas_turno)
out["personas_req_arcoiris"] = safe_div(out["horas_req_arcoiris"], horas_turno)
out["personas_req_total"] = safe_div(out["horas_req_total"], horas_turno)

out["created_at"] = datetime.now().isoformat(timespec="seconds")

keep = [
    "fecha_post", "Semana_Ventas",
    "kg_total", "cajas_total",
    "W_Blanco", "W_Arcoiris", "W_Tinturado",
    "kg_blanco", "kg_arcoiris", "kg_tinturado",
    "peso_tallo_prom_g",
    "kg_h_persona_blanco", "kg_h_persona_arcoiris", "kg_h_persona_tinturado",
    "kg_h_blanco_source",
    "horas_req_blanco", "horas_req_arcoiris", "horas_req_tinturado", "horas_req_total",
    "personas_req_blanco", "personas_req_arcoiris", "personas_req_tinturado", "personas_req_total",
    "created_at",
]

out = out[keep].sort_values("fecha_post").reset_index(drop=True)

out_path = preds_dir / "pred_horas_poscosecha_dia.parquet"
write_parquet(out, out_path)

_info(f"OK: pred_horas_poscosecha_dia={len(out)} filas -> {out_path}")
_info(f"input_pred={pred_path.name}")
_info(f"Seed mix (mediana): {seed}")
_info(f"peso_tallo_prom_g seed: {peso_seed_g}")
_info(f"horas_turno_poscosecha: {horas_turno}")
_info(f"BLANCO baseline source: {out['kg_h_blanco_source'].iloc[0] if len(out) else 'NA'}")

```

```

if __name__ == "__main__":
    main()

```

```

[125/164] FILE: \src\preds\build_pred_milestones_baseline.py

```

```

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _safe_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _choose_group_cols(fallback_group: str) -> list[str]:
    if fallback_group == "variedad":
        return ["variedad"]
    if fallback_group == "tipo_sp":
        return ["tipo_sp"]
    if fallback_group == "area":
        return ["area"]
    if fallback_group == "global":

```

```

        return []
    # default
    return ["variedad"]

def main() -> None:
    cfg = load_settings()

    silver_dir = Path(cfg["paths"]["silver"])
    preds_dir = Path(cfg.get("paths", {}).get("preds", "data/preds"))
    preds_dir.mkdir(parents=True, exist_ok=True)

    fact_path = silver_dir / "fact_ciclo_maestro.parquet"
    if not fact_path.exists():
        raise FileNotFoundError(f"No existe: {fact_path}")

    fact = read_parquet(fact_path).copy()

    # Normalizar fechas
    for c in ["fecha_sp", "fecha_inicio_cosecha", "fecha_fin_cosecha"]:
        if c in fact.columns:
            fact[c] = _norm_date(fact[c])

    # Parámetros
    dh_days = int(cfg.get("milestones", {}).get("dh_days", 7))
    post_tail_days = int(cfg.get("milestones", {}).get("post_tail_days", 2))

    fallback_group = str(cfg.get("pred_milestones", {}).get("fallback_group", "variedad")).strip().lower()
    min_hist_rows = int(cfg.get("pred_milestones", {}).get("min_hist_rows", 10))
    group_cols = _choose_group_cols(fallback_group)

    # 1) Entrenamiento baseline: solo CERRADOS con fechas completas
    estado_col = "estado" if "estado" in fact.columns else None
    if estado_col:
        hist = fact[fact[estado_col].astype(str).str.upper().eq("CERRADO")].copy()
    else:
        hist = fact.copy()

    hist = hist[
        hist["fecha_sp"].notna()
        & hist["fecha_inicio_cosecha"].notna()
        & hist["fecha_fin_cosecha"].notna()
    ].copy()

    if len(hist) == 0:
        raise ValueError("No hay historia CERRADA con fechas completas para construir baseline de milestones.")
)

hist["dias_veg"] = (hist["fecha_inicio_cosecha"] - hist["fecha_sp"]).dt.days
hist["dur_cosecha"] = (hist["fecha_fin_cosecha"] - hist["fecha_inicio_cosecha"]).dt.days

# Remover outliers obvios (protección básica)
hist = hist[(hist["dias_veg"] >= 0) & (hist["dias_veg"] <= 500)].copy()
hist = hist[(hist["dur_cosecha"] >= 0) & (hist["dur_cosecha"] <= 500)].copy()

# Medianas por grupo (o global)
if group_cols:
    agg = (hist.groupby(group_cols, dropna=False)
            .agg(
                n=("ciclo_id", "count"),
                dias_veg_mediana=("dias_veg", "median"),
                dur_cosecha_mediana=("dur_cosecha", "median"),
            )
            .reset_index())
else:
    agg = pd.DataFrame([
        {
            "n": len(hist),
            "dias_veg_mediana": float(hist["dias_veg"].median()),
            "dur_cosecha_mediana": float(hist["dur_cosecha"].median()),
        }
    ])

# Global fallback siempre disponible

```

```

global_dias_veg = int(round(float(hist["dias_veg"].median())))
global_dur_cosecha = int(round(float(hist["dur_cosecha"].median())))

# 2) Aplicación: ciclos con milestones faltantes
need_pred = fact.copy()
need_pred = need_pred[need_pred["fecha_sp"].notna()].copy()

# Definimos ?faltante? si no hay inicio cosecha o no hay fin cosecha
miss_hs = need_pred["fecha_inicio_cosecha"].isna()
miss_he = need_pred["fecha_fin_cosecha"].isna()

need_pred = need_pred[miss_hs | miss_he].copy()
if len(need_pred) == 0:
    print("No hay ciclos con milestones faltantes. No se generó pred_milestones.")
    return

# Join con tabla de medianas
if group_cols:
    need_pred = need_pred.merge(agg, on=group_cols, how="left")
    # aplicar min_hist_rows: si el grupo tiene poca historia, usamos global
    low_hist = need_pred["n"].fillna(0) < min_hist_rows
    need_pred.loc[low_hist, "dias_veg_mediana"] = np.nan
    need_pred.loc[low_hist, "dur_cosecha_mediana"] = np.nan
else:
    need_pred["dias_veg_mediana"] = agg.loc[0, "dias_veg_mediana"]
    need_pred["dur_cosecha_mediana"] = agg.loc[0, "dur_cosecha_mediana"]

# Fallback final global
need_pred["dias_veg_fill"] = need_pred["dias_veg_mediana"].fillna(global_dias_veg).round().astype(int)
need_pred["dur_cosecha_fill"] = need_pred["dur_cosecha_mediana"].fillna(global_dur_cosecha).round().astype(int)

# Predicciones
harvest_start_pred = need_pred["fecha_sp"] + pd.to_timedelta(need_pred["dias_veg_fill"], unit="D")
harvest_end_pred = harvest_start_pred + pd.to_timedelta(need_pred["dur_cosecha_fill"], unit="D")

post_start_pred = harvest_start_pred + pd.to_timedelta(dh_days, unit="D")
post_end_pred = harvest_end_pred + pd.to_timedelta(dh_days + post_tail_days, unit="D")

# Construir tabla larga pred_milestones
pred_rows = []

def add_pred(code: str, fecha: pd.Series, method: str):
    tmp = need_pred[["ciclo_id"]].copy()
    tmp["milestone_code"] = code
    tmp["fecha_pred"] = _norm_date(fecha)
    tmp["method"] = method
    tmp["model_version"] = f"baseline_median_{'_'.join(group_cols) if group_cols else 'global'}"
    tmp["created_at"] = datetime.now().isoformat(timespec="seconds")
    pred_rows.append(tmp)

add_pred("HARVEST_START", harvest_start_pred, f"median_days_to_harvest({group_cols or ['global']})")
add_pred("HARVEST_END", harvest_end_pred, f"median_harvest_duration({group_cols or ['global']})")
add_pred("POST_START", post_start_pred, f"rule:harvest_start+{dh_days}")
add_pred("POST_END", post_end_pred, f"rule:harvest_end+{dh_days}+{post_tail_days}")

pred = pd.concat(pred_rows, ignore_index=True)
pred = pred[pred["fecha_pred"].notna()].copy()

out_path = preds_dir / "pred_milestones_ciclo.parquet"
write_parquet(pred, out_path)

print(f"OK: pred_milestones_ciclo={len(pred)} filas -> {out_path}")
print(f"Global baseline: dias_veg={global_dias_veg}, dur_cosecha={global_dur_cosecha}, DH={dh_days}")

if __name__ == "__main__":
    main()

```

```

-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.strip().str.upper()

def _pick_first_existing(df: pd.DataFrame, candidates: list[str]) -> str | None:
    for c in candidates:
        if c in df.columns:
            return c
    return None

def main() -> None:
    cfg = load_settings()

    silver_dir = Path(cfg["paths"]["silver"])
    preds_dir = Path(cfg.get("paths", {}).get("preds", "data/preds"))
    preds_dir.mkdir(parents=True, exist_ok=True)

    maestro_path = silver_dir / "fact_ciclo_maestro.parquet"
    win_path = silver_dir / "milestone_window_ciclo_final.parquet"

    if not maestro_path.exists():
        raise FileNotFoundError(f"No existe: {maestro_path}")
    if not win_path.exists():
        raise FileNotFoundError(f"No existe: {win_path}. Ejecuta primero build_milestone_window_ciclo_final_wi
th_inference.")

    c = read_parquet(maestro_path).copy()
    w = read_parquet(win_path).copy()

    # Params
    use_stage = str(cfg.get("pred_oferta", {}).get("use_stage", "HARVEST")).upper()
    if use_stage not in {"HARVEST", "HARVEST_POST"}:
        raise ValueError("pred_oferta.use_stage debe ser 'HARVEST' o 'HARVEST_POST'.")

    # ---- Maestro
    c.columns = [str(x).strip() for x in c.columns]
    c["ciclo_id"] = c["ciclo_id"].astype(str)

    # variedad + map
    if "variedad" not in c.columns:
        c["variedad"] = "UNKNOWN"
    c["variedad"] = _canon_str(c["variedad"])

    var_map = (cfg.get("mappings", {}).get("variedad_map", {}) or {})
    var_map = {str(k).strip().upper(): str(v).strip().upper() for k, v in var_map.items()}
    c["variedad"] = c["variedad"].map(lambda x: var_map.get(x, x))

    for col in ["tipo_sp", "area", "estado"]:
        if col not in c.columns:
            c[col] = "UNKNOWN"

```

```

    c[col] = _canon_str(c[col])

# tallos_proy
if "tallos_proy" not in c.columns:
    raise ValueError("fact_ciclo_maestro no tiene 'tallos_proy' (requerido).")
c["tallos_proy"] = pd.to_numeric(c["tallos_proy"], errors="coerce").fillna(0.0)

# bloque y canónico bloque_base
if "bloque_base" not in c.columns:
    raise ValueError("fact_ciclo_maestro debe tener 'bloque_base' (llave canónica).")
c["bloque_base"] = pd.to_numeric(c["bloque_base"], errors="coerce").astype("Int64")

if "bloque" in c.columns:
    c["bloque"] = pd.to_numeric(c["bloque"], errors="coerce").astype("Int64")
else:
    c["bloque"] = c["bloque_base"]

# ---- Windows
w["ciclo_id"] = w["ciclo_id"].astype(str)
w["stage"] = _canon_str(w["stage"])
w["start_date"] = _to_date(w["start_date"])
w["end_date"] = _to_date(w["end_date"])

w = w[w["start_date"].notna() & w["end_date"].notna()].copy()
w = w[w["start_date"] <= w["end_date"]].copy()

# Tomar harvest_start/end por ciclo (desde ventana HARVEST)
hw = w[w["stage"].eq("HARVEST")][["ciclo_id", "start_date", "end_date"]].copy()
hw = hw.rename(columns={"start_date": "harvest_start", "end_date": "harvest_end_eff"})
hw = hw.drop_duplicates("ciclo_id")

# Expandir días desde windows (NO OUT)
parts = []
for r in w.itertuples(index=False):
    if pd.isna(r.start_date) or pd.isna(r.end_date):
        continue
    dates = pd.date_range(r.start_date, r.end_date, freq="D")
    parts.append(
        pd.DataFrame({"ciclo_id": r.ciclo_id, "fecha": dates, "stage": r.stage})
    )

if not parts:
    raise ValueError("No hay ventanas para expandir (w vacío).")

grid = pd.concat(parts, ignore_index=True)

# Join maestro + harvest meta
out = (
    grid.merge(c[["ciclo_id", "bloque", "bloque_base", "variedad", "tipo_sp", "area", "estado", "tallos_pr
oy"]],
              on="ciclo_id", how="left")
    .merge(hw, on="ciclo_id", how="left")
)

# Compat: bloque_padre = bloque_base
out["bloque_padre"] = out["bloque_base"]

# n_harvest_days
out["n_harvest_days"] = (out["harvest_end_eff"] - out["harvest_start"]).dt.days + 1
out["n_harvest_days"] = pd.to_numeric(out["n_harvest_days"], errors="coerce").astype("Int64")

# tallos_dia (solo si hay harvest)
out["tallos_dia"] = np.where(
    out["n_harvest_days"].notna() & (out["n_harvest_days"].astype(float) > 0),
    out["tallos_proy"].astype(float) / out["n_harvest_days"].astype(float),
    0.0,
)

# máscara stage
st = out["stage"].astype(str).str.upper()
if use_stage == "HARVEST":
    mask_stage = st.eq("HARVEST")

```

```

else:
    mask_stage = st.isin(["HARVEST", "POST"])

    # oferta: solo dentro harvest window y stage permitido
    in_window = out["harvest_start"].notna() & (out["fecha"] >= out["harvest_start"]) & (out["fecha"] <= out["harvest_end_eff"])
    out["tallos_pred"] = np.where(mask_stage & in_window, out["tallos_dia"], 0.0)

    out_pred = out[[
        "ciclo_id", "fecha",
        "bloque", "bloque_padre", "variedad", "tipo_sp", "area", "estado",
        "stage",
        "harvest_start", "harvest_end_eff", "n_harvest_days",
        "tallos_proy",
        "tallos_pred",
    ]].copy()

    out_pred["created_at"] = datetime.now().isoformat(timespec="seconds")

    out_path = preds_dir / "pred_oferta_dia.parquet"
    write_parquet(out_pred, out_path)

    print(f"OK: pred_oferta_dia={len(out_pred):,} filas -> {out_path}")
    print("Stage counts:\n", out_pred["stage"].value_counts(dropna=False).to_string())
    # KPI correcto: ceros SOLO en HARVEST
    harv = out_pred[out_pred["stage"].astype(str).str.upper().eq("HARVEST")]
    if len(harv) > 0:
        print("ratio tallos_pred==0 en HARVEST:", round((harv["tallos_pred"].fillna(0).eq(0)).mean() * 100, 2))
    print("ratio tallos_pred==0 total:", round((out_pred["tallos_pred"].fillna(0).eq(0)).mean() * 100, 2))

if __name__ == "__main__":
    main()

```

```

-----
[127/164] FILE: \src\preds\build_pred_oferta_dia_from_universe_mll.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd

```

```

from common.io import read_parquet, write_parquet

```

```

IN_GRID = Path("data/gold/universe_harvest_grid_mll.parquet")
IN_MAESTRO = Path("data/silver/fact_ciclo_maestro.parquet")

```

```

OUT = Path("data/preds/pred_oferta_dia.parquet")

```

```

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

```

```

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

```

```

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.upper().str.strip()

```

```

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Cols={list(df.columns)}")

```



```

def main() -> None:
    created_at = datetime.now().isoformat(timespec="seconds")

    if not IN_GRID.exists():
        raise FileNotFoundError(f"No existe: {IN_GRID}")
    if not IN_MAESTRO.exists():
        raise FileNotFoundError(f"No existe: {IN_MAESTRO}")

    grid = read_parquet(IN_GRID).copy()
    maestro = read_parquet(IN_MAESTRO).copy()

    # -----
    # Requisitos mínimos
    # -----
    _require(grid, ["ciclo_id", "fecha", "universe_harvest_grid_mll"])
    _require(maestro, ["ciclo_id", "tallos_proy"], "fact_ciclo_maestro")

    # -----
    # Canon
    # -----
    grid["ciclo_id"] = grid["ciclo_id"].astype(str)
    grid["fecha"] = _to_date(grid["fecha"])

    # bloque_base / variedad_canon desde grid (preferido)
    if "bloque_base" in grid.columns:
        grid["bloque_base"] = _canon_int(grid["bloque_base"])
    elif "bloque_padre" in grid.columns:
        grid["bloque_base"] = _canon_int(grid["bloque_padre"])

    if "variedad_canon" in grid.columns:
        grid["variedad_canon"] = _canon_str(grid["variedad_canon"])
    elif "variedad" in grid.columns:
        grid["variedad_canon"] = _canon_str(grid["variedad"])

    maestro["ciclo_id"] = maestro["ciclo_id"].astype(str)
    maestro["tallos_proy"] = pd.to_numeric(maestro["tallos_proy"], errors="coerce").astype(float)

    # opcionales de maestro
    for c in ["bloque_base", "bloque", "bloque_padre"]:
        if c in maestro.columns:
            maestro[c] = _canon_int(maestro[c])
    for c in ["variedad_canon", "variedad"]:
        if c in maestro.columns:
            maestro[c] = _canon_str(maestro[c])
    if "fecha_sp" in maestro.columns:
        maestro["fecha_sp"] = _to_date(maestro["fecha_sp"])
    for c in ["fecha_inicio_cosecha", "fecha_fin_cosecha"]:
        if c in maestro.columns:
            maestro[c] = _to_date(maestro[c])

    # -----
    # Dedupe hard del grid
    # -----
    key = ["ciclo_id", "fecha"]
    # Si tu grid trae bloque_base/variedad_canon, las metemos al grano final
    if "bloque_base" in grid.columns:
        key.append("bloque_base")
    if "variedad_canon" in grid.columns:
        key.append("variedad_canon")

    dup = int(grid.duplicated(subset=key).sum())
    if dup > 0:
        # en grid, duplicados deberían ser imposibles; los colapsamos por seguridad
        print(f"[WARN] universe_harvest_grid_mll duplicado por {key}; colapso. dup={dup:,}")
        agg = {}
        for c in ["harvest_start_pred", "harvest_end_pred", "n_harvest_days_pred"]:
            if c in grid.columns:
                agg[c] = "first"
        for c in ["area", "tipo_sp", "estado", "bloque", "bloque_padre"]:
            if c in grid.columns and c not in key:
                agg[c] = "first"

```

```

grid = grid.groupby(key, as_index=False).agg(agg) if agg else grid.drop_duplicates(subset=key)

# -----
# Join maestro (tallos_proy y metadatos)
# -----
m_take = ["ciclo_id", "tallos_proy"]
for c in ["bloque", "bloque_padre", "bloque_base", "variedad", "variedad_canon", "tipo_sp", "area", "estad
o"]):
    if c in maestro.columns:
        m_take.append(c)
for c in ["fecha_sp", "fecha_inicio_cosecha", "fecha_fin_cosecha"]:
    if c in maestro.columns:
        m_take.append(c)

m2 = maestro[m_take].drop_duplicates(subset=["ciclo_id"])
out = grid.merge(m2, on="ciclo_id", how="left", suffixes=("", "_m"))

# asegurar bloque_base/variedad_canon
if "bloque_base" not in out.columns and "bloque_base_m" in out.columns:
    out["bloque_base"] = out["bloque_base_m"]
if "variedad_canon" not in out.columns:
    if "variedad_canon_m" in out.columns:
        out["variedad_canon"] = out["variedad_canon_m"]
    elif "variedad_m" in out.columns:
        out["variedad_canon"] = out["variedad_m"]

# Ventana: preferimos la predicha del grid
if "harvest_start_pred" in out.columns:
    out["harvest_start"] = _to_date(out["harvest_start_pred"])
elif "fecha_inicio_cosecha" in out.columns:
    out["harvest_start"] = _to_date(out["fecha_inicio_cosecha"])
else:
    out["harvest_start"] = pd.NaT

if "harvest_end_pred" in out.columns:
    out["harvest_end_eff"] = _to_date(out["harvest_end_pred"])
elif "fecha_fin_cosecha" in out.columns:
    out["harvest_end_eff"] = _to_date(out["fecha_fin_cosecha"])
else:
    out["harvest_end_eff"] = pd.NaT

if "n_harvest_days_pred" in out.columns:
    out["n_harvest_days"] = pd.to_numeric(out["n_harvest_days_pred"], errors="coerce").astype("Int64")
else:
    # fallback por diferencia
    out["n_harvest_days"] = (out["harvest_end_eff"] - out["harvest_start"]).dt.days.add(1)
    out["n_harvest_days"] = pd.to_numeric(out["n_harvest_days"], errors="coerce").astype("Int64")

# -----
# Oferta baseline: uniform + ajuste de residuo último día
# -----
out["tallos_proy"] = pd.to_numeric(out["tallos_proy"], errors="coerce")
out["n_harvest_days"] = pd.to_numeric(out["n_harvest_days"], errors="coerce").astype("Int64")

bad = out["tallos_proy"].isna() | out["n_harvest_days"].isna() | (out["n_harvest_days"].astype(float) <= 0
)
if bad.any():
    nbad = int(bad.sum())
    print(f"[WARN] filas sin tallos_proy o n_harvest_days inválido: {nbad:,}. tallos_pred=0 en esas filas.
")
    out.loc[bad, "tallos_pred"] = 0.0

ok = ~bad
out.loc[ok, "tallos_pred"] = out.loc[ok, "tallos_proy"] / out.loc[ok, "n_harvest_days"].astype(float)

# ajuste de residuo para garantizar sum exacto por ciclo
grp = ["ciclo_id"]
out = out.sort_values(["ciclo_id", "fecha"]).reset_index(drop=True)

# marca último día por ciclo dentro del grid
out["_is_last"] = out["fecha"].eq(out.groupby("ciclo_id")["fecha"].transform("max"))

```

```

sums = out.groupby("ciclo_id", dropna=False)["tallos_pred"].transform("sum")
target = out.groupby("ciclo_id", dropna=False)["tallos_proy"].transform("max")

resid = (target - sums)
out.loc[out["_is_last"] & ok, "tallos_pred"] = out.loc[out["_is_last"] & ok, "tallos_pred"] + resid[out["_is_last"] & ok]
out = out.drop(columns=["_is_last"], errors="ignore")

# stage fijo (este dataset ES harvest-grid)
out["stage"] = "HARVEST"

# -----
# Salida final
# -----
out["created_at"] = created_at

cols = [
    "ciclo_id", "fecha",
    "bloque" if "bloque" in out.columns else None,
    "bloque_padre" if "bloque_padre" in out.columns else None,
    "bloque_base",
    "variedad" if "variedad" in out.columns else None,
    "variedad_canon",
    "tipo_sp" if "tipo_sp" in out.columns else None,
    "area" if "area" in out.columns else None,
    "estado" if "estado" in out.columns else None,
    "stage",
    "harvest_start",
    "harvest_end_eff",
    "n_harvest_days",
    "tallos_proy",
    "tallos_pred",
    "created_at",
]

cols = [c for c in cols if c is not None and c in out.columns]
out = out[cols].sort_values(["ciclo_id", "fecha", "bloque_base", "variedad_canon"]).reset_index(drop=True)

# Checks finales
key2 = ["ciclo_id", "fecha", "bloque_base", "variedad_canon"]
dup2 = int(out.duplicated(subset=key2).sum())
if dup2:
    raise ValueError(f"[FATAL] Salida tiene duplicados por {key2}: {dup2}")

cyc = out.groupby("ciclo_id", dropna=False).agg(
    proy=("tallos_proy", "max"),
    sum_pred=("tallos_pred", "sum"),
).reset_index()
cyc["abs_diff"] = (cyc["proy"] - cyc["sum_pred"]).abs()
print(f"[CHECK] ciclo mass-balance | max abs diff: {float(cyc['abs_diff'].max()):.12f}")

OUT.parent.mkdir(parents=True, exist_ok=True)
write_parquet(out, OUT)
print(f"OK -> {OUT} | rows={len(out):,} | fecha_min={out['fecha'].min().date()} fecha_max={out['fecha'].max().date()}")

if __name__ == "__main__":
    main()

```

```

-----
[128/164] FILE: \src\preds\build_pred_oferta_grado.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

```

```

from common.io import read_parquet, write_parquet

```

```

# -----
# Settings / helpers
# -----
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.strip().str.upper()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def _require(df: pd.DataFrame, cols: list[str], name: str) -> None:
    miss = [c for c in cols if c not in df.columns]
    if miss:
        raise ValueError(f"{name}: faltan columnas {miss}. Disponibles={list(df.columns)}")

# -----
# Main
# -----
def main() -> None:
    cfg = load_settings()

    preds_dir = Path(cfg.get("paths", {}).get("preds", "data/preds"))
    silver_dir = Path(cfg["paths"]["silver"])
    preds_dir.mkdir(parents=True, exist_ok=True)

    oferta_path = preds_dir / "pred_oferta_dia.parquet"
    dist_path = silver_dir / "dim_dist_grado_baseline.parquet"

    if not oferta_path.exists():
        raise FileNotFoundError(f"No existe: {oferta_path}. Ejecuta build_pred_oferta_dia primero.")
    if not dist_path.exists():
        raise FileNotFoundError(f"No existe: {dist_path}. Ejecuta build_dim_dist_grado_baseline primero.")

    oferta = read_parquet(oferta_path).copy()
    dist = read_parquet(dist_path).copy()

    # --- Validaciones mínimas
    _require(oferta, ["ciclo_id", "fecha", "variedad", "tallos_pred"], "pred_oferta_dia")
    _require(dist, ["variedad", "grado", "pct_grado"], "dim_dist_grado_baseline")

    # --- Canonicalizar
    oferta["ciclo_id"] = oferta["ciclo_id"].astype(str)
    oferta["fecha"] = _to_date(oferta["fecha"])
    oferta["variedad"] = _canon_str(oferta["variedad"])
    oferta["tallos_pred"] = pd.to_numeric(oferta["tallos_pred"], errors="coerce").fillna(0.0)

    # llaves opcionales (no forzamos, pero estandarizamos si existen)
    for col in ["bloque", "bloque_padre"]:
        if col in oferta.columns:
            oferta[col] = _canon_int(oferta[col])

    dist["variedad"] = _canon_str(dist["variedad"])
    dist["grado"] = pd.to_numeric(dist["grado"], errors="coerce").astype("Int64")
    dist["pct_grado"] = pd.to_numeric(dist["pct_grado"], errors="coerce").astype(float)

    # Mapeo de variedades (mismo estándar en oferta y dist)
    var_map = (cfg.get("mappings", {}).get("variedad_map", {}) or {})
    var_map = {str(k).strip().upper(): str(v).strip().upper() for k, v in var_map.items()}

```

```

oferta["variedad_std"] = oferta["variedad"].map(lambda x: var_map.get(x, x))
dist["variedad_std"] = dist["variedad"].map(lambda x: var_map.get(x, x))

# --- Validación: unicidad por (variedad_std, grado)
dup = int(dist.duplicated(subset=["variedad_std", "grado"]).sum())
if dup > 0:
    ex = dist.loc[
        dist.duplicated(subset=["variedad_std", "grado"], keep=False),
        ["variedad", "variedad_std", "grado", "pct_grado"],
    ].sort_values(["variedad_std", "grado"]).head(50)
    raise ValueError(
        "dim_dist_grado_baseline: duplicados por (variedad_std, grado). "
        f"dup={dup}. Ejemplos:\n{ex.to_string(index=False)}"
    )

# --- Renormalizar distribución para que sume 1 por variedad_std
# (esto evita que la masa ?se pierda? si la dim viene con redondeos o faltantes pequeños)
sum_by_var = dist.groupby("variedad_std", dropna=False)["pct_grado"].sum().reset_index(name="sum_pct")
dist = dist.merge(sum_by_var, on="variedad_std", how="left")
bad = dist["sum_pct"].isna() | (dist["sum_pct"] <= 0)
if bad.any():
    bad_vars = dist.loc[bad, "variedad_std"].dropna().unique()[:20]
    raise ValueError(f"Distribución inválida (sum_pct <= 0) para variedad_std. Ejemplos: {bad_vars}")

dist["pct_grado_norm"] = dist["pct_grado"] / dist["sum_pct"]

# --- Join oferta x grados
merged = oferta.merge(
    dist[["variedad_std", "grado", "pct_grado_norm"]],
    on="variedad_std",
    how="left",
)

# Si faltan variedades en dim => error (no silenciar)
if merged["pct_grado_norm"].isna().any():
    miss = merged.loc[merged["pct_grado_norm"].isna(), "variedad_std"].value_counts().head(20)
    raise ValueError(
        "Falta distribución por grado para algunas variedad_std. "
        "Corrige dim_dist_grado_baseline o variedad_map. Ejemplos:\n"
        f"{miss.to_string()}"
    )

merged["tallos_pred_grado"] = merged["tallos_pred"].astype(float) * merged["pct_grado_norm"].astype(float)

# --- Output (mantener compatibilidad downstream)
cols_out = [
    "ciclo_id", "fecha",
    "bloque", "bloque_padre", "variedad", "variedad_std", "tipo_sp", "area", "estado",
    "stage",
    "grado", "pct_grado_norm",
    "tallos_pred", "tallos_pred_grado",
]
# crear columnas faltantes opcionales como NA (por compatibilidad)
for c in cols_out:
    if c not in merged.columns:
        merged[c] = pd.NA

out = merged[cols_out].copy()
out = out.rename(columns={"pct_grado_norm": "pct_grado"}) # mantener nombre esperado

out["created_at"] = datetime.now().isoformat(timespec="seconds")

out_path = preds_dir / "pred_oferta_grado.parquet"
write_parquet(out, out_path)

# --- Auditoría masa: sum(tallos_pred_grado) ~= tallos_pred por ciclo/fecha
audit = (
    out.groupby(["ciclo_id", "fecha"], dropna=False)["tallos_pred_grado"].sum()
    .reset_index(name="sum_grado")
)
audit = audit.merge(
    oferta[["ciclo_id", "fecha", "tallos_pred"]],

```

```

        on=["ciclo_id", "fecha"],
        how="left"
    )
    audit["diff"] = audit["sum_grado"] - audit["tallos_pred"]
    audit["abs_diff"] = audit["diff"].abs()

    print(f"OK: pred_oferta_grado={len(out):,} filas -> {out_path}")
    print("Audit diff describe:\n", audit["diff"].describe().to_string())
    print("Audit abs_diff describe:\n", audit["abs_diff"].describe().to_string())
    print("Audit max abs_diff:", float(audit["abs_diff"].max() if len(audit) else 0.0))

if __name__ == "__main__":
    main()

-----
[129/164] FILE: \src\preds\build_pred_peso_final_ajustado_grado.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def _require_cols(df: pd.DataFrame, cols: list[str], name: str) -> None:
    missing = [c for c in cols if c not in df.columns]
    if missing:
        raise ValueError(f"{name}: faltan columnas {missing}. Disponibles={list(df.columns)}")

def main() -> None:
    cfg = load_settings()

    preds_dir = Path(cfg.get("paths", {}).get("preds", "data/preds"))
    silver_dir = Path(cfg["paths"]["silver"])
    preds_dir.mkdir(parents=True, exist_ok=True)

    in_path = preds_dir / "pred_peso_hidratado_grado.parquet"
    dim_path = silver_dir / "dim_mermas_ajuste_fecha_post_destino.parquet"

    if not in_path.exists():
        raise FileNotFoundError(f"No existe: {in_path}. Ejecuta build_pred_peso_hidratado_grado primero.")
    if not dim_path.exists():
        raise FileNotFoundError(f"No existe: {dim_path}. Ejecuta build_dim_mermas_ajuste_fecha_post primero.")

    df = read_parquet(in_path).copy()
    dim = read_parquet(dim_path).copy()

    df.columns = [str(c).strip() for c in df.columns]
    dim.columns = [str(c).strip() for c in dim.columns]

    # Requeridos mínimos
    _require_cols(df, ["fecha_post", "destino", "peso_hidratado_g"], "pred_peso_hidratado_grado")

```

```

_require_cols(dim, ["fecha_post", "destino", "factor_desp", "ajuste"], "dim_mermas_ajuste_fecha_post_desti
no")

# Normalización
df["fecha_post"] = _norm_date(df["fecha_post"])
df["destino"] = df["destino"].astype(str).str.strip().str.upper()
df["peso_hidratado_g"] = _to_num(df["peso_hidratado_g"]).fillna(0.0)

dim["fecha_post"] = _norm_date(dim["fecha_post"])
dim["destino"] = dim["destino"].astype(str).str.strip().str.upper()
dim["factor_desp"] = _to_num(dim["factor_desp"])
dim["ajuste"] = _to_num(dim["ajuste"])

dim = dim[dim["fecha_post"].notna() & dim["destino"].notna()].copy()

# --- Blindaje clave: colapsar dim a 1 fila por (fecha_post, destino)
# Si vinieran duplicados, usamos mediana robusta.
dim2 = (
    dim.groupby(["fecha_post", "destino"], dropna=False)
        .agg(
            factor_desp=("factor_desp", "median"),
            ajuste=("ajuste", "median"),
        )
        .reset_index()
)

# Merge
out = df.merge(
    dim2[["fecha_post", "destino", "factor_desp", "ajuste"]],
    on=["fecha_post", "destino"],
    how="left",
)

miss = int(out["factor_desp"].isna().sum())
if miss > 0:
    pct = miss / max(len(out), 1)
    print(f"[WARN] Sin match en dim para {miss} filas ({pct:.2%}). Se imputan medianas globales.")

# Fallbacks globales si faltan días
fd_med = float(np.nanmedian(dim2["factor_desp"].values)) if dim2["factor_desp"].notna().any() else 1.0
aj_med = float(np.nanmedian(dim2["ajuste"].values)) if dim2["ajuste"].notna().any() else 1.0
out["factor_desp"] = out["factor_desp"].fillna(fd_med)
out["ajuste"] = out["ajuste"].fillna(aj_med)

# Caps de seguridad
out["factor_desp"] = out["factor_desp"].clip(0.05, 1.0)
out["ajuste"] = out["ajuste"].clip(0.5, 2.0)

# 1) aplicar desperdicio (factor)
out["peso_post_desp_g"] = out["peso_hidratado_g"] * out["factor_desp"]

# 2) aplicar ajuste: peso_final / ajuste (tu regla)
out["peso_final_g"] = out["peso_post_desp_g"] / out["ajuste"]

out["created_at"] = datetime.now().isoformat(timespec="seconds")

# Mantener esquema actual; si falta alguna columna ?informativa?, la creamos NaN para no reventar
keep = [
    "ciclo_id",
    "fecha", "fecha_post", "dh_dias",
    "destino",
    "bloque", "bloque_padre",
    "variedad", "variedad_std",
    "tipo_sp", "area", "estado",
    "stage",
    "grado",
    "tallos_pred_grado",
    "peso_pred_g",
    "factor_hidr",
    "peso_hidratado_g",
    "factor_desp",
    "ajuste",

```

```

        "peso_post_desp_g",
        "peso_final_g",
        "created_at",
    ]
    for c in keep:
        if c not in out.columns:
            out[c] = pd.NA

    out = out[keep].copy()

    out_path = preds_dir / "pred_peso_final_ajustado_grado.parquet"
    write_parquet(out, out_path)

    print(f"OK: pred_peso_final_ajustado_grado={len(out)} filas -> {out_path}")
    print("factor_desp describe:\n", out["factor_desp"].describe().to_string())
    print("ajuste describe:\n", out["ajuste"].describe().to_string())
    print("peso_final_g describe:\n", out["peso_final_g"].describe().to_string())

if __name__ == "__main__":
    main()

-----
[130/164] FILE: \src\preds\build_pred_peso_grado.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _to_num(s: pd.Series) -> pd.Series:
    # robusto con strings y coma decimal
    ss = s.copy()
    if ss.dtype == object or pd.api.types.is_string_dtype(ss):
        ss = (
            ss.astype("string")
            .str.replace(" ", "", regex=False)
            .str.replace(",", ".", regex=False)
        )
    return pd.to_numeric(ss, errors="coerce")

def _validate_unique(df: pd.DataFrame, keys: list[str], name: str) -> None:
    dup = int(df.duplicated(subset=keys).sum())
    if dup > 0:
        ex = df.loc[df.duplicated(subset=keys, keep=False), keys].head(20)
        raise ValueError(
            f"{name}: llaves no únicas (dup={dup}) para keys={keys}. Ejemplos:\n{ex.to_string(index=False)}"
        )

def _get_use_stage(cfg: dict) -> str:
    # pred_oferta.use_stage = "HARVEST" por ejemplo
    use_stage = (cfg.get("pred_oferta", {}) or {}).get("use_stage", "")
    return str(use_stage).strip().upper()

```



```

def main() -> None:
    cfg = load_settings()

    preds_dir = Path(cfg.get("paths", {}).get("preds", "data/preds"))
    silver_dir = Path(cfg["paths"]["silver"])
    preds_dir.mkdir(parents=True, exist_ok=True)

    oferta_path = preds_dir / "pred_oferta_grado.parquet"
    peso_dim_path = silver_dir / "dim_peso_tallo_baseline.parquet"

    if not oferta_path.exists():
        raise FileNotFoundError(f"No existe: {oferta_path}. Ejecuta build_pred_oferta_grado primero.")
    if not peso_dim_path.exists():
        raise FileNotFoundError(f"No existe: {peso_dim_path}. Ejecuta build_dim_peso_tallo_baseline primero.")

    oferta = read_parquet(oferta_path).copy()
    dim = read_parquet(peso_dim_path).copy()

    oferta.columns = [str(c).strip() for c in oferta.columns]
    dim.columns = [str(c).strip() for c in dim.columns]

    # -----
    # Normalizar oferta
    # -----
    if "fecha" not in oferta.columns:
        raise ValueError("pred_oferta_grado: falta columna 'fecha'")
    oferta["fecha"] = _norm_date(oferta["fecha"])

    req = ["variedad", "grado", "tallos_pred_grado"]
    miss = [c for c in req if c not in oferta.columns]
    if miss:
        raise ValueError(f"pred_oferta_grado: faltan columnas requeridas: {miss}")

    oferta["variedad"] = oferta["variedad"].astype(str).str.strip().str.upper()
    oferta["grado"] = _to_num(oferta["grado"]).astype("Int64")
    oferta["tallos_pred_grado"] = _to_num(oferta["tallos_pred_grado"]).fillna(0.0).astype(float)

    # Mapping de variedad pred -> estándar
    var_map = (cfg.get("mappings", {}).get("variedad_map", {})) or {}
    var_map = {str(k).strip().upper(): str(v).strip().upper() for k, v in var_map.items()}
    oferta["variedad_std"] = oferta["variedad"].map(lambda x: var_map.get(x, x))

    # -----
    # FILTRO CLAVE: stage (HARVEST vs HARVEST_POST)
    # -----
    use_stage = _get_use_stage(cfg)
    if "stage" in oferta.columns:
        oferta["stage"] = oferta["stage"].astype(str).str.strip().str.upper()

        if use_stage:
            n0 = len(oferta)
            oferta = oferta[oferta["stage"].eq(use_stage)].copy()
            print(f"[INFO] pred_oferta_grado filtrado por stage='{use_stage}': {len(oferta):,}/{n0:,}")

            if oferta.empty:
                raise ValueError(
                    f"pred_oferta_grado quedó vacío tras filtrar stage='{use_stage}'. "
                    f"Revisa pred_oferta.use_stage en settings.yaml y/o valores reales en columna 'stage'."
                )
            else:
                print("[WARN] pred_oferta.use_stage vacío; se mantienen todos los stages (puede haber duplicados).")
        )
    else:
        # si no existe stage, no podemos resolver la duplicación por stage desde acá
        if use_stage:
            print("[WARN] pred_oferta.use_stage está definido, pero pred_oferta_grado NO trae columna 'stage'."
                "Se ignora el filtro. (Si hay duplicados, revisa build_pred_oferta_grado.)")

    # -----
    # Validar grano mínimo de oferta (después del filtro)

```

```

# -----
grain_keys = [k for k in ["ciclo_id", "fecha", "bloque", "bloque_padre", "variedad_std", "grado"] if k in
oferta.columns]
if len(grain_keys) >= 4:
    _validate_unique(oferta, grain_keys, "pred_oferta_grado (input)")

# -----
# Normalizar dim peso tallo
# -----
req_dim = ["variedad", "grado", "peso_tallo_mediana_g"]
miss_dim = [c for c in req_dim if c not in dim.columns]
if miss_dim:
    raise ValueError(f"dim_peso_tallo_baseline: faltan columnas requeridas: {miss_dim}")

dim["variedad"] = dim["variedad"].astype(str).str.strip().str.upper()
dim["grado"] = _to_num(dim["grado"]).astype("Int64")
dim["peso_tallo_mediana_g"] = _to_num(dim["peso_tallo_mediana_g"])

dim["variedad_std"] = dim["variedad"] # dim ya está estándar

# Colapsar a 1 fila por llave (robusto)
dim2 = (
    dim.groupby(["variedad_std", "grado"], dropna=False)
        .agg(peso_tallo_mediana_g=("peso_tallo_mediana_g", "median"))
        .reset_index()
)
_validate_unique(dim2, ["variedad_std", "grado"], "dim_peso_tallo_baseline (colapsado)")

# -----
# Join + cálculo
# -----
merged = oferta.merge(dim2, on=["variedad_std", "grado"], how="left")

miss_peso = int(merged["peso_tallo_mediana_g"].isna().sum())
if miss_peso > 0:
    ex = (
        merged.loc[merged["peso_tallo_mediana_g"].isna(), ["variedad_std", "grado"]]
            .drop_duplicates()
            .head(20)
    )
    raise ValueError(
        "Falta peso_tallo_mediana_g para algunas combinaciones variedad_std+grado. Ejemplos:\n"
        + ex.to_string(index=False)
    )

merged["peso_pred_g"] = merged["tallos_pred_grado"] * merged["peso_tallo_mediana_g"]

# columnas de salida (solo si existen)
want_cols = [
    "ciclo_id", "fecha",
    "bloque", "bloque_padre",
    "variedad", "variedad_std",
    "tipo_sp", "area", "estado",
    "stage",
    "grado",
    "tallos_pred_grado",
    "peso_tallo_mediana_g",
    "peso_pred_g",
]
out_cols = [c for c in want_cols if c in merged.columns]
out = merged[out_cols].copy()

out["created_at"] = datetime.now().isoformat(timespec="seconds")

# Validación final de unicidad (sin stage, porque ya filtraste stage)
out_keys = ["ciclo_id", "fecha", "bloque", "bloque_padre", "variedad_std", "grado"]
out_keys = [k for k in out_keys if k in out.columns]
_validate_unique(out, out_keys, "pred_peso_grado (output)")

out_path = preds_dir / "pred_peso_grado.parquet"
write_parquet(out, out_path)

```

```

print(f"OK: pred_peso_grado={len(out):,} filas -> {out_path}")
if "peso_pred_g" in out.columns:
    print("peso_pred_g describe:\n", out["peso_pred_g"].describe().to_string())

if __name__ == "__main__":
    main()

-----
[131/164] FILE: \src\preds\build_pred_peso_hidratado_grado.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def _validate_unique(df: pd.DataFrame, keys: list[str], name: str) -> None:
    dup = int(df.duplicated(subset=keys).sum())
    if dup > 0:
        ex = df.loc[df.duplicated(subset=keys, keep=False), keys].head(20)
        raise ValueError(
            f"{name}: llaves no únicas (dup={dup}) para keys={keys}. Ejemplos:\n{ex.to_string(index=False)}"
        )

def main() -> None:
    cfg = load_settings()

    preds_dir = Path(cfg.get("paths", {}).get("preds", "data/preds"))
    silver_dir = Path(cfg["paths"]["silver"])
    preds_dir.mkdir(parents=True, exist_ok=True)

    in_pred = preds_dir / "pred_peso_grado.parquet"
    dim_hidr = silver_dir / "dim_hidratacion_dh_grado_destino.parquet"
    fact_hidr = silver_dir / "fact_hidratacion_real_post_grado_destino.parquet"

    if not in_pred.exists():
        raise FileNotFoundError(f"No existe: {in_pred}. Ejecuta build_pred_peso_grado primero.")
    if not dim_hidr.exists():
        raise FileNotFoundError(f"No existe: {dim_hidr}. Ejecuta build_hidratacion_real_from_balanza2 primero.")
    ")

    pred = read_parquet(in_pred).copy()
    dim = read_parquet(dim_hidr).copy()

    pred.columns = [str(c).strip() for c in pred.columns]
    dim.columns = [str(c).strip() for c in dim.columns]

    # -----
    # Normalizar pred
    # -----
    if "fecha" not in pred.columns or "grado" not in pred.columns or "peso_pred_g" not in pred.columns:

```

```

        raise ValueError("pred_peso_grado: faltan columnas requeridas (fecha, grado, peso_pred_g)")

pred["fecha"] = _norm_date(pred["fecha"])
pred["grado"] = _to_num(pred["grado"]).astype("Int64")
pred["peso_pred_g"] = _to_num(pred["peso_pred_g"]).fillna(0.0).astype(float)

# -----
# Normalizar / colapsar dim hidratación
# -----
req = {"dh_dias", "grado", "destino", "hidr_pct", "factor_hidr"}
missing = req - set(dim.columns)
if missing:
    raise ValueError(f"dim_hidratacion_dh_grado_destino: faltan columnas {sorted(missing)}")

dim["dh_dias"] = _to_num(dim["dh_dias"]).astype("Int64")
dim["grado"] = _to_num(dim["grado"]).astype("Int64")
dim["destino"] = dim["destino"].astype(str).str.strip().str.upper()
dim["hidr_pct"] = _to_num(dim["hidr_pct"])
dim["factor_hidr"] = _to_num(dim["factor_hidr"])

# colapsar por si hay duplicados
dim2 = (
    dim.groupby(["dh_dias", "grado", "destino"], dropna=False)
        .agg(
            factor_hidr=("factor_hidr", "median"),
            hidr_pct=("hidr_pct", "median"),
        )
        .reset_index()
)
_validate_unique(dim2, ["dh_dias", "grado", "destino"], "dim_hidratacion_dh_grado_destino (colapsado)")

# -----
# Definir DH y destino default
# -----
post_cfg = cfg.get("post", {})
dh_default = int(post_cfg.get("dh_default", 7))
destino_default = str(post_cfg.get("destino_default", "APERTURA")).strip().upper()

dh_mediana = None
if fact_hidr.exists():
    fact = read_parquet(fact_hidr).copy()
    if "dh_dias" in fact.columns:
        fact["dh_dias"] = _to_num(fact["dh_dias"])
        if fact["dh_dias"].notna().any():
            dh_mediana = int(np.nanmedian(fact["dh_dias"].to_numpy()))

dh_pred = dh_mediana if dh_mediana is not None else dh_default
# clamp de seguridad para evitar locuras
dh_pred = int(np.clip(dh_pred, 0, 30))

# destino default: si no existe en dim, escoger el destino más frecuente
destinos_dim = set(dim2["destino"].dropna().unique().tolist())
if destino_default not in destinos_dim:
    # toma el modo (destino con más filas)
    destino_default = (
        dim2["destino"].value_counts(dropna=True).index[0]
        if len(dim2) else "APERTURA"
    )

# -----
# Enriquecer pred con DH/destino/fecha_post
# -----
pred["dh_dias"] = dh_pred
pred["fecha_post"] = _norm_date(pred["fecha"] + pd.to_timedelta(pred["dh_dias"], unit="D"))
pred["destino"] = destino_default

# -----
# Join hidratación por dh+grado+destino
# -----
out = pred.merge(
    dim2[["dh_dias", "grado", "destino", "hidr_pct", "factor_hidr"]],
    on=["dh_dias", "grado", "destino"],

```

```

        how="left",
    )

    # Fallback: mediana por (dh_dias, destino) ignorando grado
    if out["factor_hidr"].isna().any():
        fb = (
            dim2.groupby(["dh_dias", "destino"], dropna=False)["factor_hidr"]
                .median()
                .reset_index(name="factor_hidr_fb")
        )
        out = out.merge(fb, on=["dh_dias", "destino"], how="left")
        out["factor_hidr"] = out["factor_hidr"].fillna(out["factor_hidr_fb"])
        out["hidr_pct"] = out["hidr_pct"].fillna(out["factor_hidr"] - 1.0)
        out = out.drop(columns=["factor_hidr_fb"])

    # Fallback extremo
    out["factor_hidr"] = out["factor_hidr"].fillna(1.0).clip(0.5, 3.0)
    out["hidr_pct"] = out["hidr_pct"].fillna(out["factor_hidr"] - 1.0)

    out["peso_hidratado_g"] = out["peso_pred_g"] * out["factor_hidr"]

    out["created_at"] = datetime.now().isoformat(timespec="seconds")
    out["dh_source"] = "mediana_real" if dh_mediana is not None else "default"

    keep = [
        "ciclo_id",
        "fecha",
        "fecha_post",
        "dh_dias",
        "destino",
        "bloque", "bloque_padre",
        "variedad", "variedad_std",
        "tipo_sp", "area", "estado",
        "stage",
        "grado",
        "tallos_pred_grado",
        "peso_tallo_mediana_g",
        "peso_pred_g",
        "hidr_pct",
        "factor_hidr",
        "peso_hidratado_g",
        "dh_source",
        "created_at",
    ]
    keep = [c for c in keep if c in out.columns]
    out = out[keep].copy()

    out_path = preds_dir / "pred_peso_hidratado_grado.parquet"
    write_parquet(out, out_path)

    print(f"OK: pred_peso_hidratado_grado={len(out)} filas -> {out_path}")
    print(f"DH usado (pred): {dh_pred} dias (source={out['dh_source'].iloc[0]})")
    print(f"Destino default usado: {destino_default}")
    print("factor_hidr describe:\n", out["factor_hidr"].describe().to_string())
    print("peso_hidratado_g describe:\n", out["peso_hidratado_g"].describe().to_string())

if __name__ == "__main__":
    main()

```

```

-----
[132/164] FILE: \src\preds\build_pred_plan_horas_dia.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd
import yaml

```

```

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce")

def _norm_date(s: pd.Series) -> pd.Series:
    return _to_dt(s).dt.normalize()

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def _warn(msg: str) -> None:
    print(f"[WARN] {msg}")

def _norm_station_from_area(area: pd.Series) -> pd.Series:
    a = area.astype(str).str.upper().str.strip()
    is_a4 = a.isin(["A-4", "A4", "SJP", "SAN JUAN"])
    return pd.Series(np.where(is_a4, "A4", "MAIN"), index=area.index)

def _require_cols(df: pd.DataFrame, cols: list[str], name: str) -> None:
    missing = [c for c in cols if c not in df.columns]
    if missing:
        raise ValueError(f"{name}: faltan columnas {missing}. Disponibles={list(df.columns)}")

def _pick_col(df: pd.DataFrame, candidates: list[str], required: bool = True) -> str | None:
    cols = list(df.columns)
    cols_l = {c.lower(): c for c in cols}
    for cand in candidates:
        if cand in cols:
            return cand
        cl = cand.lower()
        if cl in cols_l:
            return cols_l[cl]
    if required:
        raise ValueError(f"No encontré columna. Candidatos={candidates}. Disponibles={cols}")
    return None

def _collapse_capacidad(cap: pd.DataFrame) -> pd.DataFrame:
    """
    Asegura grano único por (fecha, station, area_trabajada, variedad).
    Suma tallos/horas/personas; y promedia ponderado UPH por tallos_proy.
    """
    cap = cap.copy()
    keys = ["fecha", "station", "area_trabajada", "variedad"]

    dup = int(cap.duplicated(subset=keys).sum())
    if dup == 0:
        return cap

    _warn(f"capacidad_cosecha_dia: {dup} duplicados por {keys}. Se colapsará a grano único.")

def wavg_group(g: pd.DataFrame, xcol: str, wcol: str) -> float:
    x = pd.to_numeric(g[xcol], errors="coerce")
    w = pd.to_numeric(g[wcol], errors="coerce").fillna(0.0)
    m = x.notna() & w.gt(0)

```

```

        if not m.any():
            return np.nan
        return float((x[m] * w[m]).sum() / w[m].sum())

sum_cols = [c for c in ["tallos_proy", "horas_req", "personas_req"] if c in cap.columns]
first_cols = [c for c in ["uph_base_zcsp_equiv", "n_base", "n_horas"] if c in cap.columns]
agg = {c: "sum" for c in sum_cols}
agg.update({c: "first" for c in first_cols})

cap_g = cap.groupby(keys, dropna=False).agg(agg).reset_index()

if "tallos_proy" in cap.columns:
    if "uph_eff" in cap.columns:
        cap_g["uph_eff"] = cap.groupby(keys, dropna=False).apply(lambda g: wavg_group(g, "uph_eff", "tallos_proy")).to_numpy()
    if "factor_uph_dia" in cap.columns:
        cap_g["factor_uph_dia"] = cap.groupby(keys, dropna=False).apply(lambda g: wavg_group(g, "factor_uph_dia", "tallos_proy")).to_numpy()
    else:
        if "uph_eff" in cap.columns:
            cap_g["uph_eff"] = cap.groupby(keys, dropna=False)["uph_eff"].median().to_numpy()
        if "factor_uph_dia" in cap.columns:
            cap_g["factor_uph_dia"] = cap.groupby(keys, dropna=False)["factor_uph_dia"].median().to_numpy()

if "created_at" in cap.columns:
    cap_g["created_at"] = cap.groupby(keys, dropna=False)["created_at"].max().to_numpy()

dup2 = int(cap_g.duplicated(subset=keys).sum())
if dup2 > 0:
    ex = cap_g[cap_g.duplicated(subset=keys, keep=False)].head(20)
    raise ValueError(f"Aún hay {dup2} duplicados en capacidad después de colapsar. Ejemplos:\n{ex.to_string(index=False)}")

return cap_g

def main() -> None:
    cfg = load_settings()
    preds_dir = Path(cfg["paths"]["preds"])
    preds_dir.mkdir(parents=True, exist_ok=True)

    path_cap_cosecha = preds_dir / "capacidad_cosecha_dia.parquet"
    path_pos = preds_dir / "pred_horas_poscosecha_dia.parquet"
    path_peso_grado = preds_dir / "pred_peso_grado.parquet"

    for p in [path_cap_cosecha, path_pos, path_peso_grado]:
        if not p.exists():
            raise FileNotFoundError(f"No existe input requerido: {p}")

    cap = pd.read_parquet(path_cap_cosecha)
    pos = pd.read_parquet(path_pos)
    pg = pd.read_parquet(path_peso_grado)

    cap.columns = [str(c).strip() for c in cap.columns]
    pos.columns = [str(c).strip() for c in pos.columns]
    pg.columns = [str(c).strip() for c in pg.columns]

    # -----
    # COSECHA (capacidad)
    # -----
    _require_cols(cap, ["fecha", "station", "variedad", "tallos_proy", "uph_eff", "horas_req", "personas_req"],
, "capacidad_cosecha_dia")

    col_area_cap = _pick_col(cap, ["area_trabajada", "area", "Area"], required=False)
    if col_area_cap is None:
        raise ValueError("capacidad_cosecha_dia no trae area_trabajada/area; ajusta build_capacidad_cosecha_dia.py para incluirla.")

    cap = cap.copy()
    cap["fecha"] = _norm_date(cap["fecha"])
    cap["station"] = cap["station"].astype(str).str.upper().str.strip()
    cap["variedad"] = cap["variedad"].astype(str).str.upper().str.strip()

```

```

cap["area_trabajada"] = cap[col_area_cap].astype(str).str.upper().str.strip()
cap = cap[cap["fecha"].notna()].copy()

# asegurar llaves sin nulos críticos
cap["area_trabajada"] = cap["area_trabajada"].replace({"": "ALL"}).fillna("ALL")
cap["variedad"] = cap["variedad"].replace({"": "ALL"}).fillna("ALL")

cap = _collapse_capacidad(cap)
plan_cosecha = cap.copy()
plan_cosecha["tipo_flujo"] = "COSECHA"

# -----
# POSCOSECHA (diario)
# -----
_require_cols(pos, ["fecha_post", "kg_total", "horas_req_total", "personas_req_total"], "pred_horas_poscos
echa_dia")

pos = pos.copy()
pos["fecha"] = _norm_date(pos["fecha_post"])
pos = pos[pos["fecha"].notna()].copy()

keep_pos = [
    "fecha",
    "kg_total", "cajas_total",
    "W_Blanco", "W_Arcoiris", "W_Tinturado",
    "horas_req_blanco", "horas_req_arcoiris", "horas_req_tinturado",
    "horas_req_total",
    "personas_req_blanco", "personas_req_arcoiris", "personas_req_tinturado",
    "personas_req_total",
]
keep_pos = [c for c in keep_pos if c in pos.columns]
pos2 = pos[keep_pos].copy()
pos2["station"] = "PLANTA"
pos2["area_trabajada"] = "ALL"
pos2["variedad"] = "ALL"
pos2["tipo_flujo"] = "POSCOSECHA"

# -----
# pred_peso_grado (resumen por llave de cosecha)
# -----
_require_cols(pg, ["fecha", "tallos_pred_grado", "peso_pred_g"], "pred_peso_grado")

pg = pg.copy()
pg["fecha"] = _norm_date(pg["fecha"])
pg = pg[pg["fecha"].notna()].copy()

# variedad: preferir variedad_std, si no existe usar variedad
col_var = _pick_col(pg, ["variedad_std", "variedad"], required=True)
pg["variedad_std"] = (
    pg[col_var].astype(str).str.upper().str.strip()
    .replace({"XLENCE": "XL", "CLOUD": "CLO"})
)

# station/area_trabajada: preferir station/area_trabajada si existen
if "station" in pg.columns:
    pg["station"] = pg["station"].astype(str).str.upper().str.strip().replace({"A-4": "A4", "SJP": "A4"})
elif "area" in pg.columns:
    pg["station"] = _norm_station_from_area(pg["area"])
else:
    pg["station"] = "MAIN"

if "area_trabajada" in pg.columns:
    pg["area_trabajada"] = pg["area_trabajada"].astype(str).str.upper().str.strip()
elif "area" in pg.columns:
    pg["area_trabajada"] = pg["area"].astype(str).str.upper().str.strip()
else:
    pg["area_trabajada"] = "ALL"

pg["area_trabajada"] = pg["area_trabajada"].replace({"": "ALL"}).fillna("ALL")

pg["tallos_pred_grado"] = _to_num(pg["tallos_pred_grado"]).fillna(0.0).astype(float)
pg["peso_pred_g"] = _to_num(pg["peso_pred_g"]).astype(float)

```



```

# promedio ponderado robusto: sum(peso * tallos) / sum(tallos)
pg["peso_x_tallos"] = pg["peso_pred_g"] * pg["tallos_pred_grado"]

pg_agg = (
    pg.groupby(["fecha", "station", "area_trabajada", "variedad_std"], dropna=False)
    .agg(
        tallos_pred_total=("tallos_pred_grado", "sum"),
        peso_x_tallos_sum=("peso_x_tallos", "sum"),
        n_grados=("tallos_pred_grado", "size"),
    )
    .reset_index()
    .rename(columns={"variedad_std": "variedad"})
)

pg_agg["peso_pred_prom_g"] = np.where(
    pg_agg["tallos_pred_total"] > 0,
    pg_agg["peso_x_tallos_sum"] / pg_agg["tallos_pred_total"],
    np.nan,
)
pg_agg = pg_agg.drop(columns=["peso_x_tallos_sum"])

dup_pg = int(pg_agg.duplicated(subset=["fecha", "station", "area_trabajada", "variedad"]).sum())
if dup_pg > 0:
    ex = pg_agg[pg_agg.duplicated(subset=["fecha", "station", "area_trabajada", "variedad"], keep=False)].
head(20)
    raise ValueError(f"pred_peso_grado agregado NO es único por llave (dup={dup_pg}). Ejemplos:\n{ex.to_string(index=False)}")

# merge cosecha + resumen peso
plan_cosecha = plan_cosecha.merge(
    pg_agg,
    on=["fecha", "station", "area_trabajada", "variedad"],
    how="left",
    validate="one_to_one",
)

# -----
# Unión final
# -----
common_cols = [
    "fecha", "station", "area_trabajada", "variedad", "tipo_flujo",
    "tallos_pred_total", "peso_pred_prom_g", "n_grados",
    "tallos_proy", "uph_base_zcsp_equiv", "factor_uph_dia", "uph_eff",
    "horas_req", "personas_req", "n_base", "n_horas",
    "kg_total", "cajas_total",
    "W_Blanco", "W_Arcoiris", "W_Tinturado",
    "horas_req_blanco", "horas_req_arcoiris", "horas_req_tinturado", "horas_req_total",
    "personas_req_blanco", "personas_req_arcoiris", "personas_req_tinturado", "personas_req_total",
]

for c in common_cols:
    if c not in plan_cosecha.columns:
        plan_cosecha[c] = np.nan
    if c not in pos2.columns:
        pos2[c] = np.nan

out = pd.concat([plan_cosecha[common_cols], pos2[common_cols]], ignore_index=True)
out["created_at"] = datetime.now().isoformat(timespec="seconds")

# -----
# Validaciones / warnings
# -----
dup_out = int(out.duplicated(subset=["fecha", "station", "area_trabajada", "variedad", "tipo_flujo"]).sum())
if dup_out > 0:
    warn(f"Hay {dup_out} duplicados por (fecha,station,area,variedad,tipo_flujo). Revisa upstream.")

miss_pg = int((out["tipo_flujo"].eq("COSECHA") & out["tallos_pred_total"].isna()).sum())
if miss_pg > 0:
    warn(f"COSECHA: {miss_pg} filas sin match en pred_peso_grado (tallos_pred_total NaN).")

```

```

bad_a4_clo = out[(out["tipo_flujo"] == "COSECHA") & (out["station"] == "A4") & (out["variedad"] == "CLO")]
if len(bad_a4_clo) > 0:
    _warn(f"[DATA QUALITY] Detecté {len(bad_a4_clo)} filas COSECHA con A4+CLO (no esperado).")

out_path = preds_dir / "pred_plan_horas_dia.parquet"
write_parquet(out, out_path)

_info(f"OK: pred_plan_horas_dia={len(out)} filas -> {out_path}")
_info(f"Rango fechas: {out['fecha'].min()} -> {out['fecha'].max()}")
_info(f"Flujos: {out['tipo_flujo'].value_counts(dropna=False).to_dict()}")
_info(f"Stations: {out['station'].value_counts(dropna=False).to_dict()}")
_info(
    "Áreas (COSECHA, top): "
    f"{out[out['tipo_flujo']=='COSECHA']['area_trabajada'].value_counts().head(10).to_dict()}"
)

if __name__ == "__main__":
    main()

-----
[133/164] FILE: \src\preds\build_pred_tallos_cosecha_dia.py
-----
# src/preds/build_pred_tallos_cosecha_dia.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

# OJO: para aislar problemas, escribimos con pandas.to_parquet (no con write_parquet)

# -----
# Config / helpers
# -----
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def _warn(msg: str) -> None:
    print(f"[WARN] {msg}")

def _pick_first_existing(paths: list[Path]) -> Path | None:
    for p in paths:
        if p.exists():
            return p
    return None

def _std_variedad(v: pd.Series) -> pd.Series:
    x = v.astype(str).str.upper().str.strip()
    return x.replace({"XXLENCE": "XL", "XLENCE": "XL", "CLOUD": "CLO"})

```

```

def _infer_station_from_area(area: pd.Series) -> pd.Series:
    """
    Regla negocio (clima):
    - A-4 / A4 / SJP / SAN JUAN => A4
    - resto => MAIN
    """
    a = area.astype(str).str.upper().str.strip()
    is_a4 = a.isin(["A-4", "A4", "SJP", "SAN JUAN"])
    return pd.Series(np.where(is_a4, "A4", "MAIN"), index=area.index)

def _bloque_key_from_any(df: pd.DataFrame) -> pd.Series | None:
    """
    Intenta inferir bloque_key si existe alguna columna compatible.
    """
    for c in ["bloque", "Bloque", "bloque_norm", "Bloque_norm", "bloque_padre", "block", "Block"]:
        if c in df.columns:
            s = df[c].astype(str).str.strip()
            k = s.str.extract(r"^(\\d+)", expand=False)
            return pd.to_numeric(k, errors="coerce").astype("Int64")
    return None

def _ensure_area_trabajada(
    df: pd.DataFrame,
    preds_dir: Path,
) -> pd.DataFrame:
    """
    Asegura area_trabajada:
    1) Si viene en el insumo (area_trabajada/area/Area) => usarla
    2) Si no viene => intenta mapear desde pred_peso_grado.parquet por (fecha, bloque_key, variedad)
    3) Si no se puede => ALL (warning)
    """
    df = df.copy()

    # 1) directa
    for cand in ["area_trabajada", "area", "Area"]:
        if cand in df.columns:
            df["area_trabajada"] = df[cand].astype(str).str.upper().str.strip()
            return df

    # 2) mapping desde pred_peso_grado
    pg_path = preds_dir / "pred_peso_grado.parquet"
    if not pg_path.exists():
        _warn("No existe pred_peso_grado.parquet para mapear área -> se asigna ALL.")
        df["area_trabajada"] = "ALL"
        return df

    bloque_key = _bloque_key_from_any(df)
    if bloque_key is None:
        _warn("El insumo no trae bloque/bloque_norm para mapear área -> se asigna ALL.")
        df["area_trabajada"] = "ALL"
        return df

    df["bloque_key"] = bloque_key

    pg = pd.read_parquet(pg_path)
    pg.columns = [str(c).strip() for c in pg.columns]

    # checks mínimos
    req = {"fecha", "bloque", "area"}
    miss = req - set(pg.columns)
    if miss:
        raise ValueError(f"pred_peso_grado.parquet no tiene columnas requeridas {sorted(miss)}. Columnas={list(pg.columns)}")
    if ("variedad_std" not in pg.columns) and ("variedad" not in pg.columns):
        raise ValueError(f"pred_peso_grado.parquet: falta variedad_std/variedad. Columnas={list(pg.columns)}")

    pg["fecha"] = _norm_dt(pg["fecha"])
    pg["bloque_key"] = pd.to_numeric(pg["bloque"].astype(str).str.extract(r"^(\\d+)", expand=False), errors="coerce").astype("Int64")

```

```

if "variedad_std" in pg.columns:
    pg["variedad"] = _std_variedad(pg["variedad_std"])
else:
    pg["variedad"] = _std_variedad(pg["variedad"])

pg["area_trabajada"] = pg["area"].astype(str).str.upper().str.strip()

# mapa (fecha,bloque_key,variedad) -> area_trabajada (mode)
map_area = (
    pg.dropna(subset=["fecha", "bloque_key", "variedad", "area_trabajada"])
    .groupby(["fecha", "bloque_key", "variedad"], dropna=False)["area_trabajada"]
    .agg(lambda s: s.mode().iat[0] if len(s.mode()) else s.iloc[0])
    .reset_index()
)

# join
if "fecha" not in df.columns:
    raise ValueError("No puedo mapear área: df no tiene columna 'fecha' normalizada.")

if "variedad" not in df.columns:
    raise ValueError("No puedo mapear área: df no tiene columna 'variedad' estandarizada.")

df = df.merge(map_area, on=["fecha", "bloque_key", "variedad"], how="left")

miss_n = int(df["area_trabajada"].isna().sum())
if miss_n > 0:
    _warn(f"No se pudo mapear area_trabajada para {miss_n} filas -> se asigna ALL.")
    df["area_trabajada"] = df["area_trabajada"].fillna("ALL")

# limpieza
df = df.drop(columns=["bloque_key"])
df["area_trabajada"] = df["area_trabajada"].astype(str).str.upper().str.strip()
return df

def _validate_unique(df: pd.DataFrame, keys: list[str], name: str) -> None:
    d = df.duplicated(subset=keys).sum()
    if int(d) > 0:
        ex = df.loc[df.duplicated(subset=keys, keep=False), keys].head(20)
        raise ValueError(f"{name}: llaves no únicas dup={int(d)} keys={keys}. Ejemplos:\n{ex.to_string(index=False)}")

# -----
# Main
# -----
def main() -> None:
    cfg = load_settings()

    preds_dir = Path(cfg["paths"]["preds"])
    preds_dir.mkdir(parents=True, exist_ok=True)

    # --- DEBUG anti-?no actualiza? ---
    _info(f"running file: {__file__}")
    _info(f"cwd: {Path.cwd()}")
    _info(f"preds_dir: {preds_dir.resolve()}")

    # Insumos (semillas)
    candidates = [
        preds_dir / "pred_oferta_grado.parquet",
        preds_dir / "pred_oferta_dia.parquet",
    ]
    src_path = _pick_first_existing(candidates)
    if src_path is None:
        raise FileNotFoundError(
            "No encontré insumo para construir pred_tallos_cosecha_dia.\n"
            f"Esperaba alguno de:\n- {candidates[0]}\n- {candidates[1]}\n"
            "Ejecuta primero el script que genera pred_oferta_grado o pred_oferta_dia."
        )
    _info(f"source selected: {src_path.resolve()}")

    df = pd.read_parquet(src_path)

```

```

df.columns = [str(c).strip() for c in df.columns]
_info(f"input rows: {len(df)} cols: {len(df.columns)}")

# ----- Fecha -----
if "fecha" in df.columns:
    df["fecha"] = _norm_dt(df["fecha"])
elif "Fecha" in df.columns:
    df["fecha"] = _norm_dt(df["Fecha"])
elif "Fecha_Cosecha" in df.columns:
    df["fecha"] = _norm_dt(df["Fecha_Cosecha"])
elif "harvest_start" in df.columns:
    df["fecha"] = _norm_dt(df["harvest_start"])
else:
    raise ValueError(f"No pude inferir columna de fecha en {src_path.name}. Columnas={list(df.columns)[:80]}")

df = df[df["fecha"].notna()].copy()

# ----- Tallos -----
# ----- Tallos (fix: evitar duplicación por grado) -----
src_name = src_path.name.lower()

if "pred_oferta_grado" in src_name:
    if "tallos_pred_grado" not in df.columns:
        raise ValueError(
            f"Insumo pred_oferta_grado.parquet requiere columna 'tallos_pred_grado'. "
            f"Columnas disponibles: {list(df.columns)}"
        )
    df["tallos_pred"] = (
        _to_num(df["tallos_pred_grado"])
        .fillna(0.0)
        .astype(float)
    )
    _info("Tallos fuente: tallos_pred_grado (pred_oferta_grado)")
else:
    # comportamiento original para oferta diaria
    if "tallos_proy" in df.columns:
        df["tallos_pred"] = _to_num(df["tallos_proy"]).fillna(0.0).astype(float)
    elif "tallos_pred" in df.columns:
        df["tallos_pred"] = _to_num(df["tallos_pred"]).fillna(0.0).astype(float)
    elif "tallos" in df.columns:
        df["tallos_pred"] = _to_num(df["tallos"]).fillna(0.0).astype(float)
    elif "Tallos" in df.columns:
        df["tallos_pred"] = _to_num(df["Tallos"]).fillna(0.0).astype(float)
    else:
        raise ValueError(
            f"No pude inferir columna de tallos en {src_path.name}. "
            f"Columnas={list(df.columns)}"
        )
    _info("Tallos fuente: total diario (pred_oferta_dia)")

# ----- Variedad (semilla: XL/CLO) -----
if "variedad" in df.columns:
    df["variedad"] = _std_variedad(df["variedad"])
elif "Variedad" in df.columns:
    df["variedad"] = _std_variedad(df["Variedad"])
elif "variedad_std" in df.columns:
    df["variedad"] = _std_variedad(df["variedad_std"])
else:
    df["variedad"] = "UNKNOWN"

# Normalización final esperada
df["variedad"] = df["variedad"].replace({"XLENCE": "XL", "CLOUD": "CLO"})
# NO filtramos a XL/CLO aquí obligatoriamente; pero avisamos
if int((df["variedad"] == "UNKNOWN").sum()) > 0:
    _warn("Hay filas con variedad=UNKNOWN. Revisa el insumo si debería traer variedad.")

# ----- Área canónica (area_trabajada) -----
df = _ensure_area_trabajada(df, preds_dir=preds_dir)

# ----- Station canónica (para clima y joins posteriores) -----
df["station"] = _infer_station_from_area(df["area_trabajada"])

```

```

# ----- Sellado de grano (nivel correcto para capacidad) -----
out = (
    df.groupby(["fecha", "station", "area_trabajada", "variedad"], dropna=False)
        .agg(tallos_proy=("tallos_pred", "sum"))
        .reset_index()
)
out["tallos_proy"] = _to_num(out["tallos_proy"]).fillna(0.0).clip(lower=0.0).astype(float)

# Validación unicidad (debe ser 1 fila por llave)
_validate_unique(out, ["fecha", "station", "area_trabajada", "variedad"], "pred_tallos_cosecha_dia")

out["created_at"] = datetime.now().isoformat(timespec="seconds")

# ----- Escritura robusta (sin helper) -----
out_path = preds_dir / "pred_tallos_cosecha_dia.parquet"
_info(f"out_path: {out_path.resolve()}")

before = datetime.fromtimestamp(out_path.stat().st_mtime) if out_path.exists() else None
if before:
    _info(f"mtime BEFORE: {before.isoformat(timespec='seconds')}")

out.to_parquet(out_path, index=False)

after = datetime.fromtimestamp(out_path.stat().st_mtime) if out_path.exists() else None
if after:
    _info(f"mtime AFTER: {after.isoformat(timespec='seconds')}")

# ----- Resumen -----
_info(f"OK: pred_tallos_cosecha_dia={len(out)} filas -> {out_path.name}")
_info(f"Rango fechas: {out['fecha'].min()} -> {out['fecha'].max()}")
_info(f"Stations: {out['station'].value_counts(dropna=False).to_dict()}")
_info(f"Areas (top): {out['area_trabajada'].value_counts(dropna=False).head(15).to_dict()}")
_info(f"Variedades: {out['variedad'].value_counts(dropna=False).head(10).to_dict()}")

n_all = int((out["area_trabajada"] == "ALL").sum())
if n_all > 0:
    _warn(f"Hay {n_all} filas con area_trabajada=ALL (no se pudo inferir área).")

if __name__ == "__main__":
    main()

-----
[134/164] FILE: \src\run_pipeline.py
-----
from __future__ import annotations

import argparse
from pathlib import Path

from ops.io import load_settings
from ops.registry import build_registry
from ops.runner import run_step
from ops.validators import validate_exists, validate_parquet_nonempty

LAYERS_ORDER = ["bronze", "silver", "preds", "features", "models", "gold", "audit", "ml2"]

def parse_args() -> argparse.Namespace:
    ap = argparse.ArgumentParser("PLANIFICACION MACRO - OPS (src layout)")

    ap.add_argument("--settings", default="config/settings.yaml")

    ap.add_argument(
        "--only",
        choices=LAYERS_ORDER,
        default=None,
    )
    ap.add_argument(

```

```

        "--from",
        dest="from_layer",
        choices=LAYERS_ORDER,
        default=None,
    )
    ap.add_argument(
        "--until",
        dest="until_layer",
        choices=LAYERS_ORDER,
        default=None,
    )
    ap.add_argument("--step", default=None, help="Ejecutar un step por nombre exacto")

    ap.add_argument("--force", action="store_true", help="Re-ejecuta aunque existan outputs")
    ap.add_argument("--dry-run", action="store_true")
    ap.add_argument("--fail-fast", action="store_true")
    ap.add_argument("--validate", action="store_true")

    return ap.parse_args()

def filter_steps(steps, only, from_layer, until_layer, step_name):
    if step_name:
        return [s for s in steps if s.name == step_name]

    if only:
        return [s for s in steps if s.layer == only]

    if from_layer or until_layer:
        i0 = LAYERS_ORDER.index(from_layer) if from_layer else 0
        i1 = LAYERS_ORDER.index(until_layer) if until_layer else (len(LAYERS_ORDER) - 1)
        allowed = set(LAYERS_ORDER[i0 : i1 + 1])
        return [s for s in steps if s.layer in allowed]

    return steps

def main():
    args = parse_args()
    settings = load_settings(args.settings)

    # repo root = padre de /src
    repo_root = Path(__file__).resolve().parents[1]

    python_exe_cfg = settings.get("ops", {}).get("python", "venv/Scripts/python.exe")
    python_exe = str((repo_root / python_exe_cfg).resolve())

    artifacts_dir = (repo_root / settings.get("ops", {}).get("artifacts_dir", "data")).resolve()
    log_dir = (repo_root / settings.get("ops", {}).get("log_dir", "logs") / "ops").resolve()

    steps = filter_steps(build_registry(), args.only, args.from_layer, args.until_layer, args.step)
    if not steps:
        raise SystemExit("No steps selected.")

    print(f"repo_root={repo_root}")
    print(f"python={python_exe}")
    print(f"artifacts_dir={artifacts_dir}")
    print(f"log_dir={log_dir}")
    print(f"steps={len(steps)}")

    ok_all = True

    for s in steps:
        print(f"\n==> {s.layer}:{s.name}")

        res = run_step(
            step=s,
            python_exe=python_exe,
            repo_root=repo_root,
            artifacts_dir=artifacts_dir,
            log_dir=log_dir,
            force=args.force,

```

```

        dry_run=args.dry_run,
    )

    if not res.ok:
        ok_all = False
        print(f"    FAILED rc={res.returncode}")
        print(f"    logs:\n    - {res.stdout_path}\n    - {res.stderr_path}")
        if args.fail_fast:
            break
        continue

    if res.seconds > 0:
        print(f"    OK {res.seconds:.1f}s")
    else:
        print("    dry-run" if args.dry_run else "    skipped (outputs exist)")

    if args.validate and s.outputs_rel:
        outputs_abs = [(artifacts_dir / o).resolve() for o in s.outputs_rel]
        try:
            validate_exists(outputs_abs)
            validate_parquet_nonempty(outputs_abs)
            print("    validated OK")
        except Exception as e:
            ok_all = False
            print(f"    VALIDATION FAILED: {e}")
            if args.fail_fast:
                break

    if not ok_all:
        raise SystemExit(1)

    print("\nOPS finished OK")

```

```

if __name__ == "__main__":
    main()

```

```

-----
[135/164] FILE: \src\silver\build_ciclo_maestro_from_fenograma.py
-----

```

```

# src/silver/build_ciclo_maestro_from_fenograma.py
from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import re
import pandas as pd
import numpy as np
import yaml

```

```

from common.io import write_parquet
from common.ids import make_bloque_id, make_variedad_id, make_ciclo_id
from common.timegrid import build_grid_ciclo_fecha

```

```

# -----
# Helpers
# -----
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

```

```

def normalizar_columnas(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df.columns = [str(c).strip() for c in df.columns]
    return df

```

```

def to_datetime_safe(x):
    return pd.to_datetime(x, errors="coerce")

```



```

def to_numeric_safe(x):
    return pd.to_numeric(x, errors="coerce")

def normalize_date_col(df: pd.DataFrame, col: str):
    df = df.copy()
    df[col] = to_datetime_safe(df[col]).dt.normalize()
    return df

def reemplazos_area(series: pd.Series) -> pd.Series:
    return series.astype(str).replace({"SSJ": "A-4", "MM1": "MH1", "MM2": "MH2"})

# -----
# BRONZE RAW readers (indices: col_0..col_n)
# -----
def _strip_cell(x) -> str:
    if x is None or (isinstance(x, float) and np.isnan(x)):
        return ""
    return str(x).strip()

def _raw_has_col0(df: pd.DataFrame) -> bool:
    return any(str(c).strip().lower() == "col_0" for c in df.columns)

def _ensure_cols_are_strings(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df.columns = [str(c) for c in df.columns]
    return df

def _rebuild_header_like_xl(raw: pd.DataFrame) -> pd.DataFrame:
    """
    Replica el patrón de tu fenograma_historia_xl() original:
    raw = raw[~raw[0].isna() & (raw[0].astype(str).str.strip() != "")].copy()
    raw = raw.iloc[1:].copy()
    raw.columns = raw.iloc[0]
    df = raw.iloc[1:].copy()

    En Bronze: raw viene como col_0..col_n (todo string).
    """
    raw = _ensure_cols_are_strings(raw)

    col0 = "col_0" if "col_0" in raw.columns else raw.columns[0]
    rr = raw.copy()

    mask = rr[col0].notna() & (rr[col0].astype(str).str.strip() != "")
    rr = rr[mask].copy()

    # drop first row
    if len(rr) < 2:
        return pd.DataFrame()

    rr = rr.iloc[1:].copy()

    # header row is first row now
    header = rr.iloc[0].tolist()
    header = [re.sub(r"\s+", " ", _strip_cell(h).replace("\n", " ").replace("\r", " ")).strip() for h in header]

    rr = rr.iloc[1:].copy()
    rr.columns = header

    rr = normalizar_columnas(rr)
    return rr

def _rebuild_header_like_clo(raw: pd.DataFrame) -> pd.DataFrame:
    """

```

```

Replica el patrón de tu fenograma_historia_clo() original:
raw = raw[~raw[0].isna() & (raw[0].astype(str).str.strip() != "")].copy()
raw.columns = raw.iloc[0]
df = raw.iloc[1:].copy()

En Bronze: raw viene como col_0..col_n (todo string).
"""
raw = _ensure_cols_are_strings(raw)
col0 = "col_0" if "col_0" in raw.columns else raw.columns[0]
rr = raw.copy()

mask = rr[col0].notna() & (rr[col0].astype(str).str.strip() != "")
rr = rr[mask].copy()

if len(rr) < 2:
    return pd.DataFrame()

header = rr.iloc[0].tolist()
header = [re.sub(r"\s+", " ", _strip_cell(h).replace("\n", " ").replace("\r", " ")).strip() for h in header]

rr = rr.iloc[1:].copy()
rr.columns = header

rr = normalizar_columnas(rr)
return rr

# -----
# Fenograma activo (tu lógica)
# -----
def fenograma_activo(df_fenograma_xlsm: pd.DataFrame, bal: pd.DataFrame, hoy: pd.Timestamp) -> pd.DataFrame:
    df = normalizar_columnas(df_fenograma_xlsm)

    if "Area " in df.columns and "Area" not in df.columns:
        df = df.rename(columns={"Area ": "Area"})

    df["Bloques"] = df["Bloques"].astype(str)

    valores_excluir = {
        None, 0, 500, 625, 750, 875, 1000, 6025,
        "%1000 GR", "%350 GR", "%500 GR", "%750 GR",
        "00 a 20", "21 a 40", "41 a 60", "61 a 80", "81 a 100",
        "ANDES", "Cajas por cosechar entre:", "CAMPO", "CLOUD", "CV",
        "Distribucion de cosecha en el campo", "DISTRIBUCION POR GRADOS",
        "DOMINGO", "FLOR GRANDE = FG", "FLOR PEQUEÑA = FP", "GENERAL",
        "JUEVES", "LUNES", "MARTES", "MH1", "MH2", "MIERCOLES",
        "MILLION CLOUD", "MM1", "MM2", "OTRO",
        "PROYECCION CAJAS POR GRADOS",
        "SABADO", "SI", "SJP", "SSJ", "Tallos/caja",
        "TOTAL", "Total", "Total cajas", "TOTAL GYPSOS",
        "Total tallos", "VENTAS", "VIERNES", "x",
        "XLENCE", "XLENCE FIVE STARS"
    }

    if "Fiesta" in df.columns:
        df = df[~df["Fiesta"].isin(valores_excluir)].copy()

    df = df[df["Bloques"].notna()].copy()

    cols = ["Bloques", "Fecha S/P", "Area", "S/P", "Variedad", "Tallos / Bloque"]
    missing = [c for c in cols if c not in df.columns]
    if missing:
        raise ValueError(f"fenograma_activo: faltan columnas esperadas en fenograma_xlsm_raw: {missing}")

    df = df[cols].copy()
    df["Fecha S/P"] = to_datetime_safe(df["Fecha S/P"])
    df["Tallos / Bloque"] = to_numeric_safe(df["Tallos / Bloque"]).fillna(0)

    agg = (
        df.groupby(["Bloques", "S/P", "Variedad", "Area"], dropna=False)
        .agg(**{
            "Fecha S/P": ("Fecha S/P", "max"),

```

```

        "Tallos_Proys": ("Tallos / Bloque", "sum")
    })
    .reset_index()
)

agg = agg[agg["Area"].notna()].copy()
agg = agg[(agg["Area"] != 0) & (agg["Area"] != "0")].copy()
agg["Area"] = reemplazos_area(agg["Area"])

# Join con balanza para calcular días vegetativos
tmp = agg.merge(bal, left_on="Bloques", right_on="Bloque", how="left").drop(columns=["Bloque"])
tmp["Personalizado"] = (tmp["Fecha"] - tmp["Fecha S/P"]).dt.days

tmp_f = tmp[(tmp["Personalizado"] > 30) | (tmp["Personalizado"].isna())].copy()

g = (
    tmp_f.groupby(["Bloques", "S/P", "Fecha S/P"], dropna=False)
        .agg(
            Dias_Vegetativo=("Personalizado", "min"),
            Dias_Vegetativo_1=("Personalizado", "max")
        )
        .reset_index()
)

g["Fecha_Inicio_Cosecha (Primer Tallo)"] = g["Fecha S/P"] + pd.to_timedelta(g["Dias_Vegetativo"], unit="D")
)
g.loc[g["Dias_Vegetativo"].isna(), "Fecha_Inicio_Cosecha (Primer Tallo)"] = pd.NaT

fin_calc = g["Fecha S/P"] + pd.to_timedelta(g["Dias_Vegetativo_1"], unit="D")
g["Fecha_Fin_Cosecha"] = fin_calc

cond1 = g["Dias_Vegetativo"].isna()
cond2 = g["Dias_Vegetativo_1"] <= g["Dias_Vegetativo"]
cond3 = fin_calc >= (hoy - pd.Timedelta(days=4))
g.loc[cond1 | cond2 | cond3, "Fecha_Fin_Cosecha"] = pd.NaT

base = agg.merge(
    g[["Bloques", "Fecha S/P", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha"]],
    on=["Bloques", "Fecha S/P"],
    how="left"
)

base = base[base["Fecha S/P"] < hoy].copy()
base["Estado"] = "ACTIVO"

base = normalize_date_col(base, "Fecha S/P")
base = normalize_date_col(base, "Fecha_Inicio_Cosecha (Primer Tallo)")
base = normalize_date_col(base, "Fecha_Fin_Cosecha")

return base[[
    "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
    "Tallos_Proys", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
]]

# -----
# Históricos XL/CLO (desde BRONZE raw indices*.parquet)
# -----
def fenograma_historia_xl(raw_indices_xl: pd.DataFrame, fecha_min_hist: pd.Timestamp) -> pd.DataFrame:
    df = _rebuild_header_like_xl(raw_indices_xl)
    if df.empty:
        return pd.DataFrame(columns=[
            "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
            "Tallos_Proys", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
        ])

    # Filtros Pruebas (idéntico a tu lógica)
    if "Pruebas" in df.columns:
        df = df[df["Pruebas"].isna()].copy()
    elif "Pruebas " in df.columns:
        df = df[df["Pruebas "].isna()].copy()

```

```

# Parse fechas
if "fecha se s/p" not in df.columns:
    raise ValueError("indices_xl_raw: no encuentro columna 'fecha se s/p' tras reconstruir headers.")
df["fecha se s/p"] = to_datetime_safe(df["fecha se s/p"])
df["Fecha FIN cos"] = to_datetime_safe(df.get("Fecha FIN cos"))
df["Fecha Inc cos"] = to_datetime_safe(df.get("Fecha Inc cos"))

df = df[df["fecha se s/p"].notna()].copy()
df = df[df["fecha se s/p"] >= fecha_min_hist].copy()

needed = ["AREA_PRODUCT", "fecha se s/p", "Fecha Inc cos", "Fecha FIN cos", "Bloque", "p/s", "TALLOS TOTAL
ES EN VERDE"]
missing = [c for c in needed if c not in df.columns]
if missing:
    raise ValueError(f"indices_xl_raw: faltan columnas requeridas: {missing}")

df = df[needed].copy()
df = df[df["Fecha FIN cos"].notna()].copy()

df = df.rename(columns={
    "Fecha Inc cos": "Fecha_Inicio_Cosecha (Primer Tallo)",
    "Fecha FIN cos": "Fecha_Fin_Cosecha",
    "Bloque": "Bloques",
    "fecha se s/p": "Fecha S/P",
    "p/s": "S/P",
    "TALLOS TOTALES EN VERDE": "Tallos_Pro",
    "AREA_PRODUCT": "Area"
})

df["Estado"] = "CERRADO"
df["Variedad"] = "XL"
df["Area"] = reemplazos_area(df["Area"])
df["Bloques"] = df["Bloques"].astype(str)

df = normalize_date_col(df, "Fecha S/P")
df = normalize_date_col(df, "Fecha_Inicio_Cosecha (Primer Tallo)")
df = normalize_date_col(df, "Fecha_Fin_Cosecha")
df["Tallos_Pro"] = to_numeric_safe(df["Tallos_Pro"])

return df[[
    "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
    "Tallos_Pro", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
]]

def fenograma_historia_clo(raw_indices_clo: pd.DataFrame, fecha_min_hist: pd.Timestamp) -> pd.DataFrame:
    df = _rebuild_header_like_clo(raw_indices_clo)
    if df.empty:
        return pd.DataFrame(columns=[
            "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
            "Tallos_Pro", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
        ])

    # Parse fechas
    if "Fecha de S/P" not in df.columns:
        raise ValueError("indices_clo_raw: no encuentro columna 'Fecha de S/P' tras reconstruir headers.")
    df["Fecha de S/P"] = to_datetime_safe(df["Fecha de S/P"])
    df["Fecha FIN cos"] = to_datetime_safe(df.get("Fecha FIN cos"))
    df["Fecha Inc cos"] = to_datetime_safe(df.get("Fecha Inc cos"))

    df = df[df["Fecha de S/P"].notna()].copy()
    df = df[df["Fecha de S/P"] >= fecha_min_hist].copy()

    needed = ["AREA_PRODUCT", "Fecha de S/P", "Fecha Inc cos", "Fecha FIN cos", "BLOQUE", "p/s", "TALLOS TOTAL
ES EN VERDE"]
    missing = [c for c in needed if c not in df.columns]
    if missing:
        raise ValueError(f"indices_clo_raw: faltan columnas requeridas: {missing}")

    df = df[needed].copy()
    df = df[df["Fecha FIN cos"].notna()].copy()

```

```

df = df.rename(columns={
    "Fecha Inc cos": "Fecha_Inicio_Cosecha (Primer Tallo)",
    "Fecha FIN cos": "Fecha_Fin_Cosecha",
    "BLOQUE": "Bloques",
    "Fecha de S/P": "Fecha S/P",
    "p/s": "S/P",
    "TALLOS TOTALES EN VERDE": "Tallos_Proj",
    "AREA_PRODUCT": "Area"
})

df["Estado"] = "CERRADO"
df["Variedad"] = "CLO"
df["Area"] = reemplazos_area(df["Area"])
df["Bloques"] = df["Bloques"].astype(str)

df = normalize_date_col(df, "Fecha S/P")
df = normalize_date_col(df, "Fecha_Inicio_Cosecha (Primer Tallo)")
df = normalize_date_col(df, "Fecha_Fin_Cosecha")
df["Tallos_Proj"] = to_numeric_safe(df["Tallos_Proj"])

return df[[
    "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
    "Tallos_Proj", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
]]

# -----
# 321A/321B preferencia (tu lógica)
# -----
def preferir_con_letra_con_bloque_base(df: pd.DataFrame) -> pd.DataFrame:
    out = df.copy()
    out["Bloque_Base"] = out["Bloques"].astype(str).str.replace(r"\D", "", regex=True)

    grp_cols = ["Bloque_Base", "S/P", "Fecha S/P", "Variedad", "Area"]

    def filtrar_grupo(g: pd.DataFrame) -> pd.DataFrame:
        hay_con_letra = (g["Bloques"].astype(str) != g["Bloque_Base"].astype(str)).any()
        if hay_con_letra:
            return g[g["Bloques"].astype(str) != g["Bloque_Base"].astype(str)]
        return g

    out = out.groupby(grp_cols, dropna=False, group_keys=False).apply(filtrar_grupo)
    return out

# -----
# Main (BRONZE -> SILVER only)
# -----
def main() -> None:
    cfg = load_settings()
    hoy = pd.Timestamp(datetime.now().date())

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    fecha_min_hist = pd.to_datetime(cfg.get("sources", {}).get("fecha_min_hist", "2024-01-01"))
    horizon_days = int(cfg["pipeline"]["grid_horizon_days"])

    # 1) Leer fenograma activo desde BRONZE
    df_xlsm = pd.read_parquet(bronze_dir / "fenograma_xlsm_raw.parquet")
    df_xlsm = normalizar_columnas(df_xlsm)

    # 2) Lista de bloques (para filtrar balanza raw; solo optimización)
    bloques_candidatos = (
        df_xlsm.get("Bloques", pd.Series([], dtype=str))
        .dropna()
        .astype(str)
        .str.strip()
        .replace("", np.nan)
        .dropna()
        .unique()
    )

```

```

        .tolist()
    )

# 3) Leer balanza desde BRONZE y filtrar por bloques + fecha mínima
bal = pd.read_parquet(bronze_dir / "balanza_bloque_fecha_raw.parquet")
bal = normalizar_columnas(bal)
if "Bloque" not in bal.columns or "Fecha" not in bal.columns:
    raise ValueError("balanza_bloque_fecha_raw.parquet debe tener columnas ['Bloque','Fecha'].")

bal["Bloque"] = bal["Bloque"].astype(str)
bal["Fecha"] = to_datetime_safe(bal["Fecha"])
bal = bal[bal["Fecha"].notna()].copy()

if bloques_candidatos:
    bal = bal[bal["Bloque"].isin([str(b) for b in bloques_candidatos])].copy()
bal = bal[bal["Fecha"] >= fecha_min_hist].copy()

# 4) Construir activo + históricos (históricos desde índices raw en BRONZE)
activo = fenograma_activo(df_xlsm, bal, hoy)

frames = [activo]

# índices XL
idx_xl_path = bronze_dir / "indices_xl_raw.parquet"
if idx_xl_path.exists():
    raw_xl = pd.read_parquet(idx_xl_path)
    frames.append(fenograma_historia_xl(raw_xl, fecha_min_hist))

# índices CLO
idx_clo_path = bronze_dir / "indices_clo_raw.parquet"
if idx_clo_path.exists():
    raw_clo = pd.read_parquet(idx_clo_path)
    frames.append(fenograma_historia_clo(raw_clo, fecha_min_hist))

total = pd.concat(frames, ignore_index=True)

total = normalize_date_col(total, "Fecha S/P")
total = normalize_date_col(total, "Fecha_Inicio_Cosecha (Primer Tallo)")
total = normalize_date_col(total, "Fecha_Fin_Cosecha")
total = preferir_con_letra_con_bloque_base(total)

# 5) Normalizar a fact_ciclo_maestro (silver)
fact = total.copy()
fact["Bloques"] = fact["Bloques"].astype(str)
fact["Variedad"] = fact["Variedad"].astype(str)
fact["S/P"] = fact["S/P"].astype(str).str.strip().str.upper()
fact["Area"] = fact["Area"].astype(str)

fact = fact.rename(columns={
    "Bloques": "bloque",
    "Variedad": "variedad",
    "S/P": "tipo_sp",
    "Fecha S/P": "fecha_sp",
    "Area": "area",
    "Tallos_Proc": "tallos_proy",
    "Fecha_Inicio_Cosecha (Primer Tallo)": "fecha_inicio_cosecha",
    "Fecha_Fin_Cosecha": "fecha_fin_cosecha",
    "Estado": "estado",
    "Bloque_Base": "bloque_base",
})

# Validaciones mínimas de contrato
if fact["fecha_sp"].isna().any():
    raise ValueError("fact_ciclo_maestro: fecha_sp tiene nulos (revisar fuentes).")

fact["bloque_id"] = fact["bloque"].map(make_bloque_id)
fact["variedad_id"] = fact["variedad"].map(make_variedad_id)

fact["ciclo_id"] = [
    make_ciclo_id(b, v, t, f)
    for b, v, t, f in fact[["bloque_id", "variedad_id", "tipo_sp", "fecha_sp"]].itertuples(index=False)
]

```

```

# Deduplicación: prioriza ACTIVO sobre CERRADO
fact["__prio"] = np.where(fact["estado"].astype(str).str.upper().eq("ACTIVO"), 1, 0)
fact = (
    fact.sort_values(["ciclo_id", "__prio"], ascending=[True, False])
    .drop_duplicates(subset=["ciclo_id"], keep="first")
    .drop(columns=["__prio"])
    .reset_index(drop=True)
)

# Validación llave
if fact.duplicated(["ciclo_id"]).any():
    raise ValueError("fact_ciclo_maestro: ciclo_id no es único luego de dedupe (revisar reglas).")

write_parquet(fact, silver_dir / "fact_ciclo_maestro.parquet")

# 6) Grid (horizonte configurable)
grid = build_grid_ciclo_fecha(fact, horizon_days=horizon_days)
write_parquet(grid, silver_dir / "grid_ciclo_fecha.parquet")

print(f"OK: fact_ciclo_maestro={len(fact)} filas; grid={len(grid)} filas; horizonte={horizon_days} días")

if __name__ == "__main__":
    main()

-----
[136/164] FILE: \src\silver\build_ciclo_maestro_from_sources.py
-----
# src/silver/build_ciclo_maestro_from_sources.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import re
import pandas as pd
import numpy as np
import yaml

from common.io import write_parquet
from common.ids import make_bloque_id, make_variedad_id, make_ciclo_id
from common.timegrid import build_grid_ciclo_fecha

# -----
# Helpers (tuyos, ajustados)
# -----
def normalizar_columnas(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df.columns = [str(c).strip() for c in df.columns]
    return df

def to_datetime_safe(x):
    return pd.to_datetime(x, errors="coerce")

def to_numeric_safe(x):
    return pd.to_numeric(x, errors="coerce")

def normalize_date_col(df: pd.DataFrame, col: str):
    df = df.copy()
    df[col] = to_datetime_safe(df[col]).dt.normalize()
    return df

def reemplazos_area(series: pd.Series) -> pd.Series:
    return series.astype(str).replace({"SSJ": "A-4", "MM1": "MH1", "MM2": "MH2"})

```

```

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

# -----
# BRONZE RAW readers (indices: col_0..col_n)
# -----
def _strip_cell(x) -> str:
    if x is None or (isinstance(x, float) and np.isnan(x)):
        return ""
    return str(x).strip()

def _ensure_cols_are_strings(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df.columns = [str(c) for c in df.columns]
    return df

def _rebuild_header_like_xl(raw: pd.DataFrame) -> pd.DataFrame:
    """
    Replica tu lógica XL original (Excel):
    raw = raw[~raw[0].isna() & (raw[0].astype(str).str.strip() != "")].copy()
    raw = raw.iloc[1:].copy()
    raw.columns = raw.iloc[0]
    df = raw.iloc[1:].copy()

    En Bronze: raw viene como col_0..col_n (todo string).
    """
    raw = _ensure_cols_are_strings(raw)
    col0 = "col_0" if "col_0" in raw.columns else raw.columns[0]
    rr = raw.copy()

    mask = rr[col0].notna() & (rr[col0].astype(str).str.strip() != "")
    rr = rr[mask].copy()

    if len(rr) < 2:
        return pd.DataFrame()

    rr = rr.iloc[1:].copy()

    header = rr.iloc[0].tolist()
    header = [re.sub(r"\s+", " ", _strip_cell(h).replace("\n", " ").replace("\r", " ")).strip() for h in header]

    rr = rr.iloc[1:].copy()
    rr.columns = header
    rr = normalizar_columnas(rr)
    return rr

def _rebuild_header_like_clo(raw: pd.DataFrame) -> pd.DataFrame:
    """
    Replica tu lógica CLO original (Excel):
    raw = raw[~raw[0].isna() & (raw[0].astype(str).str.strip() != "")].copy()
    raw.columns = raw.iloc[0]
    df = raw.iloc[1:].copy()

    En Bronze: raw viene como col_0..col_n (todo string).
    """
    raw = _ensure_cols_are_strings(raw)
    col0 = "col_0" if "col_0" in raw.columns else raw.columns[0]
    rr = raw.copy()

    mask = rr[col0].notna() & (rr[col0].astype(str).str.strip() != "")
    rr = rr[mask].copy()

    if len(rr) < 2:
        return pd.DataFrame()

    header = rr.iloc[0].tolist()

```



```

header = [re.sub(r"\s+", " ", _strip_cell(h).replace("\n", " ").replace("\r", " ")).strip() for h in header]

rr = rr.iloc[1:].copy()
rr.columns = header
rr = normalizar_columnas(rr)
return rr

# -----
# Fenograma activo (tu lógica)
# -----
def fenograma_activo(df_fenograma_xlsm: pd.DataFrame, bal: pd.DataFrame, hoy: pd.Timestamp) -> pd.DataFrame:
    df = normalizar_columnas(df_fenograma_xlsm)

    if "Area " in df.columns and "Area" not in df.columns:
        df = df.rename(columns={"Area ": "Area"})

    df["Bloques"] = df["Bloques"].astype(str)

    valores_excluir = {
        None, 0, 500, 625, 750, 875, 1000, 6025,
        "%1000 GR", "%350 GR", "%500 GR", "%750 GR",
        "00 a 20", "21 a 40", "41 a 60", "61 a 80", "81 a 100",
        "ANDES", "Cajas por cosechar entre:", "CAMPO", "CLOUD", "CV",
        "Distribucion de cosecha en el campo", "DISTRIBUCION POR GRADOS",
        "DOMINGO", "FLOR GRANDE = FG", "FLOR PEQUEÑA = FP", "GENERAL",
        "JUEVES", "LUNES", "MARTES", "MH1", "MH2", "MIERCOLES",
        "MILLION CLOUD", "MM1", "MM2", "OTRO",
        "PROYECCION CAJAS POR GRADOS",
        "SABADO", "SI", "SJP", "SSJ", "Tallos/caja",
        "TOTAL", "Total", "Total cajas", "TOTAL GYPSOS",
        "Total tallos", "VENTAS", "VIERNES", "x",
        "XLENCE", "XLENCE FIVE STARS"
    }

    if "Fiesta" not in df.columns:
        raise ValueError("fenograma_xlsm_raw: no existe columna 'Fiesta' requerida por la lógica de filtrado.")

    df = df[~df["Fiesta"].isin(valores_excluir)].copy()
    df = df[df["Bloques"].notna()].copy()

    cols = ["Bloques", "Fecha S/P", "Area", "S/P", "Variedad", "Tallos / Bloque"]
    missing = [c for c in cols if c not in df.columns]
    if missing:
        raise ValueError(f"fenograma_xlsm_raw: faltan columnas requeridas: {missing}")

    df = df[cols].copy()

    df["Fecha S/P"] = to_datetime_safe(df["Fecha S/P"])
    df["Tallos / Bloque"] = to_numeric_safe(df["Tallos / Bloque"]).fillna(0)

    agg = (
        df.groupby(["Bloques", "S/P", "Variedad", "Area"], dropna=False)
        .agg(**{
            "Fecha S/P": ("Fecha S/P", "max"),
            "Tallos_Proj": ("Tallos / Bloque", "sum")
        })
        .reset_index()
    )

    agg = agg[agg["Area"].notna()].copy()
    agg = agg[(agg["Area"] != 0) & (agg["Area"] != "0")].copy()
    agg["Area"] = reemplazos_area(agg["Area"])

    # Join con balanza para calcular días vegetativos
    tmp = agg.merge(bal, left_on="Bloques", right_on="Bloque", how="left").drop(columns=["Bloque"])
    tmp["Personalizado"] = (tmp["Fecha"] - tmp["Fecha S/P"]).dt.days

    tmp_f = tmp[(tmp["Personalizado"] > 30) | (tmp["Personalizado"].isna())].copy()

```

```

g = (
    tmp_f.groupby(["Bloques", "S/P", "Fecha S/P"], dropna=False)
        .agg(
            Dias_Vegetativo=("Personalizado", "min"),
            Dias_Vegetativo_1=("Personalizado", "max")
        )
        .reset_index()
)

g["Fecha_Inicio_Cosecha (Primer Tallo)"] = g["Fecha S/P"] + pd.to_timedelta(g["Dias_Vegetativo"], unit="D")

g.loc[g["Dias_Vegetativo"].isna(), "Fecha_Inicio_Cosecha (Primer Tallo)"] = pd.NaT

fin_calc = g["Fecha S/P"] + pd.to_timedelta(g["Dias_Vegetativo_1"], unit="D")
g["Fecha_Fin_Cosecha"] = fin_calc

cond1 = g["Dias_Vegetativo"].isna()
cond2 = g["Dias_Vegetativo_1"] <= g["Dias_Vegetativo"]
cond3 = fin_calc >= (hoy - pd.Timedelta(days=4))
g.loc[cond1 | cond2 | cond3, "Fecha_Fin_Cosecha"] = pd.NaT

base = agg.merge(
    g[["Bloques", "Fecha S/P", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha"]],
    on=["Bloques", "Fecha S/P"],
    how="left"
)

base = base[base["Fecha S/P"] < hoy].copy()
base["Estado"] = "ACTIVO"

base = normalize_date_col(base, "Fecha S/P")
base = normalize_date_col(base, "Fecha_Inicio_Cosecha (Primer Tallo)")
base = normalize_date_col(base, "Fecha_Fin_Cosecha")

return base[[
    "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
    "Tallos_Proj", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
]]

# -----
# Históricos XL/CLO (desde BRONZE raw indices*.parquet)
# -----
def fenograma_historia_xl(raw_indices_xl: pd.DataFrame, fecha_min_hist: pd.Timestamp) -> pd.DataFrame:
    df = _rebuild_header_like_xl(raw_indices_xl)
    if df.empty:
        return pd.DataFrame(columns=[
            "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
            "Tallos_Proj", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
        ])

    if "Pruebas" in df.columns:
        df = df[df["Pruebas"].isna()].copy()
    elif "Pruebas " in df.columns:
        df = df[df["Pruebas "].isna()].copy()

    df["fecha se s/p"] = to_datetime_safe(df.get("fecha se s/p"))
    df["Fecha FIN cos"] = to_datetime_safe(df.get("Fecha FIN cos"))
    df["Fecha Inc cos"] = to_datetime_safe(df.get("Fecha Inc cos"))

    df = df[df["fecha se s/p"].notna()].copy()
    df = df[df["fecha se s/p"] >= fecha_min_hist].copy()

    needed = ["AREA_PRODUCT", "fecha se s/p", "Fecha Inc cos", "Fecha FIN cos", "Bloque", "p/s", "TALLOS TOTAL
ES EN VERDE"]
    missing = [c for c in needed if c not in df.columns]
    if missing:
        raise ValueError(f"indices_xl_raw: faltan columnas requeridas: {missing}")

    df = df[needed].copy()
    df = df[df["Fecha FIN cos"].notna()].copy()

```

```

df = df.rename(columns={
    "Fecha Inc cos": "Fecha_Inicio_Cosecha (Primer Tallo)",
    "Fecha FIN cos": "Fecha_Fin_Cosecha",
    "Bloque": "Bloques",
    "fecha se s/p": "Fecha S/P",
    "p/s": "S/P",
    "TALLOS TOTALES EN VERDE": "Tallos_Proj",
    "AREA_PRODUCT": "Area"
})

df["Estado"] = "CERRADO"
df["Variedad"] = "XL"
df["Area"] = reemplazos_area(df["Area"])
df["Bloques"] = df["Bloques"].astype(str)

df = normalize_date_col(df, "Fecha S/P")
df = normalize_date_col(df, "Fecha_Inicio_Cosecha (Primer Tallo)")
df = normalize_date_col(df, "Fecha_Fin_Cosecha")
df["Tallos_Proj"] = to_numeric_safe(df["Tallos_Proj"])

return df[[
    "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
    "Tallos_Proj", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
]]

def fenograma_historia_clo(raw_indices_clo: pd.DataFrame, fecha_min_hist: pd.Timestamp) -> pd.DataFrame:
    df = _rebuild_header_like_clo(raw_indices_clo)
    if df.empty:
        return pd.DataFrame(columns=[
            "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",
            "Tallos_Proj", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
        ])

    df["Fecha de S/P"] = to_datetime_safe(df.get("Fecha de S/P"))
    df["Fecha FIN cos"] = to_datetime_safe(df.get("Fecha FIN cos"))
    df["Fecha Inc cos"] = to_datetime_safe(df.get("Fecha Inc cos"))

    df = df[df["Fecha de S/P"].notna()].copy()
    df = df[df["Fecha de S/P"] >= fecha_min_hist].copy()

    needed = ["AREA_PRODUCT", "Fecha de S/P", "Fecha Inc cos", "Fecha FIN cos", "BLOQUE", "p/s", "TALLOS TOTAL
ES EN VERDE"]
    missing = [c for c in needed if c not in df.columns]
    if missing:
        raise ValueError(f"indices_clo_raw: faltan columnas requeridas: {missing}")

    df = df[needed].copy()
    df = df[df["Fecha FIN cos"].notna()].copy()

    df = df.rename(columns={
        "Fecha Inc cos": "Fecha_Inicio_Cosecha (Primer Tallo)",
        "Fecha FIN cos": "Fecha_Fin_Cosecha",
        "BLOQUE": "Bloques",
        "Fecha de S/P": "Fecha S/P",
        "p/s": "S/P",
        "TALLOS TOTALES EN VERDE": "Tallos_Proj",
        "AREA_PRODUCT": "Area"
    })

    df["Estado"] = "CERRADO"
    df["Variedad"] = "CLO"
    df["Area"] = reemplazos_area(df["Area"])
    df["Bloques"] = df["Bloques"].astype(str)

    df = normalize_date_col(df, "Fecha S/P")
    df = normalize_date_col(df, "Fecha_Inicio_Cosecha (Primer Tallo)")
    df = normalize_date_col(df, "Fecha_Fin_Cosecha")
    df["Tallos_Proj"] = to_numeric_safe(df["Tallos_Proj"])

    return df[[
        "Bloques", "Fecha S/P", "S/P", "Variedad", "Area",

```

```

        "Tallos_Pro", "Fecha_Inicio_Cosecha (Primer Tallo)", "Fecha_Fin_Cosecha", "Estado"
    ]]

# -----
# 321A/321B preferencia (tu lógica)
# -----
def preferir_con_letra_con_bloque_base(df: pd.DataFrame) -> pd.DataFrame:
    out = df.copy()

    # Base numérica (321A -> 321)
    out["Bloque_Base"] = out["Bloques"].astype(str).str.replace(r"\D", "", regex=True)

    grp_cols = ["Bloque_Base", "S/P", "Fecha S/P", "Variedad", "Area"]

    # con letra = distinto al base (321A != 321)
    out["__con_letra"] = out["Bloques"].astype(str) != out["Bloque_Base"].astype(str)

    # si en el grupo hay con letra, quedarse solo con esos; si no hay, dejar todos
    has_letra = out.groupby(grp_cols, dropna=False)["__con_letra"].transform("any")
    out = out[(~has_letra) | (out["__con_letra"])]

    return out.drop(columns=["__con_letra"])

# -----
# Construcción total + salida silver + grid (BRONZE -> SILVER)
# -----
def main() -> None:
    cfg = load_settings()
    hoy = pd.Timestamp(datetime.now().date())

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    fecha_min_hist = pd.to_datetime(cfg.get("sources", {}).get("fecha_min_hist", "2024-01-01"))
    horizon_days = int(cfg["pipeline"]["grid_horizon_days"])

    # 1) Fenograma activo desde BRONZE
    df_xlsm = pd.read_parquet(bronze_dir / "fenograma_xlsm_raw.parquet")
    df_xlsm = normalizar_columnas(df_xlsm)

    # 2) Bloques candidatos (solo para filtrar balanza raw)
    bloques_candidatos = (
        df_xlsm.get("Bloques", pd.Series([], dtype=str))
        .dropna()
        .astype(str)
        .str.strip()
        .replace("", np.nan)
        .dropna()
        .unique()
        .tolist()
    )

    # 3) Balanza desde BRONZE
    bal = pd.read_parquet(bronze_dir / "balanza_bloque_fecha_raw.parquet")
    bal = normalizar_columnas(bal)
    if "Bloque" not in bal.columns or "Fecha" not in bal.columns:
        raise ValueError("balanza_bloque_fecha_raw.parquet debe tener columnas ['Bloque', 'Fecha'].")

    bal["Bloque"] = bal["Bloque"].astype(str)
    bal["Fecha"] = to_datetime_safe(bal["Fecha"])
    bal = bal[bal["Fecha"].notna()].copy()

    if bloques_candidatos:
        bal = bal[bal["Bloque"].isin([str(b) for b in bloques_candidatos])].copy()
    bal = bal[bal["Fecha"] >= fecha_min_hist].copy()

    # 4) Activo + históricos desde BRONZE
    activo = fenograma_activo(df_xlsm, bal, hoy)

```

```

frames = [activo]

idx_xl = bronze_dir / "indices_xl_raw.parquet"
if idx_xl.exists():
    frames.append(fenograma_historia_xl(pd.read_parquet(idx_xl), fecha_min_hist))

idx_clo = bronze_dir / "indices_clo_raw.parquet"
if idx_clo.exists():
    frames.append(fenograma_historia_clo(pd.read_parquet(idx_clo), fecha_min_hist))

total = pd.concat(frames, ignore_index=True)

total = normalize_date_col(total, "Fecha S/P")
total = normalize_date_col(total, "Fecha_Inicio_Cosecha (Primer Tallo)")
total = normalize_date_col(total, "Fecha_Fin_Cosecha")
total = preferir_con_letra_con_bloque_base(total)

# 5) Normalizar a fact_ciclo_maestro (silver)
fact = total.copy()
fact["Bloques"] = fact["Bloques"].astype(str)
fact["Variedad"] = fact["Variedad"].astype(str)
fact["S/P"] = fact["S/P"].astype(str).str.strip().str.upper()
fact["Area"] = fact["Area"].astype(str)

fact = fact.rename(columns={
    "Bloques": "bloque",
    "Variedad": "variedad",
    "S/P": "tipo_sp",
    "Fecha S/P": "fecha_sp",
    "Area": "area",
    "Tallos_Proj": "tallos_proy",
    "Fecha_Inicio_Cosecha (Primer Tallo)": "fecha_inicio_cosecha",
    "Fecha_Fin_Cosecha": "fecha_fin_cosecha",
    "Estado": "estado",
    "Bloque_Base": "bloque_base",
})

# Jerarquía de bloque (tu lógica)
fact["bloque"] = fact["bloque"].astype(str).str.strip()
fact["bloque_padre"] = fact["bloque_base"].astype(str).str.strip()

# IDs
if fact["fecha_sp"].isna().any():
    raise ValueError("fact_ciclo_maestro: fecha_sp tiene nulos (revisar fuentes).")

fact["bloque_id"] = fact["bloque"].map(make_bloque_id)
fact["bloque_padre_id"] = fact["bloque_padre"].map(make_bloque_id)

fact["variedad_id"] = fact["variedad"].map(make_variedad_id)
fact["ciclo_id"] = [
    make_ciclo_id(b, v, t, f)
    for b, v, t, f in fact[["bloque_id", "variedad_id", "tipo_sp", "fecha_sp"]].itertuples(index=False)
]

# Deduplicación: prioriza ACTIVO sobre CERRADO
fact["__prio"] = np.where(fact["estado"].astype(str).str.upper().eq("ACTIVO"), 1, 0)
fact = (
    fact.sort_values(["ciclo_id", "__prio"], ascending=[True, False])
    .drop_duplicates(subset=["ciclo_id"], keep="first")
    .drop(columns=["__prio"])
    .reset_index(drop=True)
)

if fact.duplicated(["ciclo_id"]).any():
    raise ValueError("fact_ciclo_maestro: ciclo_id no es único luego de dedupe (revisar reglas).")

write_parquet(fact, silver_dir / "fact_ciclo_maestro.parquet")

# 6) Grid
grid = build_grid_ciclo_fecha(fact, horizon_days=horizon_days)
write_parquet(grid, silver_dir / "grid_ciclo_fecha.parquet")

```

```

print(f"OK: fact_ciclo_maestro={len(fact)} filas; grid={len(grid)} filas; horizonte={horizon_days} días")

if __name__ == "__main__":
    main()

-----
[137/164] FILE: \src\silver\build_dim_baseline_capacidad_tallos_h_persona.py
-----
# src/silver/build_dim_baseline_capacidad_tallos_h_persona.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def map_proceso(cod: pd.Series) -> pd.Series:
    c = cod.astype(str).str.upper().str.strip()
    # Solo los que hoy aplican (según lo que definiste)
    return np.select(
        [c.eq("CXLTA1"), c.eq("CXLTAH")],
        ["TINTURADO", "ARCOIRIS"],
        default="OTROS",
    )

def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    # Fechas mínimas (recorte para baseline)
    fecha_min = pd.to_datetime(cfg.get("pipeline", {}).get("capacidad_fecha_min", "2024-01-01"))

    # -----
    # BRONZE inputs
    # -----
    ghu_path = bronze_dir / "ghu_maestro_horas.parquet"
    per_path = bronze_dir / "personal_raw.parquet"

    if not ghu_path.exists():
        raise FileNotFoundError(f"No existe Bronze: {ghu_path}. Ejecuta src/bronze/build_ghu_maestro_horas.py")
    )

    if not per_path.exists():
        raise FileNotFoundError(f"No existe Bronze: {per_path}. Ejecuta src/bronze/build_personal_sources.py")

    df = pd.read_parquet(ghu_path)
    per = pd.read_parquet(per_path)

    df.columns = [str(c).strip() for c in df.columns]
    per.columns = [str(c).strip() for c in per.columns]

    # Validar contrato mínimo
    need_df = [

```

```

    "fecha",
    "codigo_personal",
    "codigo_actividad",
    "unidad_medida",
    "horas_presenciales",
    "unidades_producidas",
]
missing_df = [c for c in need_df if c not in df.columns]
if missing_df:
    raise ValueError(f"ghu_maestro_horas: faltan columnas requeridas: {missing_df}")

need_per = ["codigo_personal", "Activo_o_Inactivo"]
missing_per = [c for c in need_per if c not in per.columns]
if missing_per:
    raise ValueError(f"personal_raw: faltan columnas requeridas: {missing_per}")

# -----
# Transformaciones Silver
# -----
df["fecha"] = _norm_date(df["fecha"])
df = df[df["fecha"].notna()].copy()
df = df[df["fecha"] >= fecha_min].copy()

df["codigo_personal"] = pd.to_numeric(df["codigo_personal"], errors="coerce").astype("Int64")
df["codigo_actividad"] = df["codigo_actividad"].astype(str).str.upper().str.strip()
df["unidad_medida"] = df["unidad_medida"].astype(str).str.upper().str.strip()

df["horas_presenciales"] = pd.to_numeric(df["horas_presenciales"], errors="coerce").fillna(0.0)
df["unidades_producidas"] = pd.to_numeric(df["unidades_producidas"], errors="coerce").fillna(0.0)

# horas_acumula es opcional (según Bronze); si no está, se crea 0.0
if "horas_acumula" in df.columns:
    df["horas_acumula"] = pd.to_numeric(df["horas_acumula"], errors="coerce").fillna(0.0)
else:
    df["horas_acumula"] = 0.0

# Filtro negocio: solo actividades relevantes para tallos/h persona
df = df[df["codigo_actividad"].isin(["CXLTAI", "CXLTA RH"])]

# Solo tallos (si vienen kilos u otros, se ignoran para este baseline)
df = df[df["unidad_medida"].str.contains("TALLO", na=False)]

df["proceso"] = map_proceso(df["codigo_actividad"])
df = df[df["proceso"].isin(["TINTURADO", "ARCOIRIS"])]

# Personal join (Silver consume Bronze personal_raw)
per["codigo_personal"] = pd.to_numeric(per["codigo_personal"], errors="coerce").astype("Int64")
per["Activo_o_Inactivo"] = per["Activo_o_Inactivo"].astype(str).str.strip()

df = df.merge(per[["codigo_personal", "Activo_o_Inactivo"]], on="codigo_personal", how="left")
df = df.rename(columns={"Activo_o_Inactivo": "activo_inactivo"})

# Consolidación por día/persona/proceso
daily = (
    df.groupby(["fecha", "proceso", "codigo_personal"], dropna=False)
    .agg(
        horas_presenciales_dia=("horas_presenciales", "sum"),
        tallos_procesados_dia=("unidades_producidas", "sum"),
        horas_acumula_max=("horas_acumula", "max"),
        activo_inactivo=("activo_inactivo", "last"),
    )
    .reset_index()
)

daily = daily[(daily["horas_presenciales_dia"] > 0) & (daily["tallos_procesados_dia"] > 0)].copy()
daily["tallos_h_persona_dia"] = daily["tallos_procesados_dia"] / daily["horas_presenciales_dia"]

# Sanity: evita valores basura extremos
daily = daily[(daily["tallos_h_persona_dia"] > 10) & (daily["tallos_h_persona_dia"] < 2000)].copy()

# Baseline por proceso (mediana y percentiles)
base = (

```

```

        daily.groupby("proceso", dropna=False)
            .agg(
                tallos_h_persona_mediana=("tallos_h_persona_dia", "median"),
                tallos_h_persona_p25=("tallos_h_persona_dia", lambda s: float(s.quantile(0.25))),
                tallos_h_persona_p75=("tallos_h_persona_dia", lambda s: float(s.quantile(0.75))),
                n_registros=("tallos_h_persona_dia", "count"),
            )
            .reset_index()
    )
    base["created_at"] = datetime.now().isoformat(timespec="seconds")

    out_path = silver_dir / "dim_baseline_capacidad_tallos_h_persona.parquet"
    write_parquet(base, out_path)

    print(f"OK: dim_baseline_capacidad_tallos_h_persona={len(base)} filas -> {out_path}")
    print(base.to_string(index=False))

if __name__ == "__main__":
    main()

-----
[138/164] FILE: \src\silver\build_dim_capacidad_baseline_tallos_proceso.py
-----
# src/silver/build_dim_capacidad_baseline_tallos_proceso.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _to_float(s: pd.Series) -> pd.Series:
    # soporta strings con coma decimal
    return pd.to_numeric(s.astype(str).str.replace(",", ".", regex=False), errors="coerce")

def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    ghu_path = bronze_dir / "ghu_maestro_horas.parquet"
    per_path = bronze_dir / "personal_raw.parquet"

    if not ghu_path.exists():
        raise FileNotFoundError(f"No existe Bronze: {ghu_path}. Ejecuta src/bronze/build_ghu_maestro_horas.py")
    )
    if not per_path.exists():
        raise FileNotFoundError(f"No existe Bronze: {per_path}. Ejecuta src/bronze/build_personal_sources.py")

    df = pd.read_parquet(ghu_path)
    per = pd.read_parquet(per_path)

    df.columns = [str(c).strip() for c in df.columns]
    per.columns = [str(c).strip() for c in per.columns]

```



```

# Validar contrato mínimo (lo que tu SQL traía)
need_df = [
    "fecha",
    "codigo_personal",
    "codigo_actividad",
    "horas_presenciales",
    "unidades_producidas",
    "unidad_medida",
    "horas_acumula",
]
missing_df = [c for c in need_df if c not in df.columns]
if missing_df:
    raise ValueError(f"ghu_maestro_horas: faltan columnas requeridas: {missing_df}")

need_per = ["codigo_personal", "Activo_o_Inactivo"]
missing_per = [c for c in need_per if c not in per.columns]
if missing_per:
    raise ValueError(f"personal_raw: faltan columnas requeridas: {missing_per}")

# Normalizaciones (Silver)
df["fecha"] = _norm_date(df["fecha"])
df["codigo_personal"] = df["codigo_personal"].astype(str).str.strip()
df["codigo_actividad"] = df["codigo_actividad"].astype(str).str.strip().str.upper()
df["unidad_medida"] = df["unidad_medida"].astype(str).str.strip().str.upper()

df["horas_presenciales"] = _to_float(df["horas_presenciales"])
df["unidades_producidas"] = _to_float(df["unidades_producidas"])
df["horas_acumula"] = _to_float(df["horas_acumula"])

# Join con Personal (Bronze)
per["codigo_personal"] = pd.to_numeric(per["codigo_personal"], errors="coerce").astype("Int64")
per["Activo_o_Inactivo"] = per["Activo_o_Inactivo"].astype(str).str.strip()

# Ojo: df.codigo_personal está string; convertimos a Int64 para matchear como en los otros scripts
df["codigo_personal_int"] = pd.to_numeric(df["codigo_personal"], errors="coerce").astype("Int64")
df = df.merge(
    per[["codigo_personal", "Activo_o_Inactivo"]].rename(columns={"codigo_personal": "codigo_personal_int"}),
    on="codigo_personal_int",
    how="left",
)
df = df.rename(columns={"Activo_o_Inactivo": "activo_inactivo"})

# Map proceso
df["proceso"] = np.select(
    [df["codigo_actividad"].eq("CXLTA1"), df["codigo_actividad"].eq("CXLTA2H")],
    ["TINTURADO", "ARCOIRIS"],
    default="OTROS",
)

# Filtrar solo TALLOS (unidad nativa)
df = df[df["unidad_medida"].eq("TALLOS")].copy()

# Validaciones
df = df[df["fecha"].notna()].copy()
df = df[df["codigo_personal"].astype(str).str.len() > 0].copy()
df = df[df["horas_presenciales"].notna() & (df["horas_presenciales"] > 0)].copy()
df = df[df["unidades_producidas"].notna() & (df["unidades_producidas"] > 0)].copy()

# Agregar por día x persona x proceso
fact = (
    df.groupby(["fecha", "proceso", "codigo_personal"], dropna=False)
    .agg(
        activo_inactivo=("activo_inactivo", "last"),
        horas_presenciales_dia=("horas_presenciales", "sum"),
        tallos_procesados_dia=("unidades_producidas", "sum"),
        horas_acumula=("horas_acumula", "max"),
    )
    .reset_index()
)

```

```

fact["tallos_h_persona_dia"] = fact["tallos_procesados_dia"] / fact["horas_presenciales_dia"]

# Cap por proceso (1%?99%)
def cap_percentiles(g: pd.DataFrame) -> pd.DataFrame:
    g = g.copy()

    x = g["tallos_h_persona_dia"].astype(float).values
    x = x[~np.isnan(x)]
    if x.size < 5:
        return g

    p1 = float(np.nanpercentile(x, 1))
    p99 = float(np.nanpercentile(x, 99))
    g["tallos_h_persona_dia"] = g["tallos_h_persona_dia"].clip(p1, p99)
    return g

fact = (
    fact.groupby("proceso", dropna=False, group_keys=False)
        .apply(cap_percentiles, include_groups=True)
        .reset_index(drop=True)
)

fact["created_at"] = datetime.now().isoformat(timespec="seconds")

out_fact = silver_dir / "fact_capacidad_tallos_proceso_hist.parquet"
write_parquet(fact, out_fact)

base = (
    fact.groupby("proceso", dropna=False)
        .agg(
            tallos_h_persona_mediana=("tallos_h_persona_dia", "median"),
            tallos_h_persona_p25=("tallos_h_persona_dia", lambda s: float(np.nanpercentile(s, 25))),
            tallos_h_persona_p75=("tallos_h_persona_dia", lambda s: float(np.nanpercentile(s, 75))),
            n_registros=("tallos_h_persona_dia", "size"),
        )
        .reset_index()
)
base["created_at"] = datetime.now().isoformat(timespec="seconds")

out_base = silver_dir / "dim_capacidad_baseline_tallos_proceso.parquet"
write_parquet(base, out_base)

print(f"OK: fact_capacidad_tallos_proceso_hist={len(fact)} -> {out_fact}")
print("Baseline tallos/h/persona:\n", base.to_string(index=False))

if __name__ == "__main__":
    main()

```

```

-----
[139/164] FILE: \src\silver\build_dim_clima_bloque_dia.py
-----

```

```

from __future__ import annotations

from pathlib import Path
import pandas as pd

from common.io import read_parquet, write_parquet

BASE_GDC = 7.1 # umbral estándar empresa para Gypsophila

def _normalize_area(s: pd.Series) -> pd.Series:
    return (
        s.astype(str)
        .str.strip()
        .str.upper()
        .str.replace(r"\s+", " ", regex=True)
    )

```

```

)

def _area_to_station(area_norm: pd.Series) -> pd.Series:
    station = pd.Series(index=area_norm.index, dtype="object")

    is_main = area_norm.isin(["MH1", "MH2", "CV"])
    is_a4 = area_norm.isin(["A-4", "A4"])

    station.loc[is_main] = "MAIN"
    station.loc[is_a4] = "A4"
    return station

def main() -> None:
    created_at = pd.Timestamp.utcnow()

    df_weather = read_parquet(Path("data/silver/weather_hour_wide.parquet"))
    df_maestro = read_parquet(Path("data/silver/fact_ciclo_maestro.parquet"))

    # -----
    # 1) Map bloque_base -> area -> station
    # -----
    need_cols = {"bloque_base", "area"}
    missing = need_cols - set(df_maestro.columns)
    if missing:
        raise ValueError(f"fact_ciclo_maestro.parquet no tiene columnas requeridas: {sorted(missing)}")

    map_blk = df_maestro[["bloque_base", "area"]].copy()
    map_blk["area_norm"] = _normalize_area(map_blk["area"])
    map_blk["station"] = _area_to_station(map_blk["area_norm"])

    bad = map_blk[map_blk["station"].isna()][["bloque_base", "area"]].drop_duplicates()
    if len(bad) > 0:
        raise ValueError(f"No se pudo mapear area -> station. Ejemplos: {bad.head(20).to_dict('records')}")

    map_blk = (
        map_blk.groupby(["bloque_base", "area_norm", "station"], as_index=False)
            .size()
            .sort_values(["bloque_base", "size"], ascending=[True, False])
            .drop_duplicates(subset=["bloque_base"], keep="first")
    )
    map_blk = map_blk.rename(columns={"area_norm": "area"})
    map_blk = map_blk[["bloque_base", "area", "station"]]

    # -----
    # 2) Weather hourly -> daily by station (AGRONÓMICO)
    # -----
    need_wcols = {
        "dt_hora", "station",
        "rainfall_mm", "temp_avg", "solar_energy_j_m2",
        "En_Lluvia",
        "wind_speed_avg", "wind_run",    # <-- viento
    }
    missing_w = need_wcols - set(df_weather.columns)
    if missing_w:
        raise ValueError(f"weather_hour_wide.parquet no tiene columnas requeridas: {sorted(missing_w)}")

    w = df_weather[list(need_wcols)].copy()
    w["dt_hora"] = pd.to_datetime(w["dt_hora"], errors="coerce")
    if w["dt_hora"].isna().any():
        raise ValueError("dt_hora contiene valores inválidos")

    w["fecha"] = w["dt_hora"].dt.normalize()
    w["station"] = w["station"].astype(str).str.strip().str.upper()

    w["En_Lluvia"] = pd.to_numeric(w["En_Lluvia"], errors="coerce").fillna(0)
    w["En_Lluvia"] = (w["En_Lluvia"] > 0).astype(int)

    # Agregación diaria por estación
    daily = (
        w.groupby(["station", "fecha"], as_index=False)

```

```

        .agg(
            rainfall_mm_dia=("rainfall_mm", "sum"),
            temp_avg_dia=("temp_avg", "mean"),
            solar_energy_j_m2_dia=("solar_energy_j_m2", "sum"),
            horas_lluvia=("En_Lluvia", "sum"),
            en_lluvia_dia=("En_Lluvia", "max"),
            wind_speed_avg_dia=("wind_speed_avg", "mean"),
            wind_run_dia=("wind_run", "sum"),
        )
    )

# GDC diario (normalizado)
daily["gdc_base"] = BASE_GDC
daily["gdc_dia"] = (daily["temp_avg_dia"] - BASE_GDC).clip(lower=0)

# -----
# 3) Broadcast clima a bloque_base
# -----
out = map_blk.merge(daily, on="station", how="left")

if out["fecha"].isna().all():
    raise ValueError("Join bloque_base -> station con clima diario no produjo fechas. Revisa station.")

out = out.dropna(subset=["fecha"]).copy()
out["created_at"] = created_at

out = out[
    [
        "fecha",
        "bloque_base",
        "area",
        "station",
        "rainfall_mm_dia",
        "horas_lluvia",
        "en_lluvia_dia",
        "temp_avg_dia",
        "solar_energy_j_m2_dia",
        "wind_speed_avg_dia",
        "wind_run_dia",
        "gdc_base",
        "gdc_dia",
        "created_at",
    ]
].sort_values(["fecha", "bloque_base"]).reset_index(drop=True)

write_parquet(out, Path("data/silver/dim_clima_bloque_dia.parquet"))
print(f"OK -> data/silver/dim_clima_bloque_dia.parquet | rows={len(out):,}")

if __name__ == "__main__":
    main()

```

```

-----
[140/164] FILE: \src\silver\build_dim cosecha progress bloque_fecha.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
import pandas as pd
import numpy as np

```

```

from common.io import read_parquet, write_parquet

```

```

def main() -> None:
    created_at = pd.Timestamp.utcnow()

```

```

# -----
# Inputs (SILVER)
# -----

```

```

df_real = read_parquet(Path("data/silver/fact_cosecha_real_grado_dia.parquet"))

```

```

df_maestro = read_parquet(Path("data/silver/fact_ciclo_maestro.parquet"))
df_clima = read_parquet(Path("data/silver/dim_clima_bloque_dia.parquet"))

# grid_ciclo_fecha lo generas desde ciclo_maestro_from_fenograma
df_grid = read_parquet(Path("data/silver/grid_ciclo_fecha.parquet"))

# milestones: lo usamos después para validaciones / futura etapa (no es obligatorio para calcular progreso
real)
# df_milestones = read_parquet(Path("data/silver/milestones_ciclo_final.parquet"))

# -----
# Validaciones mínimas de columnas
# -----
need_real = {"fecha", "bloque_padre", "variedad", "grado", "tallos_real"}
miss = need_real - set(df_real.columns)
if miss:
    raise ValueError(f"fact_cosecha_real_grado_dia.parquet sin columnas: {sorted(miss)}")

need_maestro = {"bloque", "bloque_base", "ciclo_id", "variedad", "area"}
miss = need_maestro - set(df_maestro.columns)
if miss:
    raise ValueError(f"fact_ciclo_maestro.parquet sin columnas: {sorted(miss)}")

need_clima = {"fecha", "bloque_base", "gdc_dia"}
miss = need_clima - set(df_clima.columns)
if miss:
    raise ValueError(f"dim_clima_bloque_dia.parquet sin columnas: {sorted(miss)}")

# grid: no sé tu schema exacto; validamos lo mínimo para mapear fecha -> ciclo
# Ideal: que tenga (fecha, ciclo_id) y/o (fecha, bloque_base, ciclo_id)
if "fecha" not in df_grid.columns or "ciclo_id" not in df_grid.columns:
    raise ValueError("grid_ciclo_fecha.parquet debe tener al menos columnas: fecha, ciclo_id")

# -----
# 1) Normalizar fecha y definir bloque_base
# -----
r = df_real.copy()
r["fecha"] = pd.to_datetime(r["fecha"]).dt.normalize()

# En este dataset, bloque_padre YA ES el bloque_base
r = r.rename(columns={"bloque_padre": "bloque_base"})

# -----
# 2) Real a día-bloque_base-variedad
# -----
real_dia = (
    r.groupby(["fecha", "bloque_base", "variedad"], as_index=False)
    .agg(tallos_real_dia=("tallos_real", "sum"))
)

# -----
# 3) Asignar ciclo_id (preferir grid si tiene granularidad por bloque_base)
# -----
g = df_grid.copy()
g["fecha"] = pd.to_datetime(g["fecha"]).dt.normalize()

join_cols = set(g.columns)

if "bloque_base" in join_cols:
    # Mejor caso: grid ya está por bloque_base-fecha
    real_dia = real_dia.merge(
        g[["fecha", "bloque_base", "ciclo_id"]].drop_duplicates(),
        on=["fecha", "bloque_base"],
        how="left",
    )
else:
    # Caso mínimo: grid por fecha-ciclo (menos preciso si hay ciclos simultáneos)
    # En ese caso usamos ciclo_id del maestro como fallback para filtrar.
    real_dia = real_dia.merge(
        g[["fecha", "ciclo_id"]].drop_duplicates(),
        on=["fecha"],
        how="left",
    )

```

```

        suffixes=("", "_grid"),
    )
    # Si hay múltiples ciclos por fecha, esto puede duplicar filas.
    # Para no reventar, intentamos quedarnos con el ciclo del maestro.
    # Nota: para esto necesitamos ciclo_id por bloque_base en maestro.
    m_ciclo = df_maestro[["bloque_base", "ciclo_id"]].drop_duplicates()
    real_dia = real_dia.merge(m_ciclo, on="bloque_base", how="left", suffixes=("", "_maestro"))

    # Resolver ciclo_id final:
    # - si ciclo_id_maestro existe, úsalo
    # - si no, usa el del grid (si quedó único)
    # Primero, renombramos para no confundir
    if "ciclo_id_maestro" not in real_dia.columns:
        # si merge no lo creó por colisiones, lo armamos
        pass

    # En este branch, hay potencial de duplicación. Una forma segura:
    # nos quedamos con filas donde ciclo_id (del grid) == ciclo_id_maestro.
    if "ciclo_id_maestro" in real_dia.columns:
        real_dia = real_dia[real_dia["ciclo_id"] == real_dia["ciclo_id_maestro"]].copy()

    # Si aún queda nulo, fallback a ciclo_id_maestro
    real_dia["ciclo_id"] = real_dia["ciclo_id"].fillna(real_dia.get("ciclo_id_maestro"))

    # Limpiar columnas auxiliares
    drop_cols = [c for c in ["ciclo_id_maestro"] if c in real_dia.columns]
    if drop_cols:
        real_dia = real_dia.drop(columns=drop_cols)

if real_dia["ciclo_id"].isna().any():
    bad = real_dia[real_dia["ciclo_id"].isna()][["fecha", "bloque_base"]].drop_duplicates().head(30)
    raise ValueError(
        "No pude asignar ciclo_id a algunas filas (fecha, bloque_base). "
        f"Ejemplos: {bad.to_dict('records')}"
    )

# -----
# 4) Calcular inicio/fin real por ciclo-bloque-variedad
# -----
# Inicio real: primera fecha con tallos_real_dia > 0
real_pos = real_dia[real_dia["tallos_real_dia"] > 0].copy()

anchors = (
    real_pos.groupby(["ciclo_id", "bloque_base", "variedad"], as_index=False)
        .agg(
            fecha_inicio_real=("fecha", "min"),
            fecha_fin_real=("fecha", "max"),
            tallos_total_real=("tallos_real_dia", "sum"),
        )
)

out = real_dia.merge(anchors, on=["ciclo_id", "bloque_base", "variedad"], how="left")

# dia_rel y acumulados (solo donde hay inicio_real)
out = out.sort_values(["ciclo_id", "bloque_base", "variedad", "fecha"]).reset_index(drop=True)

# tallos_acum_real: cumsum sobre días con inicio
out["tallos_acum_real"] = (
    out.groupby(["ciclo_id", "bloque_base", "variedad"])["tallos_real_dia"].cumsum()
)

# pct_avance_real: solo si tallos_total_real > 0
out["pct_avance_real"] = np.where(
    out["tallos_total_real"].fillna(0) > 0,
    out["tallos_acum_real"] / out["tallos_total_real"],
    np.nan,
)

# dia_rel_cosecha_real
out["dia_rel_cosecha_real"] = np.where(
    out["fecha_inicio_real"].notna(),
    (out["fecha"] - out["fecha_inicio_real"]).dt.days,

```

```

        np.nan,
    )

    # en_ventana_cosecha_real (según real)
    out["en_ventana_cosecha_real"] = np.where(
        out["fecha_inicio_real"].notna() & out["fecha_fin_real"].notna(),
        ((out["fecha"] >= out["fecha_inicio_real"]) & (out["fecha"] <= out["fecha_fin_real"])).astype(int),
        0,
    )

    # -----
    # 5) Join clima (gdc_dia) y calcular gdc_acum_real
    # -----
    clima = df_clima[["fecha", "bloque_base", "gdc_dia"]].copy()
    clima["fecha"] = pd.to_datetime(clima["fecha"]).dt.normalize()

    out = out.merge(clima, on=["fecha", "bloque_base"], how="left")

    # gdc_acum_real: acumular gdc desde fecha_inicio_real
    out["gdc_dia_eff"] = out["gdc_dia"].fillna(0.0)

    # Marcador: solo acumular desde inicio_real (antes de inicio, 0)
    out["iniciado"] = np.where(
        out["fecha_inicio_real"].notna() & (out["fecha"] >= out["fecha_inicio_real"]),
        1,
        0,
    )
    out["gdc_dia_iniciado"] = out["gdc_dia_eff"] * out["iniciado"]

    out["gdc_acum_real"] = (
        out.groupby(["ciclo_id", "bloque_base", "variedad"])["gdc_dia_iniciado"].cumsum()
    )

    out = out.drop(columns=["gdc_dia_eff", "iniciado", "gdc_dia_iniciado"])

    # -----
    # 6) Final
    # -----
    out["created_at"] = created_at

    cols = [
        "ciclo_id",
        "fecha",
        "bloque_base",
        "variedad",
        "tallos_real_dia",
        "tallos_acum_real",
        "tallos_total_real",
        "pct_avance_real",
        "fecha_inicio_real",
        "fecha_fin_real",
        "dia_rel_cosecha_real",
        "en_ventana_cosecha_real",
        "gdc_dia",
        "gdc_acum_real",
        "created_at",
    ]

    out = out[cols].sort_values(["ciclo_id", "bloque_base", "variedad", "fecha"]).reset_index(drop=True)

    write_parquet(out, Path("data/silver/dim_cosecha_progress_bloque_fecha.parquet"))
    print(f"OK -> data/silver/dim_cosecha_progress_bloque_fecha.parquet | rows={len(out):,}")

if __name__ == "__main__":
    main()

```

```

-----
[141/164] FILE: \src\silver\build_dim_dh_baseline_grado_destino.py
-----
# src/silver/build_dim_dh_baseline_grado_destino.py

```

```

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def main() -> None:
    cfg = load_settings()
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    in_path = silver_dir / "fact_hidratacion_real_post_grado_destino.parquet"
    if not in_path.exists():
        raise FileNotFoundError(f"No existe: {in_path}. Ejecuta build_hidratacion_real_from_balanza2.py")

    fact = read_parquet(in_path).copy()
    fact.columns = [str(c).strip() for c in fact.columns]

    need = {"dh_dias", "grado", "destino"}
    miss = sorted(list(need - set(fact.columns)))
    if miss:
        raise ValueError(f"fact_hidratacion_real_post_grado_destino sin columnas: {miss}")

    fact["destino"] = fact["destino"].astype(str).str.strip().str.upper()
    fact["grado"] = pd.to_numeric(fact["grado"], errors="coerce").astype("Int64")
    fact["dh_dias"] = _to_num(fact["dh_dias"]).astype("Int64")

    fact = fact[
        fact["destino"].notna()
        & fact["grado"].notna()
        & fact["dh_dias"].notna()
        & fact["dh_dias"].between(0, 30)
    ].copy()

    out = (
        fact.groupby(["grado", "destino"], dropna=False)
        .agg(
            n=("dh_dias", "size"),
            dh_dias_med=("dh_dias", "median"),
            dh_dias_p25=("dh_dias", lambda s: int(np.nanpercentile(s.astype(float), 25))),
            dh_dias_p75=("dh_dias", lambda s: int(np.nanpercentile(s.astype(float), 75))),
        )
        .reset_index()
    )

    out["dh_dias_med"] = pd.to_numeric(out["dh_dias_med"], errors="coerce").round().astype("Int64")
    out["created_at"] = datetime.now().isoformat(timespec="seconds")

    out_path = silver_dir / "dim_dh_baseline_grado_destino.parquet"
    write_parquet(out, out_path)

    print(f"OK -> {out_path} | rows={len(out):,}")
    print(out.head(10).to_string(index=False))

if __name__ == "__main__":
    main()

```



```
-----  
[142/164] FILE: \src\silver\build_dim_dist_grado_baseline.py  
-----
```

```
from __future__ import annotations
```

```
from pathlib import Path  
from datetime import datetime  
import pandas as pd  
import numpy as np  
import yaml
```

```
from common.io import read_parquet, write_parquet
```

```
def load_settings() -> dict:  
    with open("config/settings.yaml", "r", encoding="utf-8") as f:  
        return yaml.safe_load(f)
```

```
def main() -> None:  
    cfg = load_settings()  
  
    silver_dir = Path(cfg["paths"]["silver"])  
    fact_path = silver_dir / "fact_cosecha_real_grado_dia.parquet"  
    if not fact_path.exists():  
        raise FileNotFoundError(f"No existe: {fact_path}. Ejecuta build_fact_cosecha_real_grado_dia primero.")  
  
    fact = read_parquet(fact_path).copy()  
  
    # Validaciones mínimas  
    needed = {"fecha", "variedad", "grado", "tallos_real"}  
    missing = needed - set(fact.columns)  
    if missing:  
        raise ValueError(f"fact_cosecha_real_grado_dia no tiene columnas requeridas: {missing}")  
  
    fact["variedad"] = fact["variedad"].astype(str).str.strip()  
    fact["grado"] = pd.to_numeric(fact["grado"], errors="coerce").astype("Int64")  
    fact["tallos_real"] = pd.to_numeric(fact["tallos_real"], errors="coerce").fillna(0.0)  
    fact = fact[(fact["grado"].notna()) & (fact["tallos_real"] > 0)].copy()  
  
    # 1) Totales por variedad y día (sumando grados)  
    day_tot = (fact.groupby(["variedad", "fecha"], dropna=False)  
                .agg(tallos_dia=("tallos_real", "sum"))  
                .reset_index())  
  
    # 2) Join para obtener % por grado por día  
    tmp = fact.merge(day_tot, on=["variedad", "fecha"], how="left")  
    tmp = tmp[tmp["tallos_dia"] > 0].copy()  
    tmp["pct_grado_dia"] = tmp["tallos_real"] / tmp["tallos_dia"]  
  
    # 3) Baseline por variedad+grado: usar mediana (robusto)  
    dim = (tmp.groupby(["variedad", "grado"], dropna=False)  
           .agg(  
               n_dias=("pct_grado_dia", "count"),  
               pct_grado=("pct_grado_dia", "median"),  
           )  
           .reset_index())  
  
    # 4) Normalización para que por variedad sume 1.0 (importante)  
    s = dim.groupby("variedad")["pct_grado"].sum().rename("sum_pct").reset_index()  
    dim = dim.merge(s, on="variedad", how="left")  
    dim["pct_grado"] = np.where(dim["sum_pct"] > 0, dim["pct_grado"] / dim["sum_pct"], dim["pct_grado"])  
    dim = dim.drop(columns=["sum_pct"])  
  
    dim["created_at"] = datetime.now().isoformat(timespec="seconds")  
  
    out_path = silver_dir / "dim_dist_grado_baseline.parquet"  
    write_parquet(dim, out_path)  
  
    # Auditoría  
    chk = dim.groupby("variedad")["pct_grado"].sum().describe()
```

```

print(f"OK: dim_dist_grado_baseline={len(dim)} filas -> {out_path}")
print("Suma pct por variedad (describe):\n", chk.to_string())

if __name__ == "__main__":
    main()

-----
[143/164] FILE: \src\silver\build_dim_estado_termico_cultivo_bloque_fecha.py
-----
from __future__ import annotations

from pathlib import Path
import pandas as pd
import numpy as np

from common.io import read_parquet, write_parquet

def main() -> None:
    created_at = pd.Timestamp.utcnow()

    df_maestro = read_parquet(Path("data/silver/fact_ciclo_maestro.parquet"))
    df_clima = read_parquet(Path("data/silver/dim_clima_bloque_dia.parquet"))
    df_grid = read_parquet(Path("data/silver/grid_ciclo_fecha.parquet"))

    # -----
    # Validaciones
    # -----
    need_m = {"ciclo_id", "bloque_base", "fecha_sp"}
    miss = need_m - set(df_maestro.columns)
    if miss:
        raise ValueError(f"fact_ciclo_maestro.parquet sin columnas: {sorted(miss)}")

    need_c = {"fecha", "bloque_base", "gdc_dia"}
    miss = need_c - set(df_clima.columns)
    if miss:
        raise ValueError(f"dim_clima_bloque_dia.parquet sin columnas: {sorted(miss)}")

    if "fecha" not in df_grid.columns or "ciclo_id" not in df_grid.columns:
        raise ValueError("grid_ciclo_fecha.parquet debe tener al menos columnas: fecha, ciclo_id")

    # -----
    # Normalización de fechas
    # -----
    m = df_maestro[["ciclo_id", "bloque_base", "fecha_sp"]].copy()
    m["fecha_sp"] = pd.to_datetime(m["fecha_sp"], errors="coerce").dt.normalize()

    # puede haber duplicados por bloque/ciclo: nos quedamos con 1 fecha_sp consistente
    m = (
        m.dropna(subset=["fecha_sp"])
        .groupby(["ciclo_id", "bloque_base"], as_index=False)
        .agg(fecha_sp=("fecha_sp", "min"))
    )

    g = df_grid[["ciclo_id", "fecha"]].copy()
    g["fecha"] = pd.to_datetime(g["fecha"], errors="coerce").dt.normalize()
    g = g.dropna(subset=["fecha"]).drop_duplicates()

    # Armamos calendario ciclo-fecha y lo cruzamos con bloques del ciclo
    # (para evitar inventar fechas fuera del ciclo)
    blocks = m[["ciclo_id", "bloque_base", "fecha_sp"]].copy()

    base = blocks.merge(g, on="ciclo_id", how="left")

    # Mantener solo fechas >= fecha_sp (térmicamente tiene sentido)
    base = base[base["fecha"] >= base["fecha_sp"]].copy()

    # -----
    # Join clima (gdc_dia)
    # -----

```

```

clima = df_clima[["fecha", "bloque_base", "gdc_dia"]].copy()
clima["fecha"] = pd.to_datetime(clima["fecha"], errors="coerce").dt.normalize()

out = base.merge(clima, on=["fecha", "bloque_base"], how="left")

# Si no hay clima para algún día, asumimos 0 GDC (o puedes dejar NaN)
out["gdc_dia"] = out["gdc_dia"].fillna(0.0)

# -----
# Cálculos desde S/P
# -----
out["dias_desde_sp"] = (out["fecha"] - out["fecha_sp"]).dt.days

out = out.sort_values(["ciclo_id", "bloque_base", "fecha"]).reset_index(drop=True)

out["gdc_acum_desde_sp"] = (
    out.groupby(["ciclo_id", "bloque_base"])["gdc_dia"].cumsum()
)

out["created_at"] = created_at

out = out[
    [
        "ciclo_id",
        "bloque_base",
        "fecha",
        "fecha_sp",
        "dias_desde_sp",
        "gdc_dia",
        "gdc_acum_desde_sp",
        "created_at",
    ]
].sort_values(["ciclo_id", "bloque_base", "fecha"]).reset_index(drop=True)

write_parquet(out, Path("data/silver/dim_estado_termico_cultivo_bloque_fecha.parquet"))
print(f"OK -> data/silver/dim_estado_termico_cultivo_bloque_fecha.parquet | rows={len(out):,}")

if __name__ == "__main__":
    main()

-----
[144/164] FILE: \src\silver\build_dim_factor_uph_cosecha_clima.py
-----
# src/silver/build_dim_factor_uph_cosecha_clima.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import write_parquet

# -----
# Config / helpers
# -----
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce")

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

```

```

def _hour_floor(dt: pd.Series) -> pd.Series:
    # pandas: "H" deprecado -> usar "h"
    return _to_dt(dt).dt.floor("h")

def _norm_station_from_area(area: pd.Series) -> pd.Series:
    """
    Regla de negocio:
    - A4/SJP/SAN JUAN => station=A4
    - else => MAIN
    """
    a = area.astype(str).str.upper().str.strip()
    is_a4 = a.isin(["A-4", "A4", "SJP", "SAN JUAN"])
    return pd.Series(np.where(is_a4, "A4", "MAIN"), index=area.index)

def _derive_variedad_from_codigo(cod: pd.Series) -> pd.Series:
    c = cod.astype(str).str.upper().str.strip()
    xl = c.isin(["ZCS", "ZCSP", "ZXL", "ZVX", "ZPPX", "ZCX"])
    clo = c.isin(["ZMP", "ZVP", "ZPC", "ZCC"])
    out = np.where(xl, "XL", np.where(clo, "CLO", np.nan))
    return pd.Series(out, index=cod.index)

def _derive_pelado_from_codigo(cod: pd.Series) -> pd.Series:
    c = cod.astype(str).str.upper().str.strip()
    return (c == "ZCSP").astype(int)

def _q(x: pd.Series, q: float) -> float:
    x = x.dropna()
    return float(x.quantile(q)) if len(x) else np.nan

def _pick_col(df: pd.DataFrame, candidates: list[str], required: bool = True) -> str | None:
    """
    Devuelve el nombre real de la columna que existe en df, probando candidatos.
    Si required=True y no encuentra, lanza error con columnas disponibles.
    """
    cols = list(df.columns)
    cols_l = {c.lower(): c for c in cols}
    for cand in candidates:
        if cand in cols:
            return cand
        cl = cand.lower()
        if cl in cols_l:
            return cols_l[cl]
    if required:
        raise ValueError(
            "No se encontró ninguna columna válida. "
            f"Candidatos={candidates}. Columnas disponibles={cols}"
        )
    return None

def _normalize_area_trabajada(df: pd.DataFrame) -> pd.DataFrame:
    """
    Normaliza area_trabajada y aplica whitelist para MAIN.
    Reglas negocio:
    - A4 puede venir como A-4 / A4 / SJP / SAN JUAN
    - MAIN (cosecha) debe estar en {MH1, MH2, CULTIVOS VARIOS}
    - Otros valores se marcan como NaN (y se excluyen por baseline/agrupación)
    """
    df = df.copy()

    if "area_trabajada" not in df.columns:
        return df

    a = df["area_trabajada"].astype(str).str.upper().str.strip()

    # normalizaciones comunes

```

```

a = a.replace(
    {
        "A-4": "A4",
        "SANJUAN": "SAN JUAN",
        "SAN_JUAN": "SAN JUAN",
    }
)

df["area_trabajada"] = a

# whitelist MAIN
main_ok = {"MH1", "MH2", "CULTIVOS VARIOS"}
is_main = df.get("station", pd.Series(index=df.index, dtype="object")).astype(str).str.upper().eq("MAIN")
is_a4 = df.get("station", pd.Series(index=df.index, dtype="object")).astype(str).str.upper().eq("A4")

# Para MAIN: si no está en whitelist, poner NaN (evita sesgos tipo EMP/ORN)
df.loc[is_main & (~df["area_trabajada"].isin(list(main_ok))), "area_trabajada"] = np.nan

# Para A4: dejarlo como venga (A4/SJP/SAN JUAN ya normalizados), no necesitamos whitelist aquí
# (si quisieras, podrías forzar df.loc[is_a4, "area_trabajada"]="A4")

return df

def _ensure_station_from_hours(df: pd.DataFrame) -> pd.DataFrame:
    """
    Station debe venir de area_trabajada (estricto).
    NO usamos area_original para evitar contaminación.
    """
    df = df.copy()

    if "area_trabajada" not in df.columns:
        raise ValueError(
            "Falta columna 'area_trabajada' en el input de cosecha horaria. "
            "No puedo inferir station sin esa columna."
        )

    df["station"] = _norm_station_from_area(df["area_trabajada"])
    df["station"] = df["station"].astype(str).str.upper().str.strip()
    df = df[df["station"].isin(["MAIN", "A4"])]

    # normalizar area_trabajada + whitelist MAIN
    df = _normalize_area_trabajada(df)

    return df

def _baseline_from_areas(
    m: pd.DataFrame,
    q_baseline_area: float = 0.75,
    min_n_area: int = 500,
) -> pd.DataFrame:
    """
    Baseline robusto:
    1) Filtra condiciones ideales: SECO, sin lluvia, no pelado
    2) Calcula mediana por (station,variedad,area_trabajada)
    3) Se queda con áreas con n>=min_n_area
    4) Baseline final por (station,variedad) = cuantil q_baseline_area de esas medianas de área

    Esto evita que la "mediana global" se deprima por colas/heterogeneidad.
    """
    base_mask = (m["Estado_Kardex"] == "SECO") & (m["En_Lluvia"] == 0) & (m["pelado"] == 0)

    ideal = m[base_mask].copy()

    if "area_trabajada" not in ideal.columns:
        raise ValueError("Para baseline por áreas, falta area_trabajada en el input (post-merge).")

    # Excluir áreas no válidas (NaN) para baseline por área
    ideal = ideal[ideal["area_trabajada"].notna()].copy()

    base_area = (

```

```

ideal.groupby(["station", "variedad", "area_trabajada"], dropna=False)
.agg(
    uph_mediana_area=("uph", "median"),
    n_area=("uph", "size"),
)
.reset_index()
)

# Filtrar áreas con volumen mínimo para que no dominen áreas raras o con poca data
base_area_ok = base_area[base_area["n_area"] >= int(min_n_area)].copy()

if base_area_ok.empty:
    # fallback conservador: usar la base_area sin filtro n
    base_area_ok = base_area.copy()

# Baseline final: cuantil alto de las medianas por área
base = (
    base_area_ok.groupby(["station", "variedad"], dropna=False)
    .agg(
        uph_base_mediana=("uph_mediana_area", lambda s: float(s.quantile(q_baseline_area)) if len(s) else
np.nan),
        n_base=("n_area", "sum"),
        n_areas=("area_trabajada", "nunique"),
    )
    .reset_index()
)

return base

# -----
# Main
# -----
def main() -> None:
    cfg = load_settings()

    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    # Inputs
    uph_fname = cfg.get("cosecha", {}).get("uph_hour_file", "fact_cosecha_uph_hora_clima.parquet")
    uph_path = silver_dir / uph_fname
    weather_path = silver_dir / "weather_hour_wide.parquet"

    if not uph_path.exists():
        raise FileNotFoundError(
            f"No existe input de UPH cosecha por hora: {uph_path}\n"
            f"Config sugerida: cosecha.uph_hour_file en settings.yaml"
        )
    if not weather_path.exists():
        raise FileNotFoundError(f"No existe weather_hour_wide: {weather_path}")

    df = pd.read_parquet(uph_path)
    w = pd.read_parquet(weather_path)

    df.columns = [str(c).strip() for c in df.columns]
    w.columns = [str(c).strip() for c in w.columns]

    # -----
    # Normalización df (cosecha)
    # -----
    if "dt_hora" in df.columns:
        df["dt_hora"] = _hour_floor(df["dt_hora"])
    elif "fecha_hora" in df.columns:
        df["dt_hora"] = _hour_floor(df["fecha_hora"])
    elif "fecha" in df.columns and "hora_n" in df.columns:
        df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
        df["hora_n"] = _to_num(df["hora_n"]).astype("Int64")
        df["dt_hora"] = df["fecha"] + pd.to_timedelta(df["hora_n"].fillna(0).astype(int), unit="h")
        df["dt_hora"] = _hour_floor(df["dt_hora"])
    else:
        raise ValueError("El input de cosecha debe traer 'dt_hora' o ('fecha' y 'hora_n') o 'fecha_hora'.")

```

```

# station (estricto desde area_trabajada)
df = _ensure_station_from_hours(df)

# actividad -> variedad/pelado
if "codigo_actividad" not in df.columns:
    raise ValueError("Falta columna codigo_actividad en input de cosecha horaria.")
df["codigo_actividad"] = df["codigo_actividad"].astype(str).str.upper().str.strip()

if "variedad" in df.columns:
    df["variedad"] = df["variedad"].astype(str).str.upper().str.strip()
    df["variedad"] = df["variedad"].replace({"XXLENCE": "XL", "XLENCE": "XL", "CLOUD": "CLO"})
else:
    df["variedad"] = _derive_variedad_from_codigo(df["codigo_actividad"])

if "pelado" in df.columns:
    df["pelado"] = _to_num(df["pelado"]).fillna(0).astype(int)
else:
    df["pelado"] = _derive_pelado_from_codigo(df["codigo_actividad"])

df = df[df["variedad"].isin(["XL", "CLO"])].copy()

# Validación negocio: A4 NO puede tener CLO
bad_a4_clo = df[(df["station"] == "A4") & (df["variedad"] == "CLO")]
if len(bad_a4_clo) > 0:
    ex = bad_a4_clo[["dt_hora", "codigo_actividad", "variedad", "station", "area_trabajada"]].head(10)
    raise ValueError(
        "Detectado CLO en station=A4 (esto no debería ocurrir según negocio). "
        "Revisa mapping de area_trabajada y/o códigos. Ejemplos:\n"
        f"{ex.to_string(index=False)}"
    )

# UPH
if "UP_por_hora_ajustada" in df.columns:
    df["uph"] = _to_num(df["UP_por_hora_ajustada"])
elif "uph_ajustada_tramo" in df.columns:
    df["uph"] = _to_num(df["uph_ajustada_tramo"])
else:
    if "UP_tramo_ajustada" not in df.columns:
        raise ValueError("Falta UP_por_hora_ajustada/uph_ajustada_tramo y UP_tramo_ajustada en input.")
    if "horas_tramo" not in df.columns:
        raise ValueError("Falta horas_tramo para calcular UPH.")
    df["UP_tramo_ajustada"] = _to_num(df["UP_tramo_ajustada"])
    df["horas_tramo"] = _to_num(df["horas_tramo"])
    df["uph"] = np.where(df["horas_tramo"] > 0, df["UP_tramo_ajustada"] / df["horas_tramo"], np.nan)

df["uph"] = _to_num(df["uph"])
df = df[df["uph"].notna()].copy()
df = df[(df["uph"] > 10) & (df["uph"] < 2000)].copy()

# -----
# Normalización clima (weather_hour_wide)
# -----
col_w_dt = _pick_col(w, ["fecha", "dt_hora", "datetime", "timestamp"])
w["dt_hora"] = _hour_floor(w[col_w_dt])

col_w_station = _pick_col(w, ["station", "estacion", "ws_station", "station_code"])
w["station"] = w[col_w_station].astype(str).str.upper().str.strip()
w = w[w["station"].isin(["MAIN", "A4"])].copy()

col_estado = _pick_col(
    w,
    ["Estado_Kardex", "estado_kardex", "estado", "estado_clima", "kardex_estado", "Estado"],
    required=True,
)
col_lluvia = _pick_col(
    w,
    ["En_Lluvia", "en_lluvia", "lluvia", "is_rain", "rain_flag"],
    required=True,
)

w["Estado_Kardex"] = w[col_estado].astype(str).str.upper().str.strip().replace({"HÚMEDO": "HUMEDO"})

```

```

w["En_Lluvia"] = _to_num(w[col_lluvia]).fillna(0).astype(int)

w2 = (
    w[["dt_hora", "station", "Estado_Kardex", "En_Lluvia"]]
    .dropna(subset=["dt_hora", "station"])
    .drop_duplicates(subset=["dt_hora", "station"], keep="last")
    .copy()
)

# -----
# Join por hora + station
# -----
for c in ["Estado_Kardex", "En_Lluvia", "estado_final"]:
    if c in df.columns:
        df = df.rename(columns={c: f"{c}_src"})

m = df.merge(w2, on=["dt_hora", "station"], how="left")

if "Estado_Kardex" not in m.columns:
    raise ValueError(f"Post-merge: no existe Estado_Kardex. Columnas={list(m.columns)}")
if "En_Lluvia" not in m.columns:
    raise ValueError(f"Post-merge: no existe En_Lluvia. Columnas={list(m.columns)}")

m["__miss_weather"] = m["Estado_Kardex"].isna().astype(int)
m["Estado_Kardex"] = m["Estado_Kardex"].fillna("SECO").astype(str).str.upper().str.strip()
m["En_Lluvia"] = pd.to_numeric(m["En_Lluvia"], errors="coerce").fillna(0).astype(int)
m["estado_final"] = np.where(m["En_Lluvia"].eq(1), "LLUVIA", m["Estado_Kardex"])

# -----
# Baseline robusto (POR ÁREA) en condiciones ideales
# -----
# Parámetros (opcionales) desde settings.yaml:
# cosecha:
#     baseline_area_quantile: 0.75
#     baseline_area_min_n: 500
q_baseline = float(cfg.get("cosecha", {}).get("baseline_area_quantile", 0.75))
min_n_area = int(cfg.get("cosecha", {}).get("baseline_area_min_n", 500))

base = _baseline_from_areas(m, q_baseline_area=q_baseline, min_n_area=min_n_area)

if base.empty or base["uph_base_mediana"].isna().all():
    raise ValueError(
        "No se pudo calcular baseline robusto por áreas. "
        "Revisa si existen registros SECO/sin lluvia/no pelado y áreas válidas."
    )

# -----
# Agregación por celda clima
# -----
g = (
    m.groupby(["station", "variedad", "pelado", "Estado_Kardex", "En_Lluvia", "estado_final"], dropna=False)
    .agg(
        uph_mediana=("uph", "median"),
        uph_p25=("uph", lambda s: _q(s, 0.25)),
        uph_p75=("uph", lambda s: _q(s, 0.75)),
        n_obs=("uph", "size"),
        n_miss_weather=("__miss_weather", "sum"),
    )
    .reset_index()
)

out = g.merge(base[["station", "variedad", "uph_base_mediana", "n_base", "n_areas"]], on=["station", "variedad"], how="left")
out["factor_uph"] = out["uph_mediana"] / out["uph_base_mediana"]
out["factor_uph"] = out["factor_uph"].clip(lower=0.30, upper=1.20)

out["created_at"] = datetime.now().isoformat(timespec="seconds")

cols = [
    "station", "variedad", "pelado",
    "Estado_Kardex", "En_Lluvia", "estado_final",

```



```

    "uph_base_mediana", "n_base", "n_areas",
    "uph_mediana", "uph_p25", "uph_p75",
    "factor_uph",
    "n_obs", "n_miss_weather",
    "created_at",
]
out = (
    out[cols]
    .sort_values(["station", "variedad", "pelado", "En_Lluvia", "Estado_Kardex"])
    .reset_index(drop=True)
)

out_path = silver_dir / "dim_factor_uph_cosecha_clima.parquet"
write_parquet(out, out_path)

base2 = base.copy()
base2["created_at"] = datetime.now().isoformat(timespec="seconds")
base_path = silver_dir / "dim_baseline_uph_cosecha.parquet"
write_parquet(base2, base_path)

# Logs
print(f"OK: dim_factor_uph_cosecha_clima={len(out)} filas -> {out_path}")
print(f"OK: dim_baseline_uph_cosecha={len(base2)} filas -> {base_path}")
print("Baseline (uph_base_mediana) por station/variedad (robusto por áreas):")
print(base2.sort_values(["station", "variedad"]).to_string(index=False))
print("Resumen factor_uph:")
print(out["factor_uph"].describe().to_string())

miss = int(out["n_miss_weather"].sum())
if miss > 0:
    print(f"[WARN] Hay {miss} tramos con clima faltante (imputados a SECO/0). Revisa cobertura de weather_
hour_wide.")

if __name__ == "__main__":
    main()

-----
[145/164] FILE: \src\silver\build_dim_hidratacion_baseline_grado_destino.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd
import yaml

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_int(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce").astype("Int64")

def main() -> None:
    cfg = load_settings()
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    # Fuente: fact real (mejor) o dim por fecha_cosecha si ya existe
    fact_path = silver_dir / "fact_hidratacion_real_post_grado_destino.parquet"

```

```

dim_fc_path = silver_dir / "dim_hidratacion_fecha_cosecha_grado_destino.parquet"

if fact_path.exists():
    df = pd.read_parquet(fact_path)
    df.columns = [str(c).strip() for c in df.columns]

    need = {"grado", "destino", "hidr_pct", "peso_base_g"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"fact_hidratacion_real_post_grado_destino sin columnas: {sorted(miss)}")

    df["grado"] = _canon_int(df["grado"])
    df["destino"] = df["destino"].astype(str).str.upper().str.strip()
    df["hidr_pct"] = pd.to_numeric(df["hidr_pct"], errors="coerce")
    df["peso_base_g"] = pd.to_numeric(df["peso_base_g"], errors="coerce").fillna(0.0)

    df = df[df["grado"].notna() & df["destino"].notna() & df["hidr_pct"].notna()].copy()
    if df.empty:
        raise ValueError("fact hidratación quedó vacío tras filtros.")

    # Peso como ponderador (si no, usa count)
    df["_w"] = df["peso_base_g"].clip(lower=0.0)

    # Cuantiles ponderados: aproximación robusta vía expansión por bins NO es viable.
    # Aquí usamos cuantiles simples + mediana ponderada aproximada por ranking con peso.
    # (Suficiente para baseline seed; ML vendrá después.)
    out_rows = []
    for (g, d), gdf in df.groupby(["grado", "destino"], dropna=False):
        x = gdf["hidr_pct"].to_numpy(dtype=float)
        w = gdf["_w"].to_numpy(dtype=float)
        if len(x) == 0:
            continue
        # fallback si pesos degeneran
        if np.isnansum(w) <= 0:
            med = float(np.nanmedian(x))
        else:
            # mediana ponderada
            o = np.argsort(x)
            x2 = x[o]
            w2 = w[o]
            cw = np.cumsum(np.nan_to_num(w2, nan=0.0))
            cut = 0.5 * cw[-1]
            med = float(x2[np.searchsorted(cw, cut, side="left")])

        p25 = float(np.nanpercentile(x, 25))
        p75 = float(np.nanpercentile(x, 75))
        out_rows.append(
            {
                "grado": int(g),
                "destino": str(d),
                "n": int(len(gdf)),
                "peso_base_g_sum": float(np.isnansum(w)),
                "hidr_pct_med": med,
                "hidr_pct_p25": p25,
                "hidr_pct_p75": p75,
                "factor_hidr_med": 1.0 + med,
                "factor_hidr_p25": 1.0 + p25,
                "factor_hidr_p75": 1.0 + p75,
            }
        )

    out = pd.DataFrame(out_rows)
elif dim_fc_path.exists():
    # fallback si no tienes fact
    df = pd.read_parquet(dim_fc_path)
    df.columns = [str(c).strip() for c in df.columns]

    need = {"grado", "destino", "factor_hidr"}
    miss = need - set(df.columns)
    if miss:
        raise ValueError(f"dim_hidratacion_fecha_cosecha_grado_destino sin columnas: {sorted(miss)}")

```

```

df["grado"] = _canon_int(df["grado"])
df["destino"] = df["destino"].astype(str).str.upper().str.strip()
df["factor_hidr"] = pd.to_numeric(df["factor_hidr"], errors="coerce")
df = df[df["grado"].notna() & df["destino"].notna() & df["factor_hidr"].notna()].copy()
df["hidr_pct"] = df["factor_hidr"] - 1.0

out = (
    df.groupby(["grado", "destino"], dropna=False)
        .agg(
            n=("factor_hidr", "size"),
            factor_hidr_med=("factor_hidr", "median"),
            factor_hidr_p25=("factor_hidr", lambda x: float(np.nanpercentile(x, 25))),
            factor_hidr_p75=("factor_hidr", lambda x: float(np.nanpercentile(x, 75))),
            hidr_pct_med=("hidr_pct", "median"),
        )
        .reset_index()
)
else:
    raise FileNotFoundError("No existe fact ni dim de hidratación para construir baseline.")

out["created_at"] = datetime.now().isoformat(timespec="seconds")
out_path = silver_dir / "dim_hidratacion_baseline_grado_destino.parquet"
write_parquet(out, out_path)

print(f"OK -> {out_path} | rows={len(out):,}")
if "factor_hidr_med" in out.columns:
    print("factor_hidr_med describe:\n", pd.to_numeric(out["factor_hidr_med"], errors="coerce").describe()
.to_string())

if __name__ == "__main__":
    main()

-----
[146/164] FILE: \src\silver\build_dim_hidratacion_fecha_cosecha_grado_destino.py
-----
# src/silver/build_dim_hidratacion_fecha_cosecha_grado_destino.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def main() -> None:
    cfg = load_settings()
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    in_path = silver_dir / "fact_hidratacion_real_post_grado_destino.parquet"
    if not in_path.exists():
        raise FileNotFoundError(f"No existe: {in_path}. Ejecuta build_hidratacion_real_from_balanza2.py")

    fact = read_parquet(in_path).copy()

```

```

fact.columns = [str(c).strip() for c in fact.columns]

need = {"fecha_cosecha", "grado", "destino", "hidr_pct", "peso_base_g"}
miss = sorted(list(need - set(fact.columns)))
if miss:
    raise ValueError(f"fact_hidratacion_real_post_grado_destino sin columnas: {miss}")

fact["fecha_cosecha"] = _norm_date(fact["fecha_cosecha"])
fact["destino"] = fact["destino"].astype(str).str.strip().str.upper()
fact["grado"] = pd.to_numeric(fact["grado"], errors="coerce").astype("Int64")
fact["hidr_pct"] = _to_num(fact["hidr_pct"])
fact["peso_base_g"] = _to_num(fact["peso_base_g"]).fillna(0.0)

fact = fact[
    fact["fecha_cosecha"].notna()
    & fact["grado"].notna()
    & fact["destino"].notna()
    & fact["hidr_pct"].notna()
    & (fact["peso_base_g"] > 0)
].copy()

fact["_w"] = fact["peso_base_g"]
fact["_hydr_w"] = fact["hidr_pct"] * fact["_w"]

out = (
    fact.groupby(["fecha_cosecha", "grado", "destino"], dropna=False)
        .agg(
            n=("hidr_pct", "size"),
            tallos=("tallos", "sum") if "tallos" in fact.columns else ("hidr_pct", "size"),
            peso_base_g=("peso_base_g", "sum"),
            sum_hydr_w=("_hydr_w", "sum"),
            sum_w=("_w", "sum"),
        )
        .reset_index()
)

out["hidr_pct"] = np.where(out["sum_w"] > 0, out["sum_hydr_w"] / out["sum_w"], np.nan)
out = out.drop(columns=["sum_hydr_w", "sum_w"], errors="ignore")
out["factor_hidr"] = (1.0 + out["hidr_pct"]).clip(0.8, 3.0) # cap amplio
out["created_at"] = datetime.now().isoformat(timespec="seconds")

out_path = silver_dir / "dim_hidratacion_fecha_cosecha_grado_destino.parquet"
write_parquet(out, out_path)

print(f"OK -> {out_path} | rows={len(out):,}")
print("factor_hidr describe:\n", out["factor_hidr"].describe().to_string())

```

```

if __name__ == "__main__":
    main()

```

```

-----
[147/164] FILE: \src\silver\build_dim_mediana_etapas_tipo_sp_variedad_area.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

```

```

from common.io import read_parquet, write_parquet

```

```

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

```

```

def _to_date(s: pd.Series) -> pd.Series:

```

```

return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.strip().str.upper()

def main() -> None:
    cfg = load_settings()
    silver_dir = Path(cfg["paths"]["silver"])

    milestones_path = silver_dir / "milestones_ciclo_final.parquet"
    maestro_path = silver_dir / "fact_ciclo_maestro.parquet"

    if not milestones_path.exists():
        raise FileNotFoundError(f"No existe: {milestones_path}")
    if not maestro_path.exists():
        raise FileNotFoundError(f"No existe: {maestro_path}")

    m = read_parquet(milestones_path).copy()
    c = read_parquet(maestro_path).copy()

    # Normalizar
    m["ciclo_id"] = m["ciclo_id"].astype(str)
    m["milestone_code"] = _canon_str(m["milestone_code"])
    m["fecha"] = _to_date(m["fecha"])

    c["ciclo_id"] = c["ciclo_id"].astype(str)
    for col in ["tipo_sp", "variedad", "area"]:
        if col not in c.columns:
            c[col] = "UNKNOWN"
        c[col] = _canon_str(c[col])

    # Mapa variedad (si existe)
    var_map = (cfg.get("mappings", {}).get("variedad_map", {}) or {})
    var_map = {str(k).strip().upper(): str(v).strip().upper() for k, v in var_map.items()}
    c["variedad_std"] = c["variedad"].map(lambda x: var_map.get(x, x))

    # Pivot milestones
    piv = (
        m.pivot_table(index="ciclo_id", columns="milestone_code", values="fecha", aggfunc="min")
        .reset_index()
    )
    for col in ["VEG_START", "HARVEST_START", "HARVEST_END", "POST_START", "POST_END"]:
        if col in piv.columns:
            piv[col] = _to_date(piv[col])

    # Traer segmentación
    seg = c[["ciclo_id", "tipo_sp", "variedad_std", "area"]].drop_duplicates("ciclo_id")
    piv = piv.merge(seg, on="ciclo_id", how="left")

    # Duraciones
    piv["dias_veg"] = (piv["HARVEST_START"] - piv["VEG_START"]).dt.days
    piv["dias_harvest"] = (piv["HARVEST_END"] - piv["HARVEST_START"]).dt.days + 1
    piv["dias_post"] = (piv["POST_END"] - piv["POST_START"]).dt.days + 1

    # Limpieza básica
    piv.loc[piv["dias_veg"] < 0, "dias_veg"] = np.nan
    piv.loc[piv["dias_harvest"] <= 0, "dias_harvest"] = np.nan
    piv.loc[piv["dias_post"] <= 0, "dias_post"] = np.nan

    base = piv[["tipo_sp", "variedad_std", "area", "dias_veg", "dias_harvest", "dias_post"]].copy()
    base = base.dropna(subset=["dias_veg", "dias_harvest"]) # post puede ser opcional

    if base.empty:
        raise ValueError("No hay ciclos con duraciones válidas para calcular medianas.")

    out = (
        base.groupby(["tipo_sp", "variedad_std", "area"], dropna=False)
        .agg(
            mediana_dias_veg=("dias_veg", "median"),
            mediana_dias_harvest=("dias_harvest", "median"),

```

```

        mediana_dias_post=("dias_post", "median"),
        n=("dias_veg", "count"),
    )
    .reset_index()
)

# Caps razonables (evita basura)
out["mediana_dias_veg"] = pd.to_numeric(out["mediana_dias_veg"], errors="coerce").clip(0, 180)
out["mediana_dias_harvest"] = pd.to_numeric(out["mediana_dias_harvest"], errors="coerce").clip(1, 180)
out["mediana_dias_post"] = pd.to_numeric(out["mediana_dias_post"], errors="coerce").clip(1, 365)

out["created_at"] = datetime.now().isoformat(timespec="seconds")

out_path = silver_dir / "dim_mediana_etapas_tipo_sp_variedad_area.parquet"
write_parquet(out, out_path)

print(f"OK: {out_path} | rows={len(out):,}")
print(out[["n"]].describe().to_string())

if __name__ == "__main__":
    main()

-----
[148/164] FILE: \src\silver\build_dim_mermas_ajuste_fecha_post.py
-----
# src/silver/build_dim_mermas_ajuste_fecha_post.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    p2a = bronze_dir / "balanza_2a_raw.parquet"
    p2 = bronze_dir / "balanza_2_raw.parquet"

    if not p2a.exists():
        raise FileNotFoundError(f"No existe Bronze: {p2a}. Ejecuta src/bronze/build_balanza_mermas_sources.py")
    )

    if not p2.exists():
        raise FileNotFoundError(f"No existe Bronze: {p2}. Ejecuta src/bronze/build_balanza_mermas_sources.py")

    df2a = pd.read_parquet(p2a)
    df2 = pd.read_parquet(p2)

    df2a.columns = [str(c).strip() for c in df2a.columns]

```

```

df2.columns = [str(c).strip() for c in df2.columns]

# -----
# BALANZA 2A (AgrupadoFinal por fecha_post+destino)
# -----
df2a["Fecha"] = _norm_date(df2a["Fecha"])
df2a["Origen"] = df2a["Origen"].astype(str).strip()
df2a["Seccion"] = df2a["Seccion"].astype(str).strip().str.upper()
df2a["Variedad"] = df2a["Variedad"].astype(str).strip().str.upper()
df2a["codigo_actividad"] = df2a["codigo_actividad"].astype(str).strip().str.lower()
df2a["Grado"] = df2a["Grado"].astype(str).strip()

base = df2a[
    (df2a["Fecha"].notna())
    & (df2a["Origen"] != "GV PELADO")
    & (df2a["Seccion"] == "CLASIFICACION")
    & (df2a["Variedad"] == "XLENCE")
    & (df2a["Fecha"] >= pd.Timestamp("2025-01-01"))
].copy()

# ReemplazoActividad + filtro psmc
base = base[base["codigo_actividad"] != "psmc"].copy()

rep = {
    "cbx": "BLANCO",
    "cxlta1": "TINTURADO",
    "05cts": "TINTURADO",
    "0504gufdgu": "GUIRNALDA",
    "cxlta1h": "ARCOIRIS",
}
base["codigo_actividad_reemplazo"] = base["codigo_actividad"].map(rep).fillna(base["codigo_actividad"].astype(str).str.upper())

base["peso_balanza"] = _to_num(base["peso_balanza"])
base["tallos"] = _to_num(base["tallos"])
base["num_bunches"] = _to_num(base["num_bunches"])

base["PESOKG"] = base["peso_balanza"] / 1000.0
base["TALLOSTOTALES"] = base["tallos"] * base["num_bunches"]

agr0 = (
    base.groupby(["Fecha", "codigo_actividad_reemplazo", "Grado"], dropna=False)
        .agg(
            Peso=("PESOKG", "sum"),
            Tallos=("TALLOSTOTALES", "sum"),
            bunches=("num_bunches", "sum"),
        )
        .reset_index()
        .rename(columns={"codigo_actividad_reemplazo": "codigo_actividad"})
)

grado = agr0["Grado"].astype(str).strip()
is_pet = grado.eq("PET")
is_bqt = grado.eq("BQT")
has_gr = grado.str.contains("GR", na=False)

agr0["Grado2"] = np.where(
    is_pet, "PET",
    np.where(is_bqt, "BQT",
        np.where(has_gr, grado, grado + "GR"))
)

g2 = agr0["Grado2"].astype(str)
num_gr = pd.to_numeric(
    g2.where(g2.str.contains("GR", na=False), np.nan)
        .str.split("GR").str[0],
    errors="coerce"
)

agr0["PESO_IDEAL"] = np.where(
    g2.str.contains("BQT", na=False), (agr0["Tallos"].astype(float) / 800.0) * 10.0,
    np.where(

```

```

        g2.str.contains("PET", na=False), (agr0["Tallos"].astype(float) / 1000.0) * 10.0,
        np.where(
            g2.str.contains("GUARNALDA", na=False), agr0["Peso"],
            np.where(g2.str.contains("GR", na=False), num_gr, np.nan)
        )
    )
)

agr0["PESIDEALTOTALKG"] = np.where(
    agr0["Grado2"].isin(["BQT", "PET", "GUARNALDAGR"]),
    agr0["PESO_IDEAL"],
    (agr0["PESO_IDEAL"] * agr0["bunches"]) / 1000.0
)

a = (
    agr0.groupby(["Fecha", "codigo_actividad"], dropna=False)
        .agg(
            w2a_kg=("Peso", "sum"),
            wideal_kg=("PESIDEALTOTALKG", "sum"),
        )
        .reset_index()
        .rename(columns={"Fecha": "fecha_post", "codigo_actividad": "destino"})
)
a["destino"] = a["destino"].astype(str).str.strip().str.upper()

# -----
# BALANZA 2 (w2_kg por fecha_post+destino)
# -----
df2["fecha_entrega"] = _norm_date(df2["fecha_entrega"])
df2["Destino"] = df2["Destino"].astype(str).str.strip().str.upper()
df2["variedad"] = df2["variedad"].astype(str).str.strip().str.upper()
df2["tipo_pelado"] = df2["tipo_pelado"].astype(str).str.strip()
df2["Origen"] = df2["Origen"].astype(str).str.strip().str.upper()

prod = df2.get("producto")
if prod is None:
    df2["producto"] = np.nan
    prod = df2["producto"]
prod = prod.astype("string")

b2 = df2[
    (df2["fecha_entrega"].notna())
    & (df2["variedad"] == "GYPXLE")
    & (df2["tipo_pelado"] == "Sin Pelar")
    & (df2["Origen"] == "APERTURA")
    & (prod.isna() | (prod.astype(str).str.upper().eq("GUARNALDA")))
].copy()

b2["destino"] = np.where(b2["Destino"].isin(["GUARNALDA", "CLASIFICACION"]), "BLANCO", b2["Destino"])
b2["peso_netto"] = _to_num(b2["peso_netto"])

b = (
    b2.groupby(["fecha_entrega", "destino"], dropna=False)
        .agg(w2_kg=("peso_netto", "sum"))
        .reset_index()
        .rename(columns={"fecha_entrega": "fecha_post"})
)
b["destino"] = b["destino"].astype(str).str.strip().str.upper()

# -----
# Join por fecha_post+destino y cálculo por destino
# -----
m = a.merge(b, on=["fecha_post", "destino"], how="left")

out0 = (
    m.groupby(["fecha_post", "destino"], dropna=False)
        .agg(
            w2_kg=("w2_kg", "sum"),
            w2a_kg=("w2a_kg", "sum"),
            wideal_kg=("wideal_kg", "sum"),
        )
        .reset_index()
)

```



```

        .sort_values(["fecha_post", "destino"], ascending=[False, True])
    )

    out0["fecha_post"] = _norm_date(out0["fecha_post"])
    for c in ["w2_kg", "w2a_kg", "wideal_kg"]:
        out0[c] = _to_num(out0[c])

    out0 = out0[out0["fecha_post"].notna()].copy()
    out0 = out0[out0["w2_kg"].notna() & (out0["w2_kg"] > 0)].copy()
    out0 = out0[out0["w2a_kg"].notna() & (out0["w2a_kg"] > 0)].copy()
    out0 = out0[out0["wideal_kg"].notna() & (out0["wideal_kg"] > 0)].copy()

    out0["desp_pct"] = (1.0 - (out0["w2a_kg"] / out0["w2_kg"])).clip(0.0, 0.95)
    out0["factor_desp"] = (out0["w2a_kg"] / out0["w2_kg"]).clip(0.05, 1.0)

    out0["ajuste"] = (out0["wideal_kg"] / out0["w2a_kg"]).clip(0.5, 2.0)
    out0["factor_ajuste"] = (out0["w2a_kg"] / out0["wideal_kg"]).clip(0.5, 2.0)

    # Mantener solo procesos relevantes
    keep_proc = {"BLANCO", "ARCOIRIS", "TINTURADO"}
    out0 = out0[out0["destino"].isin(sorted(keep_proc))].copy()

    out0["created_at"] = datetime.now().isoformat(timespec="seconds")

    out = out0[[
        "fecha_post", "destino",
        "w2_kg", "w2a_kg", "wideal_kg",
        "desp_pct", "factor_desp",
        "ajuste", "factor_ajuste",
        "created_at",
    ]].copy()

    out_path = silver_dir / "dim_mermas_ajuste_fecha_post_destino.parquet"
    write_parquet(out, out_path)

    print(f"OK: dim_mermas_ajuste_fecha_post_destino={len(out)} filas -> {out_path}")
    print("destinos:", out["destino"].value_counts().to_dict())
    print("desp_pct describe:\n", out["desp_pct"].describe().to_string())
    print("ajuste describe:\n", out["ajuste"].describe().to_string())

if __name__ == "__main__":
    main()

-----
[149/164] FILE: \src\silver\build_dim_mix_proceso_semana_from_ventas.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml
import re

from common.io import write_parquet

# =====
# Settings
# =====
ROOT = Path(__file__).resolve().parents[2] # ../src/silver -> repo root
SETTINGS_PATH = ROOT / "config" / "settings.yaml"

def load_settings() -> dict:
    p = SETTINGS_PATH if SETTINGS_PATH.exists() else Path("config/settings.yaml")
    if not p.exists():
        raise FileNotFoundError(f"No existe settings.yaml en {p} (ni en config/settings.yaml).")
    with open(p, "r", encoding="utf-8") as f:

```

```

    return yaml.safe_load(f) or {}

# =====
# Utils
# =====
def _norm_colname(c: str) -> str:
    c = str(c).strip().lower()
    c = re.sub(r"\s+", " ", c)
    return c

def _find_col(df: pd.DataFrame, candidates: list[str]) -> str | None:
    norm_map = {_norm_colname(c): c for c in df.columns}
    for cand in candidates:
        key = _norm_colname(cand)
        if key in norm_map:
            return norm_map[key]
    return None

def semana_ventas(fecha_clasificacion: pd.Series) -> pd.Series:
    """
    Semana ventas según tu regla: fecha_clasificacion + 2 días, y semana %U.
    FIX:
    - evitar np.where (ndarray) + Series sin index (alineación rara)
    - soportar NaT
    """
    d = pd.to_datetime(fecha_clasificacion, errors="coerce") + pd.Timedelta(days=2)

    yy = (d.dt.year % 100).astype("Int64")
    ww = pd.to_numeric(d.dt.strftime("%U"), errors="coerce").astype("Int64")
    ww = ww.where(ww.notna() & (ww > 0), 1)

    yy_s = yy.astype("string").str.zfill(2)
    ww_s = ww.astype("string").str.zfill(2)

    out = yy_s + ww_s
    out = out.where(d.notna(), pd.NA)
    return out

def _clean_proceso(s: pd.Series) -> pd.Series:
    """
    Normaliza proceso a 3 buckets:
    NATURAL, RAINBOW, TINT
    y todo lo demás -> OTHER (no afecta tu TOTAL porque sumas solo 3).
    FIX: NO usar np.where (convierte a ndarray y rompe .str)
    """
    s = s.astype(str).str.upper().str.strip()
    s = s.str.replace(r"\s+", " ", regex=True)

    # normalizaciones directas
    s = s.replace({
        "GARLAND NATURAL": "NATURAL",
        "GLITTER": "NATURAL",
        "RAINBOW 2": "RAINBOW",
        "ARCOIRIS": "RAINBOW",
        "BLANCO": "NATURAL",
        "TINTURADO": "TINT",
        "TINT": "TINT",
    })

    # reglas por contains (manteniendo tipo Series)
    s = s.mask(s.str.contains("TINT", na=False), "TINT")
    s = s.mask(s.str.contains("RAINBOW|ARCO", na=False), "RAINBOW")
    s = s.mask(s.str.contains("NATURAL|BLANCO", na=False), "NATURAL")

    # bucket final
    s = s.where(s.isin(["NATURAL", "RAINBOW", "TINT"]), "OTHER")

    return s

```

```

def _clean_tipo_tallo(s: pd.Series) -> pd.Series:
    return (
        s.astype(str)
        .str.upper()
        .str.replace("GR", "", regex=False)
        .str.strip()
    )

def _looks_like_date_colname(x: str) -> bool:
    s = str(x).strip()
    if re.fullmatch(r"\d{4}-\d{1,2}-\d{1,2}", s):
        return True
    if re.fullmatch(r"\d{4}-\d{1,2}-\d{1,2}\s+\d{2}:\d{2}:\d{2}", s):
        return True
    if re.fullmatch(r"\d{1,2}/\d{1,2}/\d{4}", s):
        return True
    if re.fullmatch(r"\d{1,2}/\d{1,2}/\d{4}\s+\d{2}:\d{2}:\d{2}", s):
        return True
    return False

def _is_excel_zero_date(col) -> bool:
    s = str(col).strip()
    if s == "1900-01-01 00:00:00":
        return True
    try:
        dt = pd.to_datetime(col, errors="coerce")
        return pd.notna(dt) and dt.normalize() == pd.Timestamp("1900-01-01")
    except Exception:
        return False

# =====
# IO: read ventas parquet (bronze wide con headers internos)
# =====
def read_ventas_bronze_parquet(parquet_path: Path) -> pd.DataFrame:
    if not parquet_path.exists():
        raise FileNotFoundError(f"No existe Bronze ventas parquet: {parquet_path}")

    raw = pd.read_parquet(parquet_path)

    meta_cols = [c for c in ["bronze_source", "bronze_extracted_at"] if c in raw.columns]
    df = raw.drop(columns=meta_cols, errors="ignore").copy()

    def _row_contains(row: pd.Series, token: str) -> bool:
        vals = row.astype(str).str.strip().str.upper()
        return (vals == token.upper()).any()

    header_idx = None
    scan_n = min(len(df), 50)
    for i in range(scan_n):
        r = df.iloc[i]
        if _row_contains(r, "Cliente") and _row_contains(r, "FINCA"):
            header_idx = i
            break

    if header_idx is None:
        raise ValueError(
            f"No pude detectar fila de encabezado en {parquet_path.name}. "
            "Esperaba encontrar 'Cliente' y 'FINCA' en las primeras 50 filas."
        )

    headers = df.iloc[header_idx].copy()

    new_cols = []
    for h in headers.tolist():
        if pd.isna(h):
            new_cols.append(None)
            continue

```

```

    if isinstance(h, (pd.Timestamp, datetime, np.datetime64)):
        new_cols.append(pd.to_datetime(h))
    else:
        s = str(h).replace("\n", " ").replace("\r", " ")
        s = re.sub(r"\s+", " ", s).strip()
        new_cols.append(s)

out = df.iloc[header_idx + 1:].copy()
out.columns = new_cols

out = out.loc[:, [c for c in out.columns if c is not None and str(c).strip() != ""]]
out = out.dropna(axis=1, how="all")

clean_cols = []
for c in out.columns:
    if isinstance(c, (pd.Timestamp, datetime, np.datetime64)):
        clean_cols.append(pd.to_datetime(c))
    else:
        s = str(c).replace("\n", " ").replace("\r", " ")
        s = re.sub(r"\s+", " ", s).strip()
        clean_cols.append(s)
out.columns = clean_cols

return out

# =====
# Transform: wide -> long + parse fecha robusto (FIX pyarrow string assignment)
# =====
def unpivot_ventas(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df.columns = [str(c).strip() for c in df.columns]

    c_finca = _find_col(df, ["FINCA", "Finca"])
    c_var = _find_col(df, ["VAR", "Var", "Variedad"])
    c_cliente = _find_col(df, ["Cliente", "CLIENTE", "Customer"])
    c_proceso = _find_col(df, ["Proceso", "PROCESO"])
    c_tipo_tallo = _find_col(df, ["Tipo Tallo", "TIPO TALLO", "TipoTallo", "TIPOTALLO", "Tipo_Tallo"])
    c_grado = _find_col(df, ["Grado", "GRADO"])
    c_stbx = _find_col(df, ["ST/BX", "STBX", "ST_BX"])
    c_bunca = _find_col(df, ["BUN/CAJA", "BUN CAJA", "BUNCAJA", "BUN_CAJA"])
    c_peso = _find_col(df, ["Peso", "Peso Final", "PESO", "PESO FINAL"])

    if c_finca:
        df = df[df[c_finca].astype(str).str.strip().str.upper().eq("MALIMA")].copy()

    if c_var:
        vv = df[c_var].astype(str).str.upper().str.strip()
        df = df[vv.isin(["XL", "XL 2"])].copy()

    if c_proceso:
        # filtra NATURAL PRESERVADA robusto
        proc = df[c_proceso].astype(str).str.upper().str.strip()
        df = df[~proc.str.contains("NATURAL PRESERVADA", na=False)].copy()

    if c_cliente:
        df = df[~df[c_cliente].astype(str).str.upper().isin(["ZINVENTARIOS", "ZMONJAS"])].copy()

    if c_proceso:
        df[c_proceso] = _clean_proceso(df[c_proceso])
    if c_tipo_tallo:
        df[c_tipo_tallo] = _clean_tipo_tallo(df[c_tipo_tallo])

    extra_fixed = []
    for name in [
        "Cliente", "Vendedor", "Cl2", "Nota", "Orden", "Producto", "Largo",
        "Perdida Peso", "Proceso 2", "Color", "Mercado", "Caja", "TALLOS SKU",
        "presentacion", "sku", "VAR", "FINCA", "ROT"
    ]:
        col = _find_col(df, [name])
        if col:
            extra_fixed.append(col)

```

```

fixed_real = [c for c in [c_tipo_tallo, c_grado, c_proceso, c_stbx, c_bunca, c_peso] if c] + extra_fixed
seen = set()
fixed_real = [c for c in fixed_real if not (c in seen or seen.add(c))]

value_cols = []
for c in df.columns:
    if c in fixed_real:
        continue
    if _looks_like_date_colname(c):
        value_cols.append(c)

if len(value_cols) == 0:
    for c in df.columns:
        if c in fixed_real:
            continue
        if pd.api.types.is_datetime64_any_dtype(df[c]):
            value_cols.append(c)

if len(value_cols) == 0:
    raise ValueError(
        "No se detectaron columnas fecha para unpivot.\n"
        f"Columnas disponibles: {list(df.columns)[:40]} ...\n"
        "Solución: confirma si Bronze está wide (fechas como columnas) o long (columna 'Fecha')."
    )

# excluir columna basura 1900-01-01
value_cols = [c for c in value_cols if not _is_excel_zero_date(c)]

out = df.melt(id_vars=fixed_real, value_vars=value_cols, var_name="Fecha", value_name="Valor")

# --- FIX CLAVE: no asignar datetimes sobre columna string[pyarrow] con .loc ---
fecha_str = out["Fecha"].astype(str).str.strip()

is_iso = fecha_str.str.match(r"^\d{4}-\d{1,2}-\d{1,2}$")
is_iso_dt = fecha_str.str.match(r"^\d{4}-\d{1,2}-\d{1,2}\s+\d{2}:\d{2}:\d{2}$")
is_slash = fecha_str.str.match(r"^\d{1,2}/\d{1,2}/\d{4}$")
is_slash_dt = fecha_str.str.match(r"^\d{1,2}/\d{1,2}/\d{4}\s+\d{2}:\d{2}:\d{2}$")

fecha_dt = pd.Series(pd.NaT, index=out.index, dtype="datetime64[ns]")

parsed_iso = pd.to_datetime(fecha_str.where(is_iso), format="%Y-%m-%d", errors="coerce")
parsed_iso_dt = pd.to_datetime(fecha_str.where(is_iso_dt), format="%Y-%m-%d %H:%M:%S", errors="coerce")

# Mantengo tu contrato: slash = mm/dd/yyyy
parsed_slash = pd.to_datetime(fecha_str.where(is_slash), format="%m/%d/%Y", errors="coerce")
parsed_slash_dt = pd.to_datetime(fecha_str.where(is_slash_dt), format="%m/%d/%Y %H:%M:%S", errors="coerce")

)

fecha_dt = fecha_dt.fillna(parsed_iso)
fecha_dt = fecha_dt.fillna(parsed_iso_dt)
fecha_dt = fecha_dt.fillna(parsed_slash)
fecha_dt = fecha_dt.fillna(parsed_slash_dt)

still = fecha_dt.isna()
if still.any():
    fecha_dt = fecha_dt.fillna(pd.to_datetime(fecha_str.where(still), errors="coerce"))

out["Fecha"] = fecha_dt.dt.normalize()

out["Valor"] = pd.to_numeric(out["Valor"], errors="coerce").fillna(0.0)
out = out[out["Fecha"].notna()].copy()

rename_map = {}
if c_tipo_tallo:
    rename_map[c_tipo_tallo] = "Tipo Tallo"
if c_proceso:
    rename_map[c_proceso] = "Proceso"
out = out.rename(columns=rename_map)

if "Tipo Tallo" not in out.columns:
    out["Tipo Tallo"] = np.nan

```

```

    if "Proceso" not in out.columns:
        out["Proceso"] = np.nan

    return out

# =====
# Build mix semana
# =====
def build_mix_semana(df_long: pd.DataFrame) -> pd.DataFrame:
    df = df_long.copy()

    df["Fecha_Clasificacion"] = df["Fecha"] - pd.Timedelta(days=2)
    df["Semana_Ventas"] = semana_ventas(df["Fecha_Clasificacion"])

    tipo_series = df["Tipo Tallo"] if "Tipo Tallo" in df.columns else pd.Series([np.nan] * len(df), index=df.index)
    tipo = tipo_series.astype(str).str.upper().str.strip()
    es_otro = tipo.isin(["BQT", "PET"])

    proc_series = df["Proceso"] if "Proceso" in df.columns else pd.Series([np.nan] * len(df), index=df.index)
    df["Proceso"] = _clean_proceso(proc_series)

    netas = (
        df[~es_otro]
        .groupby(["Semana_Ventas", "Proceso"], dropna=False)["Valor"]
        .sum()
        .reset_index()
        .rename(columns={"Valor": "cajas_netas"})
    )

    piv = netas.pivot_table(
        index="Semana_Ventas",
        columns="Proceso",
        values="cajas_netas",
        aggfunc="sum",
        fill_value=0.0,
    ).reset_index()

    # asegurar columnas estándar
    for c in ["NATURAL", "RAINBOW", "TINT"]:
        if c not in piv.columns:
            piv[c] = 0.0

    piv["TOTAL"] = piv["NATURAL"] + piv["RAINBOW"] + piv["TINT"]

    # weights sin np.where (más limpio)
    piv["W_Blanco"] = (piv["NATURAL"] / piv["TOTAL"]).where(piv["TOTAL"] > 0)
    piv["W_Arcoiris"] = (piv["RAINBOW"] / piv["TOTAL"]).where(piv["TOTAL"] > 0)
    piv["W_Tinturado"] = (piv["TINT"] / piv["TOTAL"]).where(piv["TOTAL"] > 0)

    out = piv[["Semana_Ventas", "W_Blanco", "W_Arcoiris", "W_Tinturado", "TOTAL"]].copy()
    out["created_at"] = datetime.now().isoformat(timespec="seconds")
    return out

# =====
# Main
# =====
def main() -> None:
    cfg = load_settings()

    if "paths" not in cfg or "bronze" not in cfg["paths"] or "silver" not in cfg["paths"]:
        raise ValueError("settings.yaml debe incluir paths.bronze y paths.silver")

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    ventas_cfg = cfg.get("ventas", {})
    p25_name = ventas_cfg.get("ventas_2025_raw_parquet", "ventas_2025_raw.parquet")
    p26_name = ventas_cfg.get("ventas_2026_raw_parquet", "ventas_2026_raw.parquet")

```

```

path_2025 = bronze_dir / p25_name
path_2026 = bronze_dir / p26_name

frames: list[pd.DataFrame] = []

if path_2026.exists():
    df26 = read_ventas_bronze_parquet(path_2026)
    frames.append(unpivot_ventas(df26))

if path_2025.exists():
    df25 = read_ventas_bronze_parquet(path_2025)
    frames.append(unpivot_ventas(df25))

if not frames:
    raise FileNotFoundError(
        "No se encontraron ventas raw en Bronze. Esperaba:\n"
        f" - {path_2026}\n"
        f" - {path_2025}\n"
        "Ajusta config/settings.yaml (paths.bronze y ventas.*_raw_parquet) o verifica nombres."
    )

df_long = pd.concat(frames, ignore_index=True)
mix = build_mix_semana(df_long)

# seed DEFAULT por mediana (si todo es NaN, fallback 1/3,1/3,1/3)
wb = float(np.nanmedian(mix["W_Blanco"].values)) if len(mix) else float("nan")
wa = float(np.nanmedian(mix["W_Arcoiris"].values)) if len(mix) else float("nan")
wt = float(np.nanmedian(mix["W_Tinturado"].values)) if len(mix) else float("nan")

seed = {"W_Blanco": wb, "W_Arcoiris": wa, "W_Tinturado": wt}
if not np.isfinite(seed["W_Blanco"]):
    seed["W_Blanco"] = 0.0
if not np.isfinite(seed["W_Arcoiris"]):
    seed["W_Arcoiris"] = 0.0
if not np.isfinite(seed["W_Tinturado"]):
    seed["W_Tinturado"] = 0.0

s = seed["W_Blanco"] + seed["W_Arcoiris"] + seed["W_Tinturado"]
if s <= 0:
    seed = {"W_Blanco": 1 / 3, "W_Arcoiris": 1 / 3, "W_Tinturado": 1 / 3}
else:
    seed = {k: v / s for k, v in seed.items()}

seed_row = pd.DataFrame([
    "Semana_Ventas": "DEFAULT",
    "W_Blanco": seed["W_Blanco"],
    "W_Arcoiris": seed["W_Arcoiris"],
    "W_Tinturado": seed["W_Tinturado"],
    "TOTAL": np.nan,
    "created_at": datetime.now().isoformat(timespec="seconds"),
])

mix2 = pd.concat([mix, seed_row], ignore_index=True)

out_path = silver_dir / "dim_mix_proceso_semana.parquet"
write_parquet(mix2, out_path)

print(f"OK: dim_mix_proceso_semana={len(mix2):,} filas -> {out_path}")
print("Seed DEFAULT (mediana histórica normalizada):", seed)

if __name__ == "__main__":
    main()

```

```

-----
[150/164] FILE: \src\silver\build_dim_peso_tallo_baseline.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path

```

```

from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def main() -> None:
    cfg = load_settings()

    silver_dir = Path(cfg["paths"]["silver"])
    in_path = silver_dir / "fact_peso_tallo_real_grado_dia.parquet"
    if not in_path.exists():
        raise FileNotFoundError(f"No existe: {in_path}. Ejecuta build_fact_peso_tallo_real_grado_dia primero."
)

    fact = read_parquet(in_path).copy()

    needed = {"fecha", "variedad", "grado", "peso_tallo_real_g"}
    missing = needed - set(fact.columns)
    if missing:
        raise ValueError(f"Faltan columnas en fact_peso_tallo_real_grado_dia: {missing}")

    fact["variedad"] = fact["variedad"].astype(str).str.strip().str.upper()
    fact["grado"] = pd.to_numeric(fact["grado"], errors="coerce").astype("Int64")
    fact["peso_tallo_real_g"] = pd.to_numeric(fact["peso_tallo_real_g"], errors="coerce")

    fact = fact[fact["grado"].notna() & fact["peso_tallo_real_g"].notna()].copy()
    fact = fact[(fact["peso_tallo_real_g"] > 1) & (fact["peso_tallo_real_g"] < 500)].copy()

    if fact.empty:
        raise ValueError(
            "fact_peso_tallo_real_grado_dia quedó vacío tras filtros (grado/peso_tallo_real_g). "
            "Revisa el fact upstream y/o los rangos de peso."
        )

    # Baseline robusto por variedad+grado
    dim = (
        fact.groupby(["variedad", "grado"], dropna=False)
        .agg(
            n_dias=("fecha", "nunique"),
            peso_tallo_mediana_g=("peso_tallo_real_g", "median"),
            peso_tallo_p25_g=("peso_tallo_real_g", lambda s: float(np.nanpercentile(s, 25))),
            peso_tallo_p75_g=("peso_tallo_real_g", lambda s: float(np.nanpercentile(s, 75))),
        )
        .reset_index()
    )

    # Guardrail: opcional mínimo de días (default 7)
    min_days = int(cfg.get("pipeline", {}).get("peso_tallo_min_days", 7))
    if int(dim["n_dias"].max()) < min_days:
        raise ValueError(
            f"Baseline peso_tallo: cobertura insuficiente. "
            f"max(n_dias)={int(dim['n_dias'].max())} < min_days={min_days}. "
            "Ejecuta/valida upstream para tener más historia."
        )

    dim["created_at"] = datetime.now().isoformat(timespec="seconds")

    out_path = silver_dir / "dim_peso_tallo_baseline.parquet"
    write_parquet(dim, out_path)

    print(f"OK: dim_peso_tallo_baseline={len(dim)} filas -> {out_path}")
    print("peso_tallo_mediana_g describe:\n", dim["peso_tallo_mediana_g"].describe().to_string())

```



```

if __name__ == "__main__":
    main()

-----
[151/164] FILE: \src\silver\build_dim_peso_tallo_promedio_dia.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def main() -> None:
    cfg = load_settings()
    silver_dir = Path(cfg["paths"]["silver"])

    in_path = silver_dir / "fact_peso_tallo_real_grado_dia.parquet"
    if not in_path.exists():
        raise FileNotFoundError(f"No existe: {in_path}. Primero construye fact_peso_tallo_real_grado_dia.")

    df = read_parquet(in_path).copy()

    needed = {"fecha", "tallos_real", "peso_tallo_real_g"}
    missing = needed - set(df.columns)
    if missing:
        raise ValueError(f"fact_peso_tallo_real_grado_dia no tiene columnas requeridas: {missing}")

    df["fecha"] = pd.to_datetime(df["fecha"], errors="coerce").dt.normalize()
    df["tallos_real"] = pd.to_numeric(df["tallos_real"], errors="coerce").fillna(0.0)
    df["peso_tallo_real_g"] = pd.to_numeric(df["peso_tallo_real_g"], errors="coerce")

    df = df[df["fecha"].notna()].copy()
    df = df[(df["tallos_real"] > 0) & df["peso_tallo_real_g"].notna()].copy()

    # Guardrail
    if df.empty:
        raise ValueError("No hay datos válidos para calcular peso_tallo_promedio_dia (df vacío tras filtros).")
)

# Promedio ponderado diario: sum(w*x)/sum(w)
df["wx"] = df["tallos_real"] * df["peso_tallo_real_g"]

daily = (df.groupby("fecha", dropna=False)
        .agg(
            tallos_dia=("tallos_real", "sum"),
            wx_dia=("wx", "sum"),
        )
        .reset_index())

daily["peso_tallo_prom_g"] = np.where(
    daily["tallos_dia"] > 0,
    daily["wx_dia"] / daily["tallos_dia"],
    np.nan
)

daily = daily[["fecha", "peso_tallo_prom_g", "tallos_dia"]].copy()
daily["created_at"] = datetime.now().isoformat(timespec="seconds")

out_path = silver_dir / "dim_peso_tallo_promedio_dia.parquet"
write_parquet(daily, out_path)

```

```

print(f"OK: dim_peso_tallo_promedio_dia={len(daily)} -> {out_path}")
print(daily.head(10).to_string(index=False))

if __name__ == "__main__":
    main()

-----
[152/164] FILE: \src\silver\build_dim_variedad_canon.py
-----
from __future__ import annotations

from pathlib import Path
import pandas as pd

from common.io import write_parquet

MAP = {
    "XLENCE": "XL",
    "XL": "XL",
    "CLOUD": "CLO",
    "CLO": "CLO",
}

def main() -> None:
    created_at = pd.Timestamp.utcnow()

    rows = []
    for k, v in MAP.items():
        rows.append({"variedad_raw": k, "variedad_canon": v})

    df = pd.DataFrame(rows).drop_duplicates().sort_values(["variedad_canon", "variedad_raw"])
    df["created_at"] = created_at

    write_parquet(df, Path("data/silver/dim_variedad_canon.parquet"))
    print(f"OK -> data/silver/dim_variedad_canon.parquet | rows={len(df):,}")

if __name__ == "__main__":
    main()

-----
[153/164] FILE: \src\silver\build_fact_capacidad_proceso_hist.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import write_parquet

# =====
# Paths / Settings
# =====
ROOT = Path(__file__).resolve().parents[2] # ../src/silver -> repo root
SETTINGS_PATH = ROOT / "config" / "settings.yaml"

def load_settings() -> dict:
    p = SETTINGS_PATH if SETTINGS_PATH.exists() else Path("config/settings.yaml")
    if not p.exists():
        raise FileNotFoundError(f"No existe settings.yaml en {p} (ni en config/settings.yaml).")
    with open(p, "r", encoding="utf-8") as f:
        return yaml.safe_load(f) or {}

```

```

# =====
# Helpers
# =====
def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _canon_str(s: pd.Series) -> pd.Series:
    return s.astype(str).str.strip().str.upper()

def _to_float(s: pd.Series) -> pd.Series:
    # robusto: acepta floats/ints, y también strings con coma decimal
    if s is None:
        return pd.Series(dtype="float64")
    ss = s.copy()
    if ss.dtype != object:
        return pd.to_numeric(ss, errors="coerce")
    return pd.to_numeric(ss.astype(str).str.replace(",", ".", regex=False), errors="coerce")

def _pick_col(df: pd.DataFrame, candidates: list[str], required: bool = True) -> str | None:
    """
    Devuelve el nombre real de la columna existente en df probando candidatos (case-insensitive).
    """
    cols = list(df.columns)
    cols_l = {str(c).strip().lower(): c for c in cols}
    for cand in candidates:
        ckey = str(cand).strip().lower()
        if ckey in cols_l:
            return cols_l[ckey]
    if required:
        raise ValueError(
            f"No se encontró columna. Candidatos={candidates}. "
            f"Disponibles={cols}"
        )
    return None

def _norm_unidad(u: pd.Series) -> pd.Series:
    """
    Normaliza unidad a etiquetas estándar. Para este fact queremos KILOS.
    """
    x = _canon_str(u).replace({"NAN": "", "NONE": ""})
    return x.replace(
        {
            "KG": "KILOS",
            "KGS": "KILOS",
            "KILO": "KILOS",
            "KILOGRAMO": "KILOS",
            "KILOGRAMOS": "KILOS",
        }
    )

def map_proceso(codigo_actividad: pd.Series) -> pd.Series:
    c = _canon_str(codigo_actividad)
    return np.select(
        [
            c.eq("CBX"),
            c.eq("CXLTA1"),
            c.eq("CXLTA1RH"),
            c.isin(["PSMC", "CXLTA", "CXLTA1RH"]),
        ],
        [
            "BLANCO",
            "TINTURADO",
            "ARCOIRIS",
            "OTROS_RESERVADOS",
        ],
    )

```

```

        default="OTROS_RESERVADOS",
    )

def _nanpercentile_safe(s: pd.Series, q: float) -> float:
    x = pd.to_numeric(s, errors="coerce").astype(float)
    x = x[np.isfinite(x)]
    if len(x) == 0:
        return float("nan")
    return float(np.nanpercentile(x, q))

# =====
# Main
# =====
def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    created_at = datetime.now().isoformat(timespec="seconds")

    # -----
    # Inputs Bronze
    # -----
    ghu_name = cfg.get("ghu", {}).get("ghu_maestro_horas_file", "ghu_maestro_horas.parquet")
    personal_name = cfg.get("ghu", {}).get("personal_file", "personal_raw.parquet")

    ghu_path = bronze_dir / ghu_name
    per_path = bronze_dir / personal_name

    if not ghu_path.exists():
        raise FileNotFoundError(f"No existe Bronze: {ghu_path}")
    if not per_path.exists():
        raise FileNotFoundError(f"No existe Bronze: {per_path}")

    mh = pd.read_parquet(ghu_path)
    per = pd.read_parquet(per_path)

    mh.columns = [str(c).strip() for c in mh.columns]
    per.columns = [str(c).strip() for c in per.columns]

    # -----
    # Resolver columnas (robusto a variaciones)
    # -----
    c_fecha = _pick_col(mh, ["fecha", "Fecha"])
    c_cod_per = _pick_col(mh, ["codigo_personal", "Codigo_Personal", "Código_Personal"])
    c_cod_act = _pick_col(mh, ["codigo_actividad", "Codigo_Actividad", "Código_Actividad"])
    c_unid = _pick_col(mh, ["unidad_medida", "Unidad_Medida", "unidad", "unidad_med"])
    c_hpres = _pick_col(mh, ["horas_presenciales", "Horas_Presenciales", "horas_trabajadas", "Horas_Trabajadas"])

    c_uniprod = _pick_col(mh, ["unidades_producidas", "Unidades_Producidas", "unidades", "unid_prod"])
    c_hacum = _pick_col(mh, ["horas_acumula", "Horas_Acumula", "horas_acumuladas"], required=False)

    # Personal
    p_cod_per = _pick_col(per, ["codigo_personal", "Codigo_Personal", "Código_Personal"])
    p_activo = _pick_col(per, ["Activo_o_Inactivo", "activo_inactivo", "estado", "Estado"], required=False)
    if p_activo is None:
        per["activo_inactivo"] = np.nan
        p_activo = "activo_inactivo"

    # -----
    # Normalización mínima (Silver staging)
    # -----
    df = mh.copy()
    df["fecha"] = _norm_date(df[c_fecha])
    df["codigo_personal"] = df[c_cod_per].astype(str).str.strip()
    df["codigo_actividad"] = _canon_str(df[c_cod_act])
    df["unidad_medida"] = _norm_unidad(df[c_unid])

```

```

df["horas_presenciales"] = _to_float(df[c_hpres])
df["unidades_producidas"] = _to_float(df[c_uniprod])

if c_hacum is not None and c_hacum in df.columns:
    df["horas_acumula"] = _to_float(df[c_hacum])
else:
    df["horas_acumula"] = np.nan

# -----
# Join con Personal (LEFT) - dedupe para evitar explosión
# -----
per2 = per[[p_cod_per, p_activo]].copy()
per2.columns = ["codigo_personal", "activo_inactivo"]
per2["codigo_personal"] = per2["codigo_personal"].astype(str).str.strip()
per2 = per2.drop_duplicates(subset=["codigo_personal"], keep="last")

df = df.merge(per2, on="codigo_personal", how="left")

# -----
# Proceso + filtros del fact
# -----
df["proceso"] = map_proceso(df["codigo_actividad"])

codigos_interes = {"CBX", "CXLTA1", "CXLTA2", "CXLTA3", "PSMC", "CXLTA", "CXLTA4"}
df = df[df["codigo_actividad"].isin(codigos_interes)].copy()

# Esta etapa: SOLO KILOS
df = df[df["unidad_medida"].eq("KILOS")].copy()

# Validaciones básicas
df = df[df["fecha"].notna()].copy()
df = df[df["codigo_personal"].astype(str).str.len() > 0].copy()
df = df[df["horas_presenciales"].notna() & (df["horas_presenciales"] > 0)].copy()
df = df[df["unidades_producidas"].notna() & (df["unidades_producidas"] > 0)].copy()

if df.empty:
    raise ValueError("No hay datos válidos para construir fact_capacidad_proceso_hist (df vacío tras filtros).")

# -----
# Agregación por día x persona x proceso
# -----
fact = (
    df.groupby(["fecha", "proceso", "codigo_personal"], dropna=False)
    .agg(
        activo_inactivo=("activo_inactivo", "last"),
        horas_presenciales_dia=("horas_presenciales", "sum"),
        kg_procesados_dia=("unidades_producidas", "sum"),
        horas_acumula=("horas_acumula", "max"),
    )
    .reset_index()
)

fact["kg_h_persona_dia"] = fact["kg_procesados_dia"] / fact["horas_presenciales_dia"]
fact["kg_h_persona_dia"] = pd.to_numeric(fact["kg_h_persona_dia"], errors="coerce")

# Outliers: cap por proceso usando percentiles (1% y 99%)
p1 = fact.groupby("proceso", dropna=False)["kg_h_persona_dia"].transform(lambda s: _nanpercentile_safe(s, 1))
p99 = fact.groupby("proceso", dropna=False)["kg_h_persona_dia"].transform(lambda s: _nanpercentile_safe(s, 99))

# si algún grupo quedó sin percentiles (nan), no capear
fact["kg_h_persona_dia"] = np.where(
    np.isfinite(p1) & np.isfinite(p99),
    fact["kg_h_persona_dia"].clip(lower=p1, upper=p99),
    fact["kg_h_persona_dia"],
)

fact["created_at"] = created_at

out_fact = silver_dir / "fact_capacidad_proceso_hist.parquet"

```

```

write_parquet(fact, out_fact)

# -----
# Baseline por proceso
# -----
base = (
    fact.groupby("proceso", dropna=False)
    .agg(
        kg_h_persona_mediana=("kg_h_persona_dia", "median"),
        kg_h_persona_p25=("kg_h_persona_dia", lambda s: _nanpercentile_safe(s, 25)),
        kg_h_persona_p75=("kg_h_persona_dia", lambda s: _nanpercentile_safe(s, 75)),
        n_registros=("kg_h_persona_dia", "size"),
    )
    .reset_index()
)
base["created_at"] = created_at

out_base = silver_dir / "dim_capacidad_baseline_proceso.parquet"
write_parquet(base, out_base)

print(f"OK: fact_capacidad_proceso_hist={len(fact):,} filas -> {out_fact}")
print(f"OK: dim_capacidad_baseline_proceso={len(base):,} filas -> {out_base}")
print("Baseline:\n", base.to_string(index=False))

if __name__ == "__main__":
    main()

-----
[154/164] FILE: \src\silver\build_fact_cosecha_real_grado_dia.py
-----
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def bloque_padre_from_bloque(b: pd.Series) -> pd.Series:
    """
    303B -> 303 ; 303 -> 303 ; limpia todo lo no numérico.
    """
    return (b.astype(str)
            .str.upper()
            .str.strip()
            .str.replace(r"^[^0-9]", "", regex=True))

def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    dist_cfg = cfg.get("dist_grado", {})
    fecha_min = pd.to_datetime(dist_cfg.get("fecha_min", "2024-01-01"))
    destino_excluir = dist_cfg.get("destino_excluir", []) or []

```

```

grados_validos = dist_cfg.get("grados_validos", []) or []
min_tallos_dia = int(dist_cfg.get("min_tallos_dia", 1))

variedades_validas = dist_cfg.get("variedades_validas", []) or []

# Input bronze
balanza_bronze_name = dist_cfg.get("balanza_cosecha_bronze_file", "balanza_cosecha_raw.parquet")
in_path = bronze_dir / balanza_bronze_name
if not in_path.exists():
    raise FileNotFoundError(
        f"No existe Bronze: {in_path}. Ejecuta src/bronze/build_balanza_cosecha_raw.py primero."
    )

df = pd.read_parquet(in_path)
df.columns = [str(c).strip() for c in df.columns]

needed = {"Fecha", "Variedad", "Bloque", "Grado", "Destino", "Tallos"}
missing = needed - set(df.columns)
if missing:
    raise ValueError(f"balanza_cosecha_raw no tiene columnas requeridas: {missing}")

# Tipos / limpieza (Silver)
df["Fecha"] = _norm_date(df["Fecha"])
df["Variedad"] = df["Variedad"].astype(str).str.strip().str.upper()
df["Bloque"] = df["Bloque"].astype(str).str.strip()
df["Destino"] = df["Destino"].astype(str).str.strip()
df["Grado"] = pd.to_numeric(df["Grado"], errors="coerce").astype("Int64")
df["Tallos"] = pd.to_numeric(df["Tallos"], errors="coerce").fillna(0.0)

# Filtros
df = df[df["Fecha"].notna()].copy()
df = df[df["Fecha"] >= fecha_min].copy()
df = df[df["Tallos"] > 0].copy()

if variedades_validas:
    vv = set(str(x).strip().upper() for x in variedades_validas)
    df = df[df["Variedad"].isin(vv)].copy()

if destino_excluir:
    df = df[~df["Destino"].isin(destino_excluir)].copy()

if grados_validos:
    gv = set(int(x) for x in grados_validos)
    df = df[df["Grado"].isin(gv)].copy()

df["bloque_padre"] = bloque_padre_from_bloque(df["Bloque"])
df = df[df["bloque_padre"].astype(str).str.len() > 0].copy()

# Agregación diaria por bloque_padre, variedad, grado
fact = (df.groupby(["Fecha", "bloque_padre", "Variedad", "Grado"], dropna=False)
        .agg(tallos_real=("Tallos", "sum"))
        .reset_index())

fact = fact.rename(columns={
    "Fecha": "fecha",
    "Variedad": "variedad",
    "Grado": "grado",
})

fact["tallos_real"] = fact["tallos_real"].astype(float)

# Filtro mínimo por día
fact = fact[fact["tallos_real"] >= float(min_tallos_dia)].copy()

fact["created_at"] = datetime.now().isoformat(timespec="seconds")

out_path = silver_dir / "fact_cosecha_real_grado_dia.parquet"
write_parquet(fact, out_path)

print(f"OK: fact_cosecha_real_grado_dia={len(fact)} filas -> {out_path}")
print("Rango fechas:", fact["fecha"].min(), "->", fact["fecha"].max())
print("Grados:", sorted(fact["grado"].dropna().unique().tolist())[:20])

```

```

if __name__ == "__main__":
    main()

-----
[155/164] FILE: \src\silver\build_fact_cosecha_uph_hora_clima.py
-----
# src/silver/build_fact_cosecha_uph_hora_clima.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import warnings
import numpy as np
import pandas as pd
import yaml

from common.io import write_parquet

# -----
# Config / helpers
# -----
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

def bloque_norm_from_raw(b: pd.Series) -> pd.Series:
    x = b.astype(str).str.strip()
    out = x.str.extract(r"^(\\d+)", expand=False)
    return pd.to_numeric(out, errors="coerce").astype("Int64")

def map_variedad(codigo_actividad: pd.Series) -> pd.Series:
    c = codigo_actividad.astype(str).str.upper().str.strip()
    xl = {"ZCS", "ZCSP", "ZXL", "ZVX", "ZPPX", "ZCX"} # XLENCE
    clo = {"ZMP", "ZVP", "ZPC", "ZCC"} # CLOUD
    out = np.where(
        c.isin(list(xl)),
        "XLENCE",
        np.where(c.isin(list(clo)), "CLOUD", None),
    )
    return pd.Series(out, index=c.index)

def map_station(area_trabajada: pd.Series) -> pd.Series:
    a = area_trabajada.astype(str).str.upper().str.strip()
    is_a4 = a.isin(["A-4", "A4", "SJP", "SAN JUAN"])
    return pd.Series(np.where(is_a4, "A4", "MAIN"), index=a.index)

def build_hours_grid() -> pd.DataFrame:
    rows = []
    for h in range(24):
        ini = pd.to_timedelta(h, unit="h")
        fin = (
            pd.to_timedelta(h + 1, unit="h")
            if h < 23
            else pd.to_timedelta(23, unit="h")
            + pd.to_timedelta(59, unit="m")
            + pd.to_timedelta(59, unit="s")
        )

```



```

    )
    rows.append({"hora_n": h, "td_ini": ini, "td_fin": fin})
return pd.DataFrame(rows)

def overlap_seconds(a_start, a_end, b_start, b_end) -> np.ndarray:
    start = np.maximum(a_start, b_start)
    end = np.minimum(a_end, b_end)
    sec = (end - start) / np.timedelta64(1, "s")
    sec = np.where(sec < 0, 0, sec)
    return sec.astype(float)

def load_kardex_from_weather_wide(weather_path: Path, fecha_min: pd.Timestamp) -> pd.DataFrame:
    if not weather_path.exists():
        raise FileNotFoundError(f"No existe {weather_path}")

    k = pd.read_parquet(weather_path)
    k.columns = [str(c).strip() for c in k.columns]

    # dt_hora
    if "dt_hora" not in k.columns:
        if "fecha" in k.columns:
            k = k.rename(columns={"fecha": "dt_hora"})
        else:
            raise ValueError("weather_hour_wide: falta dt_hora (o 'fecha' con hora).")

    # station
    if "station" not in k.columns:
        if "estacion" in k.columns:
            k = k.rename(columns={"estacion": "station"})
        else:
            raise ValueError("weather_hour_wide: falta station (MAIN/A4).")

    # Estado_Kardex
    if "Estado_Kardex" not in k.columns:
        if "estado_kardex" in k.columns:
            k = k.rename(columns={"estado_kardex": "Estado_Kardex"})
        else:
            raise ValueError("weather_hour_wide: falta Estado_Kardex.")

    # En_Lluvia
    if "En_Lluvia" not in k.columns:
        if "en_lluvia" in k.columns:
            k = k.rename(columns={"en_lluvia": "En_Lluvia"})
        else:
            raise ValueError("weather_hour_wide: falta En_Lluvia.")

    k["dt_hora"] = pd.to_datetime(k["dt_hora"], errors="coerce")
    k = k[k["dt_hora"].notna()].copy()
    k = k[k["dt_hora"] >= pd.to_datetime(fecha_min)].copy()

    k["station"] = k["station"].astype(str).str.upper().str.strip().replace({"A-4": "A4", "SJP": "A4"})
    k["fecha"] = k["dt_hora"].dt.normalize()
    k["hora_n"] = k["dt_hora"].dt.hour.astype(int)

    k["Estado_Kardex"] = k["Estado_Kardex"].astype(str).str.upper().str.strip().replace({"HÚMEDO": "HUMEDO"})
    k["En_Lluvia"] = _to_num(k["En_Lluvia"]).fillna(0).astype(int)

    return k[["station", "fecha", "hora_n", "Estado_Kardex", "En_Lluvia"]].drop_duplicates()

def _robust_median(s: pd.Series) -> float:
    s = pd.to_numeric(s, errors="coerce").dropna()
    if len(s) == 0:
        return np.nan
    return float(s.median())

def compute_factor_codigo_zcsp(
    df_hourly: pd.DataFrame,
    *,

```

```

min_n_codigo: int = 200,
clip_low: float = 0.60,
clip_high: float = 1.60,
min_horas_tramo: float = 0.10,
) -> pd.DataFrame:
    """
    Aprende factor por station/variedad/codigo_actividad para llevar UPH -> escala ZCSP
    en contexto base: SECO y En_Lluvia=0.
    factor = median(uph_ZCSP) / median(uph_codigo)
    """
    d = df_hourly.copy()

    base = d[
        (d["Estado_Kardex"].astype(str).str.upper().str.strip() == "SECO")
        & (pd.to_numeric(d["En_Lluvia"], errors="coerce").fillna(0).astype(int) == 0)
        & (pd.to_numeric(d["horas_tramo"], errors="coerce").fillna(0.0) >= float(min_horas_tramo))
    ].copy()

    if base.empty:
        raise ValueError("No hay data base (SECO sin lluvia) para calibrar factor_codigo_zcsp.")

    z = base[base["codigo_actividad"] == "ZCSP"].copy()
    ref = (
        z.groupby(["station", "variedad"], dropna=False)
        .agg(uph_zcsp_mediana=("uph_tramo", _robust_median), n_zcsp=("uph_tramo", "size"))
        .reset_index()
    )

    cod = (
        base.groupby(["station", "variedad", "codigo_actividad"], dropna=False)
        .agg(uph_cod_mediana=("uph_tramo", _robust_median), n_cod=("uph_tramo", "size"))
        .reset_index()
    )

    out = cod.merge(ref, on=["station", "variedad"], how="left")
    out["factor_codigo_zcsp"] = out["uph_zcsp_mediana"] / out["uph_cod_mediana"]
    out["factor_codigo_zcsp"] = out["factor_codigo_zcsp"].replace([np.inf, -np.inf], np.nan)

    low_n = out["n_cod"].fillna(0).astype(int) < int(min_n_codigo)
    out.loc[low_n, "factor_codigo_zcsp"] = np.nan

    out.loc[out["codigo_actividad"] == "ZCSP", "factor_codigo_zcsp"] = 1.0
    out["factor_codigo_zcsp"] = out["factor_codigo_zcsp"].fillna(1.0).astype(float)
    out["factor_codigo_zcsp"] = out["factor_codigo_zcsp"].clip(lower=float(clip_low), upper=float(clip_high))

    out["created_at"] = datetime.now().isoformat(timespec="seconds")

    cols = [
        "station", "variedad", "codigo_actividad",
        "uph_cod_mediana", "n_cod",
        "uph_zcsp_mediana", "n_zcsp",
        "factor_codigo_zcsp",
        "created_at",
    ]
    return out[cols].sort_values(["station", "variedad", "codigo_actividad"]).reset_index(drop=True)

def _require_cols(df: pd.DataFrame, required: list[str], df_name: str) -> None:
    missing = [c for c in required if c not in df.columns]
    if missing:
        raise ValueError(f"{df_name}: faltan columnas requeridas: {missing}. Columnas disponibles={list(df.columns)}")

# -----
# Main
# -----
def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])

```

```

silver_dir.mkdir(parents=True, exist_ok=True)

# seeds clima
seeds = cfg.get("uph_clima_seed", {})
f_seco = float(seeds.get("f_seco", 1.00))
f_humedo = float(seeds.get("f_humedo", 0.88))
f_mojado = float(seeds.get("f_mojado", 0.80))
f_lluvia = float(seeds.get("f_lluvia", 0.72))

# cosecha cfg
cosecha_cfg = cfg.get("uph_cosecha", {})
fecha_min = pd.to_datetime(cosecha_cfg.get("fecha_min", "2024-01-01"))

station_by_area_only = bool(cosecha_cfg.get("station_by_area_trabajada_only", True))
drop_cloud_in_a4 = bool(cosecha_cfg.get("drop_cloud_in_a4", True))

valid_main = set(s.upper() for s in cosecha_cfg.get("valid_areas_main", []) if str(s).strip())
valid_a4 = set(s.upper() for s in cosecha_cfg.get("valid_areas_a4", []) if str(s).strip())

# calibración factor código
cal_cfg = cosecha_cfg.get("cal_factor_codigo_zcsp", {})
min_n_codigo = int(cal_cfg.get("min_n_codigo", 200))
min_horas_tramo = float(cal_cfg.get("min_horas_tramo", 0.10))
clip_low = float(cal_cfg.get("clip_low", 0.60))
clip_high = float(cal_cfg.get("clip_high", 1.60))

# clima (silver)
weather_path = silver_dir / "weather_hour_wide.parquet"
kardex = load_kardex_from_weather_wide(weather_path, fecha_min)

# -----
# INPUT BRONZE: ghu_maestro_horas
# -----
ghu_fname = cfg.get("ghu", {}).get("ghu_maestro_horas_file", "ghu_maestro_horas.parquet")
ghu_path = bronze_dir / ghu_fname
if not ghu_path.exists():
    raise FileNotFoundError(
        f"No existe input Bronze: {ghu_path}\n"
        "Ejecuta src/bronze/build_ghu_maestro_horas.py primero "
        "o define ghu.ghu_maestro_horas_file en settings.yaml."
    )

ghu = pd.read_parquet(ghu_path)
ghu.columns = [str(c).strip() for c in ghu.columns]

_require_cols(
    ghu,
    [
        "fecha",
        "codigo_personal",
        "nombres",
        "actividad",
        "codigo_actividad",
        "tipo_actividad",
        "area_trabajada",
        "area_original",
        "horas_acumula",
        "bloque",
        "hora_ingreso",
        "hora_salida",
        "horas_presenciales",
        "unidades_producidas",
    ],
    "ghu_maestro_horas (bronze)",
)

# Filtros equivalentes a tu SQL
ghu["fecha"] = _norm_date(ghu["fecha"])
ghu = ghu[ghu["fecha"].notna()].copy()
ghu = ghu[ghu["fecha"] >= pd.to_datetime(fecha_min)].copy()

ghu["tipo_actividad"] = ghu["tipo_actividad"].astype(str).str.strip()

```

```

ghu = ghu[ghu["tipo_actividad"].eq("COS-1")].copy()
ghu = ghu[ghu["bloque"].notna()].copy()

# -----
# Preparación GHU
# -----
ghu["codigo_actividad"] = ghu["codigo_actividad"].astype(str).str.upper().str.strip()
ghu["bloque_norm"] = bloque_norm_from_raw(ghu["bloque"])
ghu = ghu[ghu["bloque_norm"].notna()].copy()

ghu["area_trabajada"] = ghu["area_trabajada"].astype(str).str.upper().str.strip()
if station_by_area_only:
    ghu = ghu[ghu["area_trabajada"].notna() & (ghu["area_trabajada"] != "")].copy()

ghu["station"] = map_station(ghu["area_trabajada"])

if valid_main:
    mask_main = (ghu["station"] == "MAIN") & (ghu["area_trabajada"].isin(valid_main))
else:
    mask_main = (ghu["station"] == "MAIN")

if valid_a4:
    mask_a4 = (ghu["station"] == "A4") & (ghu["area_trabajada"].isin(valid_a4))
else:
    mask_a4 = (ghu["station"] == "A4")

ghu = ghu[mask_main | mask_a4].copy()

# variedad
ghu["variedad"] = map_variedad(ghu["codigo_actividad"])
ghu = ghu[ghu["variedad"].notna()].copy()

if drop_cloud_in_a4:
    bad = (ghu["station"] == "A4") & (ghu["variedad"].astype(str).str.upper().str.contains("CLOUD"))
    n_bad = int(bad.sum())
    if n_bad > 0:
        warnings.warn(f"[DATA QUALITY] Eliminando {n_bad} registros CLOUD en A4 (no debería existir).")
        ghu = ghu[~bad].copy()

# horas de ingreso/salida (en horas decimales)
hi = _to_num(ghu["hora_ingreso"]).fillna(0.0)
hf = _to_num(ghu["hora_salida"]).fillna(0.0)

ghu["mins_ini"] = np.round(hi * 60.0).astype(int).clip(0, 24 * 60 - 1)
ghu["mins_fin"] = np.round(hf * 60.0).astype(int).clip(0, 24 * 60 - 1)

ghu["dt_ini"] = ghu["fecha"] + pd.to_timedelta(ghu["mins_ini"], unit="m")
ghu["dt_fin"] = ghu["fecha"] + pd.to_timedelta(ghu["mins_fin"], unit="m")

bad_time = ghu["dt_fin"] <= ghu["dt_ini"]
if bad_time.any():
    warnings.warn(f"Registros con hora_salida <= hora_ingreso: {int(bad_time.sum())} (se ajustan +1 minuto)")
    ghu.loc[bad_time, "dt_fin"] = ghu.loc[bad_time, "dt_ini"] + pd.to_timedelta(1, unit="m")

ghu["horas_presenciales"] = _to_num(ghu["horas_presenciales"]).fillna(0.0)
ghu["unidades_producidas"] = _to_num(ghu["unidades_producidas"]).fillna(0.0)
ghu = ghu[(ghu["horas_presenciales"] > 0) & (ghu["unidades_producidas"] > 0)].copy()

# rec_id
ghu["rec_id"] = (
    ghu.groupby(
        ["fecha", "codigo_personal", "bloque_norm", "mins_ini", "mins_fin", "horas_presenciales", "unidades_producidas"],
        dropna=False,
    )
    .cumcount()
    + 1
)

# -----
# Explosión por hora (e)

```

```

# -----
horas = build_hours_grid()
ghu["_key"] = 1
horas["_key"] = 1
e = ghu.merge(horas, on="_key", how="inner").drop(columns=["_key"])

e["tramo_ini"] = e["fecha"] + e["td_ini"]
e["tramo_fin"] = e["fecha"] + e["td_fin"]

e["seg_tramo"] = overlap_seconds(
    e["dt_ini"].values.astype("datetime64[ns]"),
    e["dt_fin"].values.astype("datetime64[ns]"),
    e["tramo_ini"].values.astype("datetime64[ns]"),
    e["tramo_fin"].values.astype("datetime64[ns]"),
)
e = e[e["seg_tramo"] > 0].copy()

keys = ["fecha", "codigo_personal", "bloque_norm", "rec_id"]
seg_total = e.groupby(keys, dropna=False)["seg_tramo"].sum().rename("seg_total").reset_index()
e = e.merge(seg_total, on=keys, how="left")
e["W"] = np.where(e["seg_total"] > 0, e["seg_tramo"] / e["seg_total"], 0.0)

# join clima
e["station"] = e["station"].astype(str).str.upper().str.strip().replace({"A-4": "A4", "SJP": "A4"})
e = e.merge(kardex, on=["station", "fecha", "hora_n"], how="left")

miss = int(e["Estado_Kardex"].isna().sum())
if miss > 0:
    warnings.warn(f"Faltan {miss} joins de clima (Estado_Kardex null). Se asume SECO y En_Lluvia=0.")
    e["Estado_Kardex"] = e["Estado_Kardex"].fillna("SECO")
    e["En_Lluvia"] = e["En_Lluvia"].fillna(0)

e["Estado_Kardex"] = e["Estado_Kardex"].astype(str).str.upper().str.strip().replace({"HÚMEDO": "HUMEDO"})
e["En_Lluvia"] = _to_num(e["En_Lluvia"]).fillna(0).astype(int)
e["estado_final"] = np.where(e["En_Lluvia"].eq(1), "LLUVIA", e["Estado_Kardex"])

# base UPH por tramo (SIN normalizar aún, solo para calibrar factor por código)
e["horas_tramo"] = e["seg_tramo"] / 3600.0
e["UP_total_reg_real"] = e["unidades_producidas"].astype(float)
e["UP_tramo_real"] = e["UP_total_reg_real"] * e["W"]
e["uph_tramo"] = np.where(e["horas_tramo"] > 0, e["UP_tramo_real"] / e["horas_tramo"], np.nan)

# -----
# 1) Calibrar factor_codigo_zcsp (y guardarlo como dim)
# -----
dim_factor = compute_factor_codigo_zcsp(
    e[["station", "variedad", "codigo_actividad",
        "Estado_Kardex", "En_Lluvia",
        "horas_tramo", "uph_tramo"]
    ].copy(),
    min_n_codigo=min_n_codigo,
    min_horas_tramo=min_horas_tramo,
    clip_low=clip_low,
    clip_high=clip_high,
)

dim_path = silver_dir / "dim_factor_codigo_zcsp_cosecha.parquet"
write_parquet(dim_factor, dim_path)

# merge factor al hourly
e = e.merge(
    dim_factor[["station", "variedad", "codigo_actividad", "factor_codigo_zcsp"]],
    on=["station", "variedad", "codigo_actividad"],
    how="left",
)

miss_fac = int(e["factor_codigo_zcsp"].isna().sum())
if miss_fac > 0:
    warnings.warn(f"[WARN] {miss_fac} filas sin factor_codigo_zcsp. Se imputan a 1.0.")
    e["factor_codigo_zcsp"] = e["factor_codigo_zcsp"].fillna(1.0)

```

```

# -----
# 2) Aplicar factor => UP_total_reg_equiv (ZCSP scale)
# -----
e["UP_total_reg_equiv"] = e["UP_total_reg_real"] * e["factor_codigo_zcsp"].astype(float)

# -----
# 3) Ajuste clima (sobre el equivalente)
# -----
state_factor = {"SECO": f_seco, "HUMEDO": f_humedo, "MOJADO": f_mojado, "LLUVIA": f_lluvia}
e["factor_clima"] = e["estado_final"].map(state_factor).fillna(f_seco).astype(float)

e["factor_total"] = e["factor_clima"]

e["WF"] = e["W"] * e["factor_total"]
sum_wf = e.groupby(keys, dropna=False)["WF"].sum().rename("sum_WF").reset_index()
e = e.merge(sum_wf, on=keys, how="left")

e["UP_total_reg"] = e["UP_total_reg_equiv"]
e["HP_total_reg"] = e["horas_presenciales"].astype(float)

e["UP_tramo_raw"] = e["UP_total_reg"] * e["W"]
e["UP_tramo_ajustada"] = np.where(e["sum_WF"] > 0, e["UP_total_reg"] * (e["WF"] / e["sum_WF"]), 0.0)

e["uph_raw_tramo_equiv"] = np.where(e["horas_tramo"] > 0, e["UP_tramo_raw"] / e["horas_tramo"], np.nan)
e["uph_ajustada_tramo_equiv"] = np.where(e["horas_tramo"] > 0, e["UP_tramo_ajustada"] / e["horas_tramo"],
np.nan)

e["dt_hora"] = e["fecha"] + pd.to_timedelta(e["hora_n"].astype(int), unit="h")

out = e[[
    "fecha", "hora_n", "dt_hora",
    "codigo_personal", "nombres",
    "area_trabajada", "area_original",
    "bloque_norm", "rec_id",
    "codigo_actividad", "actividad",
    "variedad", "station",

    "Estado_Kardex", "En_Lluvia", "estado_final",

    "seg_tramo", "seg_total", "horas_tramo",
    "UP_total_reg_real", "UP_total_reg_equiv",
    "HP_total_reg",

    "W",
    "factor_codigo_zcsp",
    "factor_clima", "factor_total",
    "WF", "sum_WF",

    "UP_tramo_raw", "UP_tramo_ajustada",
    "uph_raw_tramo_equiv", "uph_ajustada_tramo_equiv",
]]

out[["created_at"]] = datetime.now().isoformat(timespec="seconds")

out_path = silver_dir / "fact_cosecha_uph_hora_clima.parquet"
write_parquet(out, out_path)

print(f"OK: dim_factor_codigo_zcsp_cosecha={len(dim_factor)} filas -> {dim_path}")
print(f"OK: fact_cosecha_uph_hora_clima={len(out)} filas -> {out_path}")
print("Rango fechas:", out["fecha"].min(), "->", out["fecha"].max())
print("Stations:", out["station"].value_counts(dropna=False).to_dict())
print("Variedades:", out["variedad"].value_counts(dropna=False).to_dict())

if __name__ == "__main__":
    main()

```

```

-----
[156/164] FILE: \src\silver\build_fact_peso_tallo_real_grado_dia.py
-----
# src/silver/build_fact_peso_tallo_real_grado_dia.py

```

```

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def bloque_padre_from_bloque(b: pd.Series) -> pd.Series:
    return (
        b.astype(str)
        .str.upper()
        .str.strip()
        .str.replace(r"^[0-9]", "", regex=True)
    )

def _require_cols(df: pd.DataFrame, required: list[str], df_name: str) -> None:
    missing = [c for c in required if c not in df.columns]
    if missing:
        raise ValueError(
            f"{df_name}: faltan columnas requeridas: {missing}. "
            f"Columnas disponibles={list(df.columns)}"
        )

def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    # -----
    # Input BRONZE: balanza raw
    # -----
    balanza_cfg = cfg.get("balanza", {})
    balanza_file = balanza_cfg.get("balanza_file", "balanza_raw.parquet")
    in_path = bronze_dir / balanza_file

    if not in_path.exists():
        raise FileNotFoundError(
            f"No existe input Bronze de balanza: {in_path}\n"
            "Crea primero la capa Bronze de balanza (la que contiene peso_menos_vegetativo), "
            "o ajusta balanza.balanza_file en settings.yaml."
        )

    df = read_parquet(in_path).copy()
    df.columns = [str(c).strip() for c in df.columns]

    # Requeridas para este fact
    needed = ["Fecha", "Variedad", "Bloque", "Grado", "Tallos", "peso_menos_vegetativo"]
    _require_cols(df, needed, f"balanza bronze ({balanza_file})")

    # -----
    # Filtros config (igual que antes)
    # -----
    dist_cfg = cfg.get("dist_grado", {})
    fecha_min = pd.to_datetime(dist_cfg.get("fecha_min", "2024-01-01"))

```

```

variedades_validas = dist_cfg.get("variedades_validas", []) or []
vv = set(str(x).strip().upper() for x in variedades_validas) if variedades_validas else None

# -----
# Limpieza + tipos
# -----
df["Fecha"] = _norm_date(df["Fecha"])
df["Variedad"] = df["Variedad"].astype(str).str.strip().str.upper()
df["Bloque"] = df["Bloque"].astype(str).str.strip()
df["Grado"] = pd.to_numeric(df["Grado"], errors="coerce").astype("Int64")
df["Tallos"] = pd.to_numeric(df["Tallos"], errors="coerce").fillna(0.0)
df["peso_menos_vegetativo"] = pd.to_numeric(df["peso_menos_vegetativo"], errors="coerce").fillna(0.0)

# Filtros operativos (idénticos)
df = df[df["Fecha"].notna()].copy()
df = df[df["Fecha"] >= fecha_min].copy()
df = df[(df["Tallos"] > 0) & (df["peso_menos_vegetativo"] > 0)].copy()
df = df[df["Grado"].notna()].copy()

if vv is not None:
    df = df[df["Variedad"].isin(vv)].copy()

df["bloque_padre"] = bloque_padre_from_bloque(df["Bloque"])
df = df[df["bloque_padre"].astype(str).str.len() > 0].copy()

# -----
# Agregación diaria por bloque_padre, variedad, grado
# -----
fact = (
    df.groupby(["Fecha", "bloque_padre", "Variedad", "Grado"], dropna=False)
    .agg(
        tallos_real=("Tallos", "sum"),
        peso_real_kg=("peso_menos_vegetativo", "sum"),
    )
    .reset_index()
)

fact = fact.rename(
    columns={
        "Fecha": "fecha",
        "Variedad": "variedad",
        "Grado": "grado",
    }
)

fact["tallos_real"] = fact["tallos_real"].astype(float)
fact["peso_real_kg"] = fact["peso_real_kg"].astype(float)

# convertir a gramos
fact["peso_real_g"] = fact["peso_real_kg"] * 1000.0
fact["peso_tallo_real_g"] = fact["peso_real_g"] / fact["tallos_real"]

# eliminar kg para no mezclar unidades
fact = fact.drop(columns=["peso_real_kg"])

# sanity caps (evitar basura extrema)
fact = fact[(fact["peso_tallo_real_g"] > 1) & (fact["peso_tallo_real_g"] < 500)].copy()

fact["created_at"] = datetime.now().isoformat(timespec="seconds")

out_path = silver_dir / "fact_peso_tallo_real_grado_dia.parquet"
write_parquet(fact, out_path)

print(f"OK: fact_peso_tallo_real_grado_dia={len(fact)} filas -> {out_path}")
print("Rango fechas:", fact["fecha"].min(), "->", fact["fecha"].max())
print("peso_tallo_real_g describe:\n", fact["peso_tallo_real_g"].describe().to_string())

if __name__ == "__main__":
    main()

```

[157/164] FILE: \src\silver\build_hidratacion_real_from_balanza2.py

```
from __future__ import annotations
```

```
from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml
```

```
from common.io import write_parquet
```

```
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)
```

```
def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()
```

```
def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")
```

```
def _info(msg: str) -> None:
    print(f"[INFO] {msg}")
```

```
def _warn(msg: str) -> None:
    print(f"[WARN] {msg}")
```

```
def _require_cols(df: pd.DataFrame, required: list[str], df_name: str) -> None:
    missing = [c for c in required if c not in df.columns]
    if missing:
        raise ValueError(
            f"{df_name}: faltan columnas requeridas: {missing}. "
            f"Columnas disponibles: {list(df.columns)}"
        )
```

```
def _normalize_grado_to_int(s: pd.Series) -> pd.Series:
    # Soporta "60", 60, "60GR", "60 GR", "BQT", "PET" -> PET/BQT quedarán NaN
    x = s.astype(str).str.upper().str.strip()
    # extrae primer número
    n = x.str.extract(r"(\d+)", expand=False)
    return pd.to_numeric(n, errors="coerce").astype("Int64")
```

```
def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

    p2 = bronze_dir / "balanza_2_raw.parquet"
    plc = bronze_dir / "balanza_1c_raw.parquet"

    if not p2.exists():
        raise FileNotFoundError(f"No existe Bronze: {p2}")
    if not plc.exists():
        raise FileNotFoundError(f"No existe Bronze: {plc}. Ejecuta build_balanza_1c_raw.py")

    b2 = pd.read_parquet(p2)
    b1c = pd.read_parquet(plc)

    b2.columns = [str(c).strip() for c in b2.columns]
    b1c.columns = [str(c).strip() for c in b1c.columns]
```

```

_require_cols(
    b2,
    required=["fecha_entrega", "Lote", "Grado", "Destino", "Tallos", "peso_netto", "variedad", "tipo_pelado", "Origen"],
    df_name="balanza_2_raw",
)
_require_cols(
    b1c,
    required=["Fecha", "Destino", "Variedad", "Grado", "peso_balanza", "tallos"],
    df_name="balanza_1c_raw",
)

fecha_min = pd.to_datetime(cfg.get("hidratacion", {}).get("fecha_min", "2025-01-01"))

# =====
# 1) BALANZA 2 (post)
# =====
_info(f"Input b2: {len(b2)} filas")
b2["fecha_post"] = _norm_date(b2["fecha_entrega"])
b2["fecha_cosecha"] = _norm_date(b2["Lote"])

b2["Destino"] = b2["Destino"].astype(str).str.strip().str.upper()
b2["destino"] = np.where(b2["Destino"].isin(["GUERNALDA", "CLASIFICACION"]), "BLANCO", b2["Destino"])

b2["variedad"] = b2["variedad"].astype(str).str.strip().str.upper()
b2["tipo_pelado"] = b2["tipo_pelado"].astype(str).str.strip()
b2["Origen"] = b2["Origen"].astype(str).str.strip().str.upper()

if "producto" not in b2.columns:
    b2["producto"] = np.nan
prod = b2["producto"].astype("string")

b2["grado"] = _normalize_grado_to_int(b2["Grado"])
b2["tallos"] = _to_num(b2["Tallos"]).fillna(0.0)
b2["peso_post_raw"] = _to_num(b2["peso_netto"])

# Diagnóstico de dominios
_info("b2 dominios:")
_info(f" fecha_post: {b2['fecha_post'].min()} -> {b2['fecha_post'].max()}")
_info(f" fecha_cosecha(Lote): {b2['fecha_cosecha'].min()} -> {b2['fecha_cosecha'].max()}")
_info(f" variedad top: {b2['variedad'].value_counts(dropna=False).head(5).to_dict()}")
_info(f" tipo_pelado top: {b2['tipo_pelado'].value_counts(dropna=False).head(5).to_dict()}")
_info(f" Origen top: {b2['Origen'].value_counts(dropna=False).head(5).to_dict()}")

# Filtros negocio (como tu SQL)
m2 = (
    (b2["fecha_post"].notna())
    & (b2["fecha_cosecha"].notna())
    & (b2["fecha_post"] >= fecha_min)
    & (b2["grado"].notna())
    & (b2["variedad"].eq("GYXPXE"))
    & (b2["tipo_pelado"].eq("Sin Pelar"))
    & (b2["Origen"].eq("APERTURA"))
    & (prod.isna() | (prod.astype(str).str.upper().eq("GUERNALDA")))
    & (b2["tallos"] > 0)
    & (b2["peso_post_raw"].notna() & (b2["peso_post_raw"] > 0))
)

b2f = b2[m2].copy()
_info(f"b2 filtrado (SQL-like): {len(b2f)} filas")

if b2f.empty:
    _warn("b2f quedó vacío. Revisa filtros: variedad/tipo_pelado/Origen/producto.")
    # No seguimos para que no te dé todo NaN sin explicación.
    return

# --- Auto-unidad peso_netto ---
# Heurística: si el "peso por tallo" implícito queda demasiado alto, es probable que esté en gramos.
# (peso_post_raw / tallos) debería ser ~0.005 a 0.1 kg/tallo típico (5g a 100g)
ratio = (b2f["peso_post_raw"].astype(float) / b2f["tallos"].astype(float)).replace([np.inf, -np.inf], np.nan)

```

```

med_ratio = float(np.nanmedian(ratio.values))

# Si med_ratio > 1.0 => probablemente gramos (porque sería >1 kg por tallo, absurdo)
# Si med_ratio < 0.001 => también raro
if med_ratio > 1.0:
    _warn(f"peso_netto parece estar en GRAMOS (med peso/tallo={med_ratio:.3f}). Se convierte a kg (/1000).")
)

b2f["peso_post_kg"] = b2f["peso_post_raw"] / 1000.0
else:
    b2f["peso_post_kg"] = b2f["peso_post_raw"]

_info(f"peso_post_kg describe:\n{b2f['peso_post_kg'].describe().to_string()}")

b2g = (
    b2f.groupby(["fecha_post", "fecha_cosecha", "grado", "destino"], dropna=False)
        .agg(
            tallos=("tallos", "sum"),
            peso_post_kg=("peso_post_kg", "sum"),
        )
        .reset_index()
)
_info(f"b2 agregado: {len(b2g)} filas")

# =====
# 2) BALANZA 1C (base cosecha)
# =====
_info(f"Input blc: {len(blcf)} filas")

blc["fecha_cosecha"] = _norm_date(blcf["Fecha"])
blc["Destino"] = blc["Destino"].astype(str).str.strip().str.upper()
blc["Variedad"] = blc["Variedad"].astype(str).str.strip().str.upper()

blc["grado"] = _normalize_grado_to_int(blcf["Grado"])
blc["peso_balanza_raw"] = _to_num(blcf["peso_balanza"])
blc["tallos"] = _to_num(blcf["tallos"]).fillna(0.0)

_info("blc dominios:")
_info(f" fecha_cosecha: {blc['fecha_cosecha'].min()} -> {blc['fecha_cosecha'].max()}")
_info(f" Destino top: {blc['Destino'].value_counts(dropna=False).head(5).to_dict()}")
_info(f" Variedad top: {blc['Variedad'].value_counts(dropna=False).head(5).to_dict()}")

m1 = (
    (blc["fecha_cosecha"].notna())
    & (blc["fecha_cosecha"] >= fecha_min)
    & (blc["Destino"].eq("APERTURA"))
    & (blc["Variedad"].isin(["GYPXLE", "XLENCE"]))
    & (blc["grado"].notna())
    & (blc["peso_balanza_raw"].notna() & (blc["peso_balanza_raw"] > 0))
    & (blc["tallos"] > 0)
)
blcf = blc[m1].copy()
_info(f"blc filtrado (SQL-like): {len(blcf)} filas")

if blcf.empty:
    _warn("blcf quedó vacío. Revisa Destino='APERTURA' y Variedad in (GYPXLE, XLENCE).")
    return

blg = (
    blcf.groupby(["fecha_cosecha", "grado"], dropna=False)
        .agg(
            peso_balanza_sum=("peso_balanza_raw", "sum"),
            tallos_sum=("tallos", "sum"),
        )
        .reset_index()
)

# --- Auto-unidad peso_balanza ---
# Si peso_balanza es KG: (kg*1000)/tallos ~ 5-80 g/tallo típico.
# Si ya es GR: (gr)/tallos ~ 5-80 g/tallo típico.
peso_g_as_kg = (blg["peso_balanza_sum"] * 1000.0) / blg["tallos_sum"]
peso_g_as_g = (blg["peso_balanza_sum"]) / blg["tallos_sum"]

```

```

med_as_kg = float(np.nanmedian(peso_g_as_kg.values))
med_as_g = float(np.nanmedian(peso_g_as_g.values))

# Elegimos la opción cuyo valor mediano cae en rango plausible (2g a 200g)
def in_plausible(x: float) -> bool:
    return (x >= 2.0) and (x <= 200.0)

if in_plausible(med_as_g) and not in_plausible(med_as_kg):
    _warn(f"peso_balanza parece estar en GRAMOS (med g/tallo={med_as_g:.2f}). No se multiplica por 1000.")
    blg["peso_por_tallo_g"] = peso_g_as_g
elif in_plausible(med_as_kg) and not in_plausible(med_as_g):
    _info(f"peso_balanza parece estar en KILOS (med g/tallo={med_as_kg:.2f}). Se multiplica por 1000.")
    blg["peso_por_tallo_g"] = peso_g_as_kg
else:
    # Ambiguo: por defecto asumimos kg (tu supuesto original), pero lo avisamos
    _warn(
        f"Unidad de peso_balanza ambigua. med_as_kg={med_as_kg:.2f}, med_as_g={med_as_g:.2f}. "
        "Se asume KILOS por defecto. Si sale mal, fuerza en settings.yaml."
    )
    blg["peso_por_tallo_g"] = peso_g_as_kg

_info(f"peso_por_tallo_g describe:\n{blg['peso_por_tallo_g'].describe().to_string()}")
blg = blg[["fecha_cosecha", "grado", "peso_por_tallo_g"]].copy()

# =====
# 3) JOIN + HIDR
# =====
# Diagnóstico de cobertura antes del join: cuántas llaves coinciden
keys_b2 = b2g[["fecha_cosecha", "grado"]].drop_duplicates()
keys_b1 = blg[["fecha_cosecha", "grado"]].drop_duplicates()

inter = keys_b2.merge(keys_b1, on=["fecha_cosecha", "grado"], how="inner")
_info(f"Keys b2 (fecha_cosecha,grado): {len(keys_b2)}; Keys b1: {len(keys_b1)}; Intersección: {len(inter)}")
")

if inter.empty:
    _warn(
        "No hay intersección de llaves (fecha_cosecha, grado) entre balanza_2 y balanza_1c.\n"
        "Revisa: (1) Lote realmente es la fecha de cosecha, (2) grado viene comparable, (3) rangos de fecha."
    )
    # Te dejo muestras rápidas para inspección
    _info("Muestras b2 keys:\n" + keys_b2.head(10).to_string(index=False))
    _info("Muestras b1 keys:\n" + keys_b1.head(10).to_string(index=False))
    return

m = b2g.merge(blg, on=["fecha_cosecha", "grado"], how="left")
miss = int(m["peso_por_tallo_g"].isna().sum())
_info(f"Post-join: filas={len(m)}; sin match peso_por_tallo_g={miss}")

m = m[m["peso_por_tallo_g"].notna() & (m["peso_por_tallo_g"] > 0)].copy()
if m.empty:
    _warn("Después del join, todo quedó sin peso_por_tallo_g. Es mismatch total o nulls.")
    return

m["peso_base_kg"] = (m["peso_por_tallo_g"] * m["tallos"]) / 1000.0
m["hidr_pct"] = (m["peso_post_kg"] / m["peso_base_kg"]) - 1.0
m["dh_dias"] = (m["fecha_post"] - m["fecha_cosecha"]).dt.days.astype("Int64")

_info(f"hidr_pct BEFORE caps:\n{m['hidr_pct'].describe().to_string()}")
_info(f"dh_dias BEFORE caps:\n{m['dh_dias'].dropna().astype(int).describe().to_string()}")

# Caps seguridad
m = m[m["dh_dias"].between(0, 30)].copy()
m = m[m["hidr_pct"].between(-0.2, 3.0)].copy()

_info(f"Después caps: {len(m)} filas")

if m.empty:
    _warn(
        "Todo se eliminó por caps de dh_dias/hidr_pct. "
        "Probable problema de unidades (peso) o Lote incorrecto."
    )

```

```

    )
    return

m["peso_base_g"] = m["peso_base_kg"] * 1000.0
m["peso_post_g"] = m["peso_post_kg"] * 1000.0

fact = m[[
    "fecha_cosecha", "fecha_post", "dh_dias",
    "grado", "destino",
    "tallos",
    "peso_base_g", "peso_post_g",
    "hidr_pct",
]]

fact["created_at"] = datetime.now().isoformat(timespec="seconds")
out_fact = silver_dir / "fact_hidratacion_real_post_grado_destino.parquet"
write_parquet(fact, out_fact)

# =====
# 4) Dims (ponderado) sin apply
# =====
fact["_w"] = _to_num(fact["peso_base_g"]).fillna(0.0)
fact["_hydr_w"] = _to_num(fact["hidr_pct"]) * fact["_w"]

dim_dh = (
    fact.groupby(["dh_dias", "grado", "destino"], dropna=False)
        .agg(
            n=("hidr_pct", "size"),
            tallos=("tallos", "sum"),
            peso_base_g=("peso_base_g", "sum"),
            sum_hydr_w=("_hydr_w", "sum"),
            sum_w=("_w", "sum"),
        )
        .reset_index()
)

dim_dh["hidr_pct"] = np.where(dim_dh["sum_w"] > 0, dim_dh["sum_hydr_w"] / dim_dh["sum_w"], np.nan)
dim_dh = dim_dh.drop(columns=["sum_hydr_w", "sum_w"])
dim_dh["factor_hidr"] = 1.0 + dim_dh["hidr_pct"]
dim_dh["created_at"] = datetime.now().isoformat(timespec="seconds")

out_dim_dh = silver_dir / "dim_hidratacion_dh_grado_destino.parquet"
write_parquet(dim_dh, out_dim_dh)

dim_fecha = (
    fact.groupby(["fecha_post", "grado", "destino"], dropna=False)
        .agg(
            n=("hidr_pct", "size"),
            tallos=("tallos", "sum"),
            peso_base_g=("peso_base_g", "sum"),
            sum_hydr_w=("_hydr_w", "sum"),
            sum_w=("_w", "sum"),
        )
        .reset_index()
)

dim_fecha["hidr_pct"] = np.where(dim_fecha["sum_w"] > 0, dim_fecha["sum_hydr_w"] / dim_fecha["sum_w"], np.
nan)
dim_fecha = dim_fecha.drop(columns=["sum_hydr_w", "sum_w"])
dim_fecha["factor_hidr"] = 1.0 + dim_fecha["hidr_pct"]
dim_fecha["created_at"] = datetime.now().isoformat(timespec="seconds")

out_dim_fecha = silver_dir / "dim_hidratacion_fecha_post_grado_destino.parquet"
write_parquet(dim_fecha, out_dim_fecha)

_info(f"OK: fact_hidratacion_real_post_grado_destino={len(fact)} -> {out_fact}")
_info(f"OK: dim_hidratacion_dh_grado_destino={len(dim_dh)} -> {out_dim_dh}")
_info(f"OK: dim_hidratacion_fecha_post_grado_destino={len(dim_fecha)} -> {out_dim_fecha}")
_info(f"hidr_pct FINAL describe:\n" + fact["hidr_pct"].describe().to_string())

if __name__ == "__main__":
    main()

```

```
-----  
[158/164] FILE: \src\silver\build_milestone_window_ciclo_final_with_inference.py  
-----
```

```
from __future__ import annotations
```

```
from pathlib import Path  
from datetime import datetime  
import pandas as pd  
import numpy as np  
import yaml
```

```
from common.io import read_parquet, write_parquet
```

```
def load_settings() -> dict:  
    with open("config/settings.yaml", "r", encoding="utf-8") as f:  
        return yaml.safe_load(f)
```

```
def _to_date(s: pd.Series) -> pd.Series:  
    return pd.to_datetime(s, errors="coerce").dt.normalize()
```

```
def _canon_str(s: pd.Series) -> pd.Series:  
    return s.astype(str).str.strip().str.upper()
```

```
def _pick_first_existing(df: pd.DataFrame, candidates: list[str]) -> str | None:  
    for c in candidates:  
        if c in df.columns:  
            return c  
    return None
```

```
def main() -> None:
```

```
    cfg = load_settings()  
    silver_dir = Path(cfg["paths"]["silver"])
```

```
    milestones_path = silver_dir / "milestones_ciclo_final.parquet"  
    maestro_path = silver_dir / "fact_ciclo_maestro.parquet"  
    med_path = silver_dir / "dim_mediana_etapas_tipo_sp_variedad_area.parquet"
```

```
    if not milestones_path.exists():  
        raise FileNotFoundError(f"No existe: {milestones_path}")  
    if not maestro_path.exists():  
        raise FileNotFoundError(f"No existe: {maestro_path}")  
    if not med_path.exists():  
        raise FileNotFoundError(f"No existe: {med_path}")
```

```
    m = read_parquet(milestones_path).copy()  
    c = read_parquet(maestro_path).copy()  
    med = read_parquet(med_path).copy()
```

```
    # ---- Maestro  
    c["ciclo_id"] = c["ciclo_id"].astype(str)  
    for col in ["tipo_sp", "variedad", "area", "estado"]:  
        if col not in c.columns:  
            c[col] = "UNKNOWN"  
        c[col] = _canon_str(c[col])
```

```
    var_map = (cfg.get("mappings", {}).get("variedad_map", {}) or {})  
    var_map = {str(k).strip().upper(): str(v).strip().upper() for k, v in var_map.items()}  
    c["variedad_std"] = c["variedad"].map(lambda x: var_map.get(x, x))
```

```
    sp_col = _pick_first_existing(c, ["fecha_sp", "sp_date", "s_p", "sp"])  
    if sp_col is None:  
        raise ValueError("fact_ciclo_maestro: no encontré columna fecha_sp (o equivalente).")  
    c["fecha_sp"] = _to_date(c[sp_col])
```

```
    # ---- Milestones reales (pivot)  
    m["ciclo_id"] = m["ciclo_id"].astype(str)
```

```

m["milestone_code"] = _canon_str(m["milestone_code"])
m["fecha"] = _to_date(m["fecha"])

piv = (
    m.pivot_table(index="ciclo_id", columns="milestone_code", values="fecha", aggfunc="min")
    .reset_index()
)
for col in ["VEG_START", "HARVEST_START", "HARVEST_END", "POST_START", "POST_END"]:
    if col in piv.columns:
        piv[col] = _to_date(piv[col])

# ---- Base por ciclo
seg = c[["ciclo_id", "tipo_sp", "variedad_std", "area", "fecha_sp", "estado"]].drop_duplicates("ciclo_id")
base = seg.merge(piv, on="ciclo_id", how="left")

# ---- Medianas por segmento + fallback global
for col in ["tipo_sp", "variedad_std", "area"]:
    med[col] = _canon_str(med[col])

med["mediana_dias_veg"] = pd.to_numeric(med["mediana_dias_veg"], errors="coerce")
med["mediana_dias_harvest"] = pd.to_numeric(med["mediana_dias_harvest"], errors="coerce")
if "mediana_dias_post" in med.columns:
    med["mediana_dias_post"] = pd.to_numeric(med["mediana_dias_post"], errors="coerce")
else:
    med["mediana_dias_post"] = np.nan

base = base.merge(
    med[["tipo_sp", "variedad_std", "area", "mediana_dias_veg", "mediana_dias_harvest", "mediana_dias_post"]],
    on=["tipo_sp", "variedad_std", "area"],
    how="left",
)

g_veg = float(np.nanmedian(med["mediana_dias_veg"].values)) if len(med) else 60.0
g_harv = float(np.nanmedian(med["mediana_dias_harvest"].values)) if len(med) else 30.0
g_post = float(np.nanmedian(med["mediana_dias_post"].values)) if len(med) else 180.0

if not np.isfinite(g_veg):
    g_veg = 60.0
if not np.isfinite(g_harv):
    g_harv = 30.0
if not np.isfinite(g_post):
    g_post = 180.0

base["mediana_dias_veg"] = base["mediana_dias_veg"].fillna(g_veg).clip(0, 180)
base["mediana_dias_harvest"] = base["mediana_dias_harvest"].fillna(g_harv).clip(1, 180)
base["mediana_dias_post"] = base["mediana_dias_post"].fillna(g_post).clip(1, 365)

# ---- Resolver fechas finales (real si existe; si no inferir)
base["veg_start_final"] = base.get("VEG_START")
base.loc[base["veg_start_final"].isna(), "veg_start_final"] = base.loc[base["veg_start_final"].isna(), "fecha_sp"]

base["harvest_start_final"] = base.get("HARVEST_START")
miss_hs = base["harvest_start_final"].isna() & base["veg_start_final"].notna()
base.loc[miss_hs, "harvest_start_final"] = base.loc[miss_hs, "veg_start_final"] + pd.to_timedelta(
    base.loc[miss_hs, "mediana_dias_veg"], unit="D"
)

base["harvest_end_final"] = base.get("HARVEST_END")
miss_he = base["harvest_end_final"].isna() & base["harvest_start_final"].notna()
base.loc[miss_he, "harvest_end_final"] = base.loc[miss_he, "harvest_start_final"] + pd.to_timedelta(
    base.loc[miss_he, "mediana_dias_harvest"] - 1, unit="D"
)

base["post_start_final"] = base.get("POST_START")
miss_ps = base["post_start_final"].isna() & base["harvest_end_final"].notna()
base.loc[miss_ps, "post_start_final"] = base.loc[miss_ps, "harvest_end_final"] + pd.to_timedelta(1, unit="D")

base["post_end_final"] = base.get("POST_END")
miss_pe = base["post_end_final"].isna() & base["post_start_final"].notna()

```

```

base.loc[miss_pe, "post_end_final"] = base.loc[miss_pe, "post_start_final"] + pd.to_timedelta(
    base.loc[miss_pe, "mediana_dias_post"] - 1, unit="D"
)

# ---- Ventanas
rows: list[pd.DataFrame] = []

def add(stage: str, s_col: str, e_col: str, rule: str) -> None:
    tmp = base[["ciclo_id", s_col, e_col]].copy()
    tmp = tmp.rename(columns={s_col: "start_date", e_col: "end_date"})
    tmp["stage"] = stage
    tmp["rule"] = rule
    rows.append(tmp)

veg_end = base["harvest_start_final"] - pd.to_timedelta(1, unit="D")
tmp_veg = base[["ciclo_id", "veg_start_final"]].copy()
tmp_veg["start_date"] = tmp_veg["veg_start_final"]
tmp_veg["end_date"] = veg_end
tmp_veg["stage"] = "VEG"
tmp_veg["rule"] = "VEG_START (real o SP) -> HARVEST_START-1 (real o inferido)"
rows.append(tmp_veg[["ciclo_id", "stage", "start_date", "end_date", "rule"]])

add("HARVEST", "harvest_start_final", "harvest_end_final", "HARVEST (real o inferido)")
add("POST", "post_start_final", "post_end_final", "POST (real o inferido/regla)")

win = pd.concat(rows, ignore_index=True)

win["start_date"] = _to_date(win["start_date"])
win["end_date"] = _to_date(win["end_date"])

win = win[win["start_date"].notna() & win["end_date"].notna()].copy()
win = win[win["start_date"] <= win["end_date"]].copy()

win["created_at"] = datetime.now().isoformat(timespec="seconds")

out_path = silver_dir / "milestone_window_ciclo_final.parquet"
write_parquet(win, out_path)

print(f"OK: {out_path} | rows={len(win):,}")
print("Stage counts:\n", win["stage"].value_counts(dropna=False).to_string())

# auditoría cobertura harvest
cyc_total = win["ciclo_id"].nunique()
cyc_h = win[win["stage"].eq("HARVEST")]["ciclo_id"].nunique()
print("% ciclos con HARVEST (windows):", round(cyc_h / max(cyc_total, 1) * 100, 2))

if __name__ == "__main__":
    main()

-----
[159/164] FILE: \src\silver\build_milestones_final.py
-----
# src/silver/build_milestones_final.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import yaml

from common.io import read_parquet, write_parquet

# -----
# Config / helpers
# -----
def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

```



```

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _ensure_cols(df: pd.DataFrame, required: set[str], df_name: str) -> None:
    missing = required - set(df.columns)
    if missing:
        raise ValueError(f"{df_name}: faltan columnas requeridas: {sorted(missing)}. Columnas={list(df.columns)}")

def _as_str(df: pd.DataFrame, col: str) -> None:
    if col in df.columns:
        df[col] = df[col].astype(str).str.strip()

def main() -> None:
    cfg = load_settings()

    silver_dir = Path(cfg["paths"]["silver"])
    preds_dir = Path(cfg.get("paths", {})).get("preds", "data/preds")

    fact_path = silver_dir / "fact_milestones_ciclo.parquet"
    pred_path = preds_dir / "pred_milestones_ciclo.parquet"

    if not fact_path.exists():
        raise FileNotFoundError(f"No existe: {fact_path}")
    if not pred_path.exists():
        raise FileNotFoundError(
            f"No existe: {pred_path}. Ejecuta primero build_pred_milestones_baseline."
        )

    run_ts = datetime.now().isoformat(timespec="seconds")

    fact = read_parquet(fact_path).copy()
    pred = read_parquet(pred_path).copy()

    fact.columns = [str(c).strip() for c in fact.columns]
    pred.columns = [str(c).strip() for c in pred.columns]

    # -----
    # Validaciones mínimas
    # -----
    _ensure_cols(fact, {"ciclo_id", "milestone_code", "fecha"}, "fact_milestones_ciclo")
    # source puede no existir en algunos builds antiguos; si no está, lo llenamos
    if "source" not in fact.columns:
        fact["source"] = "FACT"

    _ensure_cols(pred, {"ciclo_id", "milestone_code", "fecha_pred"}, "pred_milestones_ciclo")
    # method/model_version/created_at pueden variar; creamos defaults si faltan
    if "method" not in pred.columns:
        pred["method"] = "BASELINE"
    if "model_version" not in pred.columns:
        pred["model_version"] = "unknown"
    if "created_at" not in pred.columns:
        pred["created_at"] = run_ts

    # Normalizar ids/códigos como string (estable)
    for c in ["ciclo_id", "milestone_code"]:
        _as_str(fact, c)
        _as_str(pred, c)

    # Fechas
    fact["fecha"] = _norm_date(fact["fecha"])
    pred["fecha_pred"] = _norm_date(pred["fecha_pred"])

    # Filtrar filas inválidas (evita que entren NaT y generen joins basura)
    fact = fact[fact["ciclo_id"].notna() & fact["milestone_code"].notna() & fact["fecha"].notna()].copy()
    pred = pred[pred["ciclo_id"].notna() & pred["milestone_code"].notna() & pred["fecha_pred"].notna()].copy()

    # -----

```

```

# Estandarizar a esquema común
# -----
fact2 = fact[["ciclo_id", "milestone_code", "fecha", "source"]].copy()
fact2["kind"] = "FACT"
fact2["method"] = fact2["source"].astype(str)
fact2["model_version"] = "FACT"
fact2["created_at"] = run_ts

pred2 = pred[["ciclo_id", "milestone_code", "fecha_pred", "method", "model_version", "created_at"]].copy()
pred2 = pred2.rename(columns={"fecha_pred": "fecha"})
pred2["kind"] = "PRED"
# "source" para PRED = model_version (útil para auditoría)
pred2["source"] = pred2["model_version"].astype(str)

# -----
# Unir y priorizar
# 1) FACT sobre PRED
# 2) Entre PRED, el más reciente (created_at mayor) si hay duplicados
# -----
allm = pd.concat(
    [
        fact2[["ciclo_id", "milestone_code", "fecha", "kind", "method", "model_version", "source", "created_at"]],
        pred2[["ciclo_id", "milestone_code", "fecha", "kind", "method", "model_version", "source", "created_at"]],
    ],
    ignore_index=True,
)

# Priorización
allm["__prio_kind"] = (allm["kind"] == "FACT").astype(int)
# Para ordenar created_at como datetime (si viene como string ISO)
allm["__created_at_dt"] = pd.to_datetime(allm["created_at"], errors="coerce")

final = (
    allm.sort_values(
        ["ciclo_id", "milestone_code", "__prio_kind", "__created_at_dt"],
        ascending=[True, True, False, False],
    )
    .drop_duplicates(subset=["ciclo_id", "milestone_code"], keep="first")
    .drop(columns=["__prio_kind", "__created_at_dt"])
    .reset_index(drop=True)
)

out_path = silver_dir / "milestones_ciclo_final.parquet"
write_parquet(final, out_path)

print(f"OK: milestones_ciclo_final={len(final)} filas -> {out_path}")
print("Counts by kind:\n", final["kind"].value_counts(dropna=False).to_string())

if __name__ == "__main__":
    main()

```

```

-----
[160/164] FILE: \src\silver\build_milestones_windows.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

```

```

from common.io import read_parquet, write_parquet

```

```

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

```

```

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _ensure_required_cols(df: pd.DataFrame, required: set[str], name: str) -> None:
    missing = required - set(df.columns)
    if missing:
        raise ValueError(f"{name}: faltan columnas requeridas: {sorted(missing)}")

def build_fact_milestones(fact_ciclo: pd.DataFrame, dh_days: int, post_tail_days: int) -> pd.DataFrame:
    """
    Devuelve tabla larga:
    ciclo_id, milestone_code, fecha, source, created_at
    """
    df = fact_ciclo.copy()

    required = {"ciclo_id", "fecha_sp", "fecha_inicio_cosecha", "fecha_fin_cosecha"}
    _ensure_required_cols(df, required, "fact_ciclo_maestro")

    df["ciclo_id"] = df["ciclo_id"].astype(str).str.strip()
    df["fecha_sp"] = _norm_date(df["fecha_sp"])
    df["fecha_inicio_cosecha"] = _norm_date(df["fecha_inicio_cosecha"])
    df["fecha_fin_cosecha"] = _norm_date(df["fecha_fin_cosecha"])

    rows: list[pd.DataFrame] = []

    def add_milestone(code: str, fecha: pd.Series, source: str) -> None:
        tmp = df[["ciclo_id"]].copy()
        tmp["milestone_code"] = code
        tmp["fecha"] = _norm_date(fecha)
        tmp["source"] = source
        rows.append(tmp)

    add_milestone("VEG_START", df["fecha_sp"], "fenograma")
    add_milestone("HARVEST_START", df["fecha_inicio_cosecha"], "fenograma/balanza")
    add_milestone("HARVEST_END", df["fecha_fin_cosecha"], "fenograma/balanza")

    post_start = df["fecha_inicio_cosecha"] + pd.to_timedelta(int(dh_days), unit="D")
    add_milestone("POST_START", post_start, f"rule:harvest_start+{int(dh_days)}")

    post_end = df["fecha_fin_cosecha"] + pd.to_timedelta(int(dh_days) + int(post_tail_days), unit="D")
    add_milestone("POST_END", post_end, f"rule:harvest_end+{int(dh_days)}+{int(post_tail_days)}")

    milestones = pd.concat(rows, ignore_index=True)

    milestones = milestones[milestones["fecha"].notna()].copy()
    milestones["created_at"] = datetime.now().isoformat(timespec="seconds")

    # Unicidad por ciclo_id + milestone_code
    milestones = (
        milestones.sort_values(["ciclo_id", "milestone_code", "fecha"], ascending=[True, True, True])
        .drop_duplicates(subset=["ciclo_id", "milestone_code"], keep="first")
        .reset_index(drop=True)
    )

    return milestones

def build_windows_from_milestones(fact_ciclo: pd.DataFrame, dh_days: int, post_tail_days: int) -> pd.DataFrame:
    """
    Ventanas por etapa en formato largo:
    ciclo_id, stage, start_date, end_date, rule
    Reglas:
    VEG: fecha_sp -> (harvest_start - 1)
    HARVEST: harvest_start -> harvest_end
    POST: (harvest_start+dh) -> (harvest_end+dh+tail)
    """
    df = fact_ciclo.copy()

```

```

required = {"ciclo_id", "fecha_sp", "fecha_inicio_cosecha", "fecha_fin_cosecha"}
_ensure_required_cols(df, required, "fact_ciclo_maestro")

df["ciclo_id"] = df["ciclo_id"].astype(str).str.strip()
df["fecha_sp"] = _norm_date(df["fecha_sp"])
df["fecha_inicio_cosecha"] = _norm_date(df["fecha_inicio_cosecha"])
df["fecha_fin_cosecha"] = _norm_date(df["fecha_fin_cosecha"])

# VEG
veg_start = df["fecha_sp"]
veg_end = df["fecha_inicio_cosecha"] - pd.to_timedelta(1, unit="D")
veg_end = veg_end.where(df["fecha_inicio_cosecha"].notna(), pd.NaT)

# HARVEST
harv_start = df["fecha_inicio_cosecha"]
harv_end = df["fecha_fin_cosecha"]

# POST
post_start = df["fecha_inicio_cosecha"] + pd.to_timedelta(int(dh_days), unit="D")
post_end = df["fecha_fin_cosecha"] + pd.to_timedelta(int(dh_days) + int(post_tail_days), unit="D")
post_start = post_start.where(df["fecha_inicio_cosecha"].notna(), pd.NaT)
post_end = post_end.where(df["fecha_fin_cosecha"].notna(), pd.NaT)

windows: list[pd.DataFrame] = []

def add_window(stage: str, s: pd.Series, e: pd.Series, rule: str) -> None:
    tmp = df[["ciclo_id"]].copy()
    tmp["stage"] = stage
    tmp["start_date"] = _norm_date(s)
    tmp["end_date"] = _norm_date(e)
    tmp["rule"] = rule
    windows.append(tmp)

add_window("VEG", veg_start, veg_end, "fecha_sp -> harvest_start-1")
add_window("HARVEST", harv_start, harv_end, "harvest_start -> harvest_end")
add_window("POST", post_start, post_end, f"harvest_start+{int(dh_days)} -> harvest_end+{int(dh_days)}+{int(post_tail_days)}")

out = pd.concat(windows, ignore_index=True)

# Quitar ventanas sin start_date
out = out[out["start_date"].notna()].copy()

# Regla de integridad: end_date no puede ser < start_date
bad = out["end_date"].notna() & (out["end_date"] < out["start_date"])
out.loc[bad, "end_date"] = pd.NaT

out["created_at"] = datetime.now().isoformat(timespec="seconds")

# Dedup defensivo
out = (
    out.sort_values(["ciclo_id", "stage", "start_date"], ascending=[True, True, True])
    .drop_duplicates(subset=["ciclo_id", "stage"], keep="first")
    .reset_index(drop=True)
)

return out

def main() -> None:
    cfg = load_settings()

    silver_dir = Path(cfg["paths"]["silver"])
    fact_path = silver_dir / "fact_ciclo_maestro.parquet"
    if not fact_path.exists():
        raise FileNotFoundError(f"No existe: {fact_path}")

    fact = read_parquet(fact_path)

    dh_days = int(cfg.get("milestones", {}).get("dh_days", 7))
    post_tail_days = int(cfg.get("milestones", {}).get("post_tail_days", 2))

```

```

milestones = build_fact_milestones(fact, dh_days=dh_days, post_tail_days=post_tail_days)
windows = build_windows_from_milestones(fact, dh_days=dh_days, post_tail_days=post_tail_days)

write_parquet(milestones, silver_dir / "fact_milestones_ciclo.parquet")
write_parquet(windows, silver_dir / "milestone_window_ciclo.parquet")

print(f"OK milestones: {len(milestones)} filas -> fact_milestones_ciclo.parquet")
print(f"OK windows: {len(windows)} filas -> milestone_window_ciclo.parquet")
print(f"Reglas: DH={dh_days}, post_tail={post_tail_days}")

if __name__ == "__main__":
    main()

-----
[161/164] FILE: \src\silver\build_patch_ciclo_maestro_from_pred_tallos.py
-----
# src/silver/build_patch_ciclo_maestro_from_pred_tallos.py
from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import numpy as np
import yaml

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_cols(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()
    df.columns = [str(c).strip() for c in df.columns]
    return df

def _to_dt_norm(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def _to_num(s: pd.Series) -> pd.Series:
    ss = s.copy()
    if ss.dtype == object or pd.api.types.is_string_dtype(ss):
        ss = (
            ss.astype("string")
            .str.replace(" ", "", regex=False)
            .str.replace(",", ".", regex=False)
        )
    return pd.to_numeric(ss, errors="coerce")

def _build_fin_mask(df: pd.DataFrame, fin_col: str, fin_value: int) -> pd.Series:
    """
    - Si fin_col es datetime -> mask = notna()
    - Si fin_col es num/bool -> mask = (== fin_value)
    - Si es object -> intenta num; si no, intenta datetime; fallback strings
    """
    if fin_col not in df.columns:
        raise ValueError(f"No existe columna '{fin_col}' en fact_ciclo_maestro.")

    s = df[fin_col]

    if pd.api.types.is_datetime64_any_dtype(s):
        return s.notna()

    sn = _to_num(s)

```

```

if sn.notna().sum() > 0:
    return sn.fillna(0).astype("Int64").eq(int(fin_value))

sd = pd.to_datetime(s, errors="coerce")
if sd.notna().sum() > 0:
    return sd.notna()

st = s.astype("string").str.strip().str.lower()
return st.isin([str(fin_value), "true", "si", "sí", "yes", "y"])

def main() -> None:
    cfg = load_settings()

    # dirs
    data_root = Path(cfg["paths"].get("data_root", "data"))
    silver_dir = Path(cfg["paths"]["silver"])
    preds_dir = Path(cfg["paths"].get("preds", "data/preds"))

    # resolvemos rutas relativas al repo root actual (CWD = repo root usualmente)
    silver_dir = Path(silver_dir)
    preds_dir = Path(preds_dir)

    scfg = (cfg.get("silver", {}) or {}).get("patch_ciclo_maestro_from_pred_tallos", {}) or {}
    enabled = bool(scfg.get("enabled", True))
    if not enabled:
        print("[INFO] patch_ciclo_maestro_from_pred_tallos: disabled")
        return

    in_ciclo_rel = scfg.get("in_ciclo", "fact_ciclo_maestro.parquet")
    in_pred_rel = scfg.get("in_pred_tallos", "../preds/pred_tallos_ciclo.parquet")
    out_report_rel = scfg.get("out_report", "fact_ciclo_maestro_patch_pred_tallos_report.parquet")

    ciclo_id_col = scfg.get("ciclo_id_col", "ciclo_id")
    fin_col = scfg.get("fin_col", "fecha_fin_cosecha")
    fin_value = int(scfg.get("fin_value", 1))
    tallos_new_col = scfg.get("tallos_new_col", "tallos_proy_n")

    require_positive = bool(scfg.get("require_positive", True))

    in_ciclo = (silver_dir / in_ciclo_rel).resolve()
    # in_pred puede venir como "../preds/..." relativo a silver_dir
    in_pred = (silver_dir / in_pred_rel).resolve()
    out_report = (silver_dir / out_report_rel).resolve()

    if not in_ciclo.exists():
        raise FileNotFoundError(f"No existe: {in_ciclo}")

    # =====
    # CONDICIÓN CLAVE: si preds no existe -> NO HACER NADA
    # =====
    if not in_pred.exists():
        print(f"[INFO] No existe pred_tallos_ciclo -> SKIP. ({in_pred})")
        # igual escribimos reporte ?skipped? para trazabilidad y para que el step tenga output
        rep = pd.DataFrame([
            {
                "patched_at": datetime.now().isoformat(timespec="seconds"),
                "status": "skipped_missing_pred",
                "in_ciclo": str(in_ciclo),
                "in_pred": str(in_pred),
                "rows_ciclo": None,
                "rows_pred": None,
                "rows_matched": None,
                "rows_replaced": None,
            }
        ])
        out_report.parent.mkdir(parents=True, exist_ok=True)
        write_parquet(rep, out_report)
        print(f"[INFO] report: {out_report}")
        return

    # =====
    # LECTURAS
    # =====

```

```

ciclo = _norm_cols(pd.read_parquet(in_ciclo))
pred = _norm_cols(pd.read_parquet(in_pred))

# validación columnas
if ciclo_id_col not in ciclo.columns:
    raise ValueError(f"fact_ciclo_maestro: falta columna '{ciclo_id_col}'")
if "tallos_proy" not in ciclo.columns:
    raise ValueError("fact_ciclo_maestro: falta columna 'tallos_proy'")
if fin_col not in ciclo.columns:
    raise ValueError(f"fact_ciclo_maestro: falta columna '{fin_col}'")

if ciclo_id_col not in pred.columns:
    raise ValueError(f"pred_tallos_ciclo: falta columna '{ciclo_id_col}'")
if tallos_new_col not in pred.columns:
    raise ValueError(f"pred_tallos_ciclo: falta columna '{tallos_new_col}'")

# normalizar tipos
ciclo[ciclo_id_col] = ciclo[ciclo_id_col].astype("string").str.strip()
pred[ciclo_id_col] = pred[ciclo_id_col].astype("string").str.strip()

ciclo["tallos_proy"] = _to_num(ciclo["tallos_proy"])
pred[tallos_new_col] = _to_num(pred[tallos_new_col])

# por seguridad: 1 ciclo_id -> 1 valor (tomamos max no nulo)
pred_slim = pred[[ciclo_id_col, tallos_new_col]].copy()
pred_slim = (
    pred_slim.groupby(ciclo_id_col, as_index=False)
    .agg(**{tallos_new_col: (tallos_new_col, "max")})
)

# merge
merged = ciclo.merge(pred_slim, on=ciclo_id_col, how="left")

fin_mask = _build_fin_mask(merged, fin_col=fin_col, fin_value=fin_value)
# fin_mask = True -> tiene fecha_fin_cosecha / fin=1 (ciclo cerrado)
# open_mask = True -> NO tiene fecha_fin_cosecha / fin!=1 (ciclo abierto)
open_mask = ~fin_mask

has_new = merged[tallos_new_col].notna()
if require_positive:
    has_new = has_new & (merged[tallos_new_col] > 0)

# ? Reemplazar solo si está ABIERTO (sin fecha_fin_cosecha) y hay tallos_proy_n
mask_replace = has_new & open_mask

rows_ciclo = int(len(merged))
rows_pred = int(len(pred_slim))
rows_matched = int(merged[tallos_new_col].notna().sum())
rows_replaced = int(mask_replace.sum())

if rows_replaced > 0:
    merged.loc[mask_replace, "tallos_proy"] = merged.loc[mask_replace, tallos_new_col]

# limpiar col auxiliar
merged = merged.drop(columns=[tallos_new_col])

# overwrite maestro
write_parquet(merged, in_ciclo)

# reporte
rep = pd.DataFrame([
    "patched_at": datetime.now().isoformat(timespec="seconds"),
    "status": "patched",
    "rows_ciclo": rows_ciclo,
    "rows_pred": rows_pred,
    "rows_matched_new": rows_matched,
    "rows_replaced": rows_replaced,
    "require_positive": require_positive,
    "fin_col": fin_col,
    "fin_value": fin_value,
    "tallos_new_col": tallos_new_col,
])

```

```

        "in_ciclo": str(in_ciclo),
        "in_pred": str(in_pred),
        "out_report": str(out_report),
    })

    out_report.parent.mkdir(parents=True, exist_ok=True)
    write_parquet(rep, out_report)

    print(f"OK patch ciclo_maestro from preds: replaced={rows_replaced} matched_new={rows_matched} rows={rows_
ciclo}")
    print(f" - updated: {in_ciclo}")
    print(f" - report : {out_report}")

if __name__ == "__main__":
    main()

-----
[162/164] FILE: \src\silver\build_weather_hour_estado.py
-----

from __future__ import annotations

from dataclasses import dataclass
from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd
import yaml

from common.io import write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _to_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce")

def _hour_floor(s: pd.Series) -> pd.Series:
    return _to_dt(s).dt.floor("h") # "H" deprecated

@dataclass(frozen=True)
class KardexParams:
    th_lluvia: float = 0.3 # mm/h
    th_seco: float = 0.3 # K <= th_seco => SECO
    th_humedo: float = 2.0 # K <= th_humedo => HUMEDO; > => MOJADO
    kardex_cap: float = 3.0 # tope mm

def compute_kardex_estado(df: pd.DataFrame, p: KardexParams) -> pd.DataFrame:
    """
    df: estación única, ordenada por fecha_hora
    salida: agrega kardex_prev_mm, kardex_mm, Estado_Kardex, En_Lluvia
    """
    out = df.copy()

    out["lluvia_mm"] = pd.to_numeric(out.get("lluvia_mm", 0), errors="coerce").fillna(0.0).astype(float)
    out["et_mm"] = pd.to_numeric(out.get("et_mm", 0), errors="coerce").fillna(0.0).astype(float)

    out = out.sort_values("fecha_hora").reset_index(drop=True)
    out["En_Lluvia"] = (out["lluvia_mm"] >= float(p.th_lluvia)).astype("int8")

    # Recurrencia: Kt = clip(Kt-1 + lluvia - ET, 0, cap)
    k = np.zeros(len(out), dtype=float)
    for i in range(len(out)):
        delta = float(out.loc[i, "lluvia_mm"]) - float(out.loc[i, "et_mm"])
        if i == 0:

```



```

        k[i] = np.clip(delta, 0.0, float(p.kardex_cap))
    else:
        k[i] = np.clip(k[i - 1] + delta, 0.0, float(p.kardex_cap))

out["kardex_prev_mm"] = np.r_[0.0, k[:-1]]
out["kardex_mm"] = k

def estado(x: float) -> str:
    if x <= float(p.th_seco):
        return "SECO"
    if x <= float(p.th_humedo):
        return "HUMEDO"
    return "MOJADO"

out["Estado_Kardex"] = out["kardex_mm"].map(estado)
return out

def _standardize_bronze_weather(df: pd.DataFrame, station_expected: str | None = None) -> pd.DataFrame:
    """
    Estandariza los BRONZE existentes:
    - fecha -> fecha_hora (floor hora)
    - rainfall_mm -> lluvia_mm
    - et -> et_mm
    - station upper
    Mantiene el resto de columnas (temp_avg, solar_energy, etc.) para usarlas después si quieres.
    """
    d = df.copy()
    d.columns = [str(c).strip() for c in d.columns]

    if "fecha" not in d.columns:
        raise ValueError(f"Bronze weather: falta columna 'fecha'. Columnas={list(d.columns)}")

    d = d.rename(columns={"fecha": "fecha_hora"})
    d["fecha_hora"] = _hour_floor(d["fecha_hora"])
    d = d[d["fecha_hora"].notna()].copy()

    # Renombres a esquema canónico
    if "rainfall_mm" in d.columns and "lluvia_mm" not in d.columns:
        d = d.rename(columns={"rainfall_mm": "lluvia_mm"})
    if "et" in d.columns and "et_mm" not in d.columns:
        d = d.rename(columns={"et": "et_mm"})

    # Asegurar columnas mínimas
    if "lluvia_mm" not in d.columns:
        d["lluvia_mm"] = np.nan
    if "et_mm" not in d.columns:
        d["et_mm"] = np.nan

    d["lluvia_mm"] = pd.to_numeric(d["lluvia_mm"], errors="coerce").fillna(0.0)
    d["et_mm"] = pd.to_numeric(d["et_mm"], errors="coerce").fillna(0.0)

    if "station" not in d.columns:
        # Por seguridad (aunque tus scripts BRONZE sí la ponen)
        d["station"] = station_expected if station_expected else "UNKNOWN"

    d["station"] = d["station"].astype(str).str.upper().str.strip().replace({"A-4": "A4", "SJP": "A4"})
    if station_expected:
        d = d[d["station"].eq(station_expected.upper())].copy()

    # Deduplicación técnica (si existen duplicados por hora)
    d = d.sort_values("fecha_hora").drop_duplicates(subset=["station", "fecha_hora"], keep="last")
    return d

def main() -> None:
    cfg = load_settings()

    bronze_dir = Path(cfg["paths"]["bronze"])
    silver_dir = Path(cfg["paths"]["silver"])
    silver_dir.mkdir(parents=True, exist_ok=True)

```

```

p_main = bronze_dir / "weather_hour_main.parquet"
p_a4 = bronze_dir / "weather_hour_a4.parquet"

if not p_main.exists():
    raise FileNotFoundError(f"No existe Bronze: {p_main}. Ejecuta src/bronze/build_weather_hour_main.py")
if not p_a4.exists():
    raise FileNotFoundError(f"No existe Bronze: {p_a4}. Ejecuta src/bronze/build_weather_hour_a4.py")

df_main = pd.read_parquet(p_main)
df_a4 = pd.read_parquet(p_a4)

main_std = _standardize_bronze_weather(df_main, station_expected="MAIN")
a4_std = _standardize_bronze_weather(df_a4, station_expected="A4")

weather = pd.concat([main_std, a4_std], ignore_index=True)
weather = weather[weather["station"].isin(["MAIN", "A4"])]

# Parámetros kardex
p = KardexParams(
    th_lluvia=float(cfg.get("weather", {}).get("th_lluvia", 0.3)),
    th_seco=float(cfg.get("weather", {}).get("th_seco", 0.3)),
    th_humedo=float(cfg.get("weather", {}).get("th_humedo", 2.0)),
    kardex_cap=float(cfg.get("weather", {}).get("kardex_cap", 3.0)),
)

out_frames = []
for station, g in weather.groupby("station", dropna=False):
    out_frames.append(compute_kardex_estado(g, p))

out = pd.concat(out_frames, ignore_index=True)
out["created_at"] = datetime.now().isoformat(timespec="seconds")

out_path = silver_dir / "weather_hour_estado.parquet"
write_parquet(out, out_path)

print(f"OK: weather_hour_estado={len(out)} filas -> {out_path}")
print("Stations:", out["station"].value_counts(dropna=False).to_dict())
print("Rango:", out["fecha_hora"].min(), "->", out["fecha_hora"].max())
print("Estado_Kardex:", out["Estado_Kardex"].value_counts(dropna=False).to_dict())

if __name__ == "__main__":
    main()

```

```

-----
[163/164] FILE: \src\silver\build_weather_hour_wide.py
-----

```

```

from __future__ import annotations

```

```

from pathlib import Path
from datetime import datetime
import numpy as np
import pandas as pd
import yaml

```

```

from common.io import write_parquet

```

```

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

```

```

def _to_dt(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce")

```

```

def _to_num(s: pd.Series) -> pd.Series:
    return pd.to_numeric(s, errors="coerce")

```

```

def _hour_floor(s: pd.Series) -> pd.Series:
    return _to_dt(s).dt.floor("h")

def _info(msg: str) -> None:
    print(f"[INFO] {msg}")

def _warn(msg: str) -> None:
    print(f"[WARN] {msg}")

def _standardize_bronze_weather(df: pd.DataFrame, station_expected: str | None = None) -> pd.DataFrame:
    """
    Normaliza BRONZE weather_hour_{main,a4}.parquet a un esquema común:
    - fecha -> fecha_hora (floor hora)
    - station upper
    - asegura presence de columnas esperadas (con NaN si faltan)
    """
    d = df.copy()
    d.columns = [str(c).strip() for c in d.columns]

    if "fecha" not in d.columns:
        raise ValueError(f"Bronze weather: falta columna 'fecha'. Columnas={list(d.columns)}")

    d["fecha_hora"] = _hour_floor(d["fecha"])
    d = d[d["fecha_hora"].notna()].copy()

    if "station" not in d.columns:
        d["station"] = station_expected if station_expected else "UNKNOWN"

    d["station"] = (
        d["station"].astype(str).str.upper().str.strip()
        .replace({"A-4": "A4", "SJP": "A4"})
    )
    if station_expected:
        d = d[d["station"].eq(station_expected.upper())].copy()

    # Columnas posibles (las que ya traes en BRONZE)
    wanted = [
        "rainfall_mm", "et", "temp_avg", "wind_dir_of_avg", "wind_run",
        "solar_rad_avg", "solar_rad_hi", "solar_energy",
        "hum_last", "uv_index_avg", "wind_speed_avg",
    ]
    for c in wanted:
        if c not in d.columns:
            d[c] = np.nan

    # Tipos numéricos
    for c in wanted:
        d[c] = _to_num(d[c])

    # Deduplicación técnica por station+hora
    d = d.sort_values("fecha_hora").drop_duplicates(subset=["station", "fecha_hora"], keep="last")
    return d

def add_solar_joules(df: pd.DataFrame) -> pd.DataFrame:
    out = df.copy()

    # fallback: si no hay solar_rad_avg pero sí solar_rad_hi, usar hi como proxy
    if out["solar_rad_avg"].isna().all() and not out["solar_rad_hi"].isna().all():
        _warn("solar_rad_avg vacío; usando solar_rad_hi como proxy.")
        out["solar_rad_avg"] = out["solar_rad_hi"]

    # Joules por hora: W/m2 * 3600 s
    out["solar_energy_j_m2"] = out["solar_rad_avg"].astype(float) * 3600.0
    return out

def main() -> None:
    cfg = load_settings()

```

```

bronze_dir = Path(cfg["paths"]["bronze"])
silver_dir = Path(cfg["paths"]["silver"])
silver_dir.mkdir(parents=True, exist_ok=True)

# Inputs BRONZE
p_main = bronze_dir / "weather_hour_main.parquet"
p_a4 = bronze_dir / "weather_hour_a4.parquet"
if not p_main.exists():
    raise FileNotFoundError(f"No existe BRONZE: {p_main}. Ejecuta build_weather_hour_main.py")
if not p_a4.exists():
    raise FileNotFoundError(f"No existe BRONZE: {p_a4}. Ejecuta build_weather_hour_a4.py")

# Input SILVER (estado ya calculado)
p_estado = silver_dir / "weather_hour_estado.parquet"
if not p_estado.exists():
    raise FileNotFoundError(
        f"No existe SILVER: {p_estado}. Ejecuta primero src/silver/build_weather_hour_estado.py"
    )

df_main = pd.read_parquet(p_main)
df_a4 = pd.read_parquet(p_a4)
est = pd.read_parquet(p_estado)

main_std = _standardize_bronze_weather(df_main, station_expected="MAIN")
a4_std = _standardize_bronze_weather(df_a4, station_expected="A4")
raw = pd.concat([main_std, a4_std], ignore_index=True)

# Normalizar/validar estado
est.columns = [str(c).strip() for c in est.columns]
# En el script de estado que te dejé: fecha_hora + station + Estado_Kardex + En_Lluvia + kardex_mm...
required_estado = {"fecha_hora", "station", "Estado_Kardex", "En_Lluvia", "kardex_mm", "kardex_prev_mm"}
missing = required_estado - set(est.columns)
if missing:
    raise ValueError(
        f"weather_hour_estado.parquet no tiene columnas requeridas: {sorted(missing)}. "
        "Alinea build_weather_hour_estado.py con ese esquema."
    )

est["fecha_hora"] = _hour_floor(est["fecha_hora"])
est["station"] = est["station"].astype(str).str.upper().str.strip()

# Join raw + estado
wide = raw.merge(
    est[["fecha_hora", "station", "En_Lluvia", "kardex_prev_mm", "kardex_mm", "Estado_Kardex"]],
    on=["fecha_hora", "station"],
    how="left",
)

# Imputación conservadora si faltan horas de estado (no debería, pero cubrimos gaps)
miss = int(wide["Estado_Kardex"].isna().sum())
if miss > 0:
    _warn(f"Hay {miss} filas sin estado (join). Se imputa SECO/0 y kardex=0. Revisa cobertura.")
    wide["Estado_Kardex"] = wide["Estado_Kardex"].fillna("SECO")
    wide["En_Lluvia"] = _to_num(wide["En_Lluvia"]).fillna(0).astype(int)
    wide["kardex_prev_mm"] = _to_num(wide["kardex_prev_mm"]).fillna(0.0)
    wide["kardex_mm"] = _to_num(wide["kardex_mm"]).fillna(0.0)

# Joules
wide = add_solar_joules(wide)

wide["created_at"] = datetime.now().isoformat(timespec="seconds")

# Output final: mantener nombres esperados por tus downstream (dt_hora/fecha si necesitas)
# En tus otros scripts tú usas weather_hour_wide con columnas tipo dt_hora/fecha.
# Aquí dejamos dt_hora = fecha_hora para que el join sea más directo.
wide = wide.rename(columns={"fecha_hora": "dt_hora"})

out_path = silver_dir / "weather_hour_wide.parquet"
write_parquet(wide, out_path)

_info(f"OK: weather_hour_wide={len(wide)} filas -> {out_path}")
_info(f"Stations: {wide['station'].value_counts(dropna=False).to_dict()}")

```

```

_info(f"Rango: {wide['dt_hora'].min()} -> {wide['dt_hora'].max()}")

if __name__ == "__main__":
    main()

-----
[164/164] FILE: \src\silver\build_windows_from_milestones_final.py
-----

from __future__ import annotations

from pathlib import Path
from datetime import datetime
import pandas as pd
import yaml

from common.io import read_parquet, write_parquet

def load_settings() -> dict:
    with open("config/settings.yaml", "r", encoding="utf-8") as f:
        return yaml.safe_load(f)

def _norm_date(s: pd.Series) -> pd.Series:
    return pd.to_datetime(s, errors="coerce").dt.normalize()

def main() -> None:
    cfg = load_settings()
    silver_dir = Path(cfg["paths"]["silver"])

    milestones_path = silver_dir / "milestones_ciclo_final.parquet"
    if not milestones_path.exists():
        raise FileNotFoundError(f"No existe: {milestones_path}. Ejecuta primero build_milestones_final.")

    m = read_parquet(milestones_path).copy()
    m["fecha"] = _norm_date(m["fecha"])

    # Pivot
    piv = (m.pivot_table(index="ciclo_id", columns="milestone_code", values="fecha", aggfunc="min")
           .reset_index())

    # columnas
    veg_start = piv.get("VEG_START")
    hs = piv.get("HARVEST_START")
    he = piv.get("HARVEST_END")
    ps = piv.get("POST_START")
    pe = piv.get("POST_END")

    # Ventanas
    rows = []

    def add(stage: str, s: pd.Series, e: pd.Series, rule: str):
        tmp = pd.DataFrame({"ciclo_id": piv["ciclo_id"], "stage": stage})
        tmp["start_date"] = s
        tmp["end_date"] = e
        tmp["rule"] = rule
        rows.append(tmp)

    # VEG: veg_start -> hs-1 (si hs existe)
    veg_end = hs - pd.to_timedelta(1, unit="D") if hs is not None else pd.Series([pd.NaT] * len(piv))
    veg_end = veg_end.where(hs.notna(), pd.NaT) if hs is not None else veg_end
    add("VEG", veg_start, veg_end, "VEG_START -> HARVEST_START-1")

    # HARVEST: hs -> he (si hs existe)
    add("HARVEST", hs, he, "HARVEST_START -> HARVEST_END")

    # POST: ps -> pe (si ps existe)
    add("POST", ps, pe, "POST_START -> POST_END")

```

```
win = pd.concat(rows, ignore_index=True)
win["start_date"] = _norm_date(win["start_date"])
win["end_date"] = _norm_date(win["end_date"])

win = win[win["start_date"].notna()].copy()
win["created_at"] = datetime.now().isoformat(timespec="seconds")

out_path = silver_dir / "milestone_window_ciclo_final.parquet"
write_parquet(win, out_path)

print(f"OK: milestone_window_ciclo_final={len(win)} filas -> {out_path}")
print("Stage counts:\n", win["stage"].value_counts().to_string())

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