Next-Year Financial Distress Prediction (Compustat Annual Panel) — Reproducible ML Pipeline

Goal. Predict the probability that a firm is in *financial distress* in fiscal year **t+1** using accounting (and permitted market) information available at fiscal year **t**.

Important scope note. The outcome is an **engineered distress proxy** (high leverage / balance-sheet stress), not a realized legal default or bankruptcy. The notebook is therefore a **predictive measurement and decision-support pipeline**, not a causal identification design.

Notebook structure (Data Science Lifecycle — 10 phases)

1. Problem Definition & Setup
2. Data Collection & Panel Integrity
3. Data Cleaning & Missingness Handling (leakage-aware)
4. Exploratory Data Analysis (EDA)
5. Feature Engineering & Target Construction
6. Preprocessing for Modeling (train-only fitting)
7. Model Selection & Training (7A Logit; 7B Trees)
8. Model Evaluation & Diagnostic Monitoring
9. Operational Risk Management Layer (Events + PDs)
10. Results Summary, Guardrails, and Replication Artifacts

> This organization mirrors the course lifecycle guidance and the project's technical review action items (see provided PDF and technical report).

How to run (replication package convention)

1. Place `data.csv` in the project root (or update `CONFIG["DATA_PATH"]` in Section 1).
2. Keep `Variables.xlsx` (variable dictionary) alongside the notebook for automatic documentation.
3. Run **Kernel → Restart & Run All**.

The notebook creates an `outputs/` folder containing:

- a predictions export (`predictions.csv`),
- configuration and threshold tables,
- model summary tables suitable for an appendix,
- figures saved as PNG for paper workflow.

1. Problem Definition & Setup

1.1 Prediction target, success metrics, and decision objective

- **Target (supervised label):** `target_next_v1` , `target_next_v2` , or `target_next_v3` (separate distress proxies). Downstream modeling uses `target_next_v2` by default.
- **Primary performance metrics (out-of-sample):**
  - ROC-AUC (ranking quality),
  - PR-AUC (class imbalance),
  - Brier score (probability accuracy / calibration).
- **Decision objective (screening):** convert predicted PDs into a review policy using:
  - **misclassification costs** (`COST_FN` , `COST_FP`) and
  - **capacity constraints** (screen top `CAPACITY_PCT` percent of firms).

This is a *risk scoring* workflow: calibrated probabilities and operational interpretability matter more than headline accuracy.

1.2 Configuration, determinism, and library versions

In [58]:
```
# Core numerics
import os
import sys
import math
import json
import warnings
from pathlib import Path
```

```python
from dataclasses import dataclass, asdict

import numpy as np
import pandas as pd

# ML / metrics
from sklearn.model_selection import ParameterGrid
from sklearn.preprocessing import StandardScaler
from sklearn.impute import KNNImputer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    roc_auc_score,
    average_precision_score,
    brier_score_loss,
    confusion_matrix,
    precision_recall_curve,
    roc_curve,
)
from sklearn.calibration import calibration_curve
from sklearn.isotonic import IsotonicRegression

# Stats / inference
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.stats.sandwich_covariance import cov_cluster, cov_cluster_2groups
from scipy import stats

# Trees / explainability
import xgboost as xgb

import matplotlib.pyplot as plt
from IPython.display import display

warnings.filterwarnings("ignore")

# ----------------------------
# Determinism
# ----------------------------
SEED = 42
np.random.seed(SEED)
USING_SYNTHETIC_DATA = False # Global flag for data mode

# ----------------------------
# Configuration (edit here)
# ----------------------------
CONFIG = {
    # Data inputs
    "DATA_PATH": "data.csv",
    "VARIABLES_XLSX_PATH": "Variables.xlsx",

    # Temporal splitting via label_year = fyear + 1
    "TRAIN_CUTOFF_LABEL_YEAR": 2022,   # label_year <= cutoff -> train/val pool; later -> test
    "VAL_YEARS": 1,                    # number of last label years inside the train pool used as validation

    # Missingness / imputation
    "KNN_K": 25,
    "IMPUTE_LO_Q": 0.01,
    "IMPUTE_HI_Q": 0.99,

    # Preprocessing
    "WINSOR_LO_Q": 0.01,
    "WINSOR_HI_Q": 0.99,

    # Logit hyperparameter search
    "LOGIT_C_GRID": [0.01, 0.1, 1.0, 10.0],

    # Tree model (XGBoost) parameters (conservative / regularized)
    "XGB_PARAMS": {
        "max_depth": 4,
        "min_child_weight": 5,
        "subsample": 0.8,
        "colsample_bytree": 0.8,
        "eta": 0.05,
        "reg_lambda": 10.0,
        "reg_alpha": 0.0,
        "objective": "binary:logistic",
        "eval_metric": "aucpr",
        "tree_method": "hist",
        "seed": SEED,
    },
    "XGB_NUM_BOOST_ROUND": 5000,
    "XGB_EARLY_STOPPING": 100,
```

```
    # Decision policy parameters (costs + capacity)
    "COST_FN": 10.0,
    "COST_FP": 1.0,
    "CAPACITY_PCT": 0.20,  # screen top 20% by PD as a capacity policy

    # Outputs
    "OUTPUT_DIR": "outputs",
    "FIG_DIR": "figures",
}

Path(CONFIG["OUTPUT_DIR"]).mkdir(parents=True, exist_ok=True)
Path(CONFIG["FIG_DIR"]).mkdir(parents=True, exist_ok=True)

print("CONFIG (key parameters):")
for k in ["DATA_PATH","TRAIN_CUTOFF_LABEL_YEAR","VAL_YEARS","KNN_K","WINSOR_LO_Q","WINSOR_HI_Q","COST_FN","COST_
    print(f"  {k}: {CONFIG[k]}")
print("\nPython:", sys.version.split()[0])
print("pandas:", pd.__version__)
print("numpy:", np.__version__)
```

```
CONFIG (key parameters):
  DATA_PATH: data.csv
  TRAIN_CUTOFF_LABEL_YEAR: 2022
  VAL_YEARS: 1
  KNN_K: 25
  WINSOR_LO_Q: 0.01
  WINSOR_HI_Q: 0.99
  COST_FN: 10.0
  COST_FP: 1.0
  CAPACITY_PCT: 0.2

Python: 3.13.5
pandas: 2.3.1
numpy: 2.2.5
```

1.3 Helper utilities (robust ratios, transforms, and reporting)

In [59]:

```
def signed_log1p(x: pd.Series) -> pd.Series:
    """Signed log1p transform: sign(x) * log1p(|x|). Preserves zero and sign, stabilizes tails."""
    x = pd.to_numeric(x, errors="coerce")
    return np.sign(x) * np.log1p(np.abs(x))

def safe_divide(numer: pd.Series, denom: pd.Series, denom_floor: float = None) -> pd.Series:
    """Safe divide with optional denominator floor for stability. Returns float with NaN where undefined."""
    numer = pd.to_numeric(numer, errors="coerce")
    denom = pd.to_numeric(denom, errors="coerce")
    if denom_floor is not None:
        denom = denom.where(denom.abs() >= denom_floor, other=np.sign(denom).replace(0, 1) * denom_floor)
    out = numer / denom
    out = out.replace([np.inf, -np.inf], np.nan)
    return out

def ensure_nullable_float(s: pd.Series) -> pd.Series:
    """Convert to pandas nullable Float64 to enable NA-aware comparisons (returns <NA> instead of False)."""
    return pd.to_numeric(s, errors="coerce").astype("Float64")

def winsorize_train_bounds(x: pd.Series, lo: float, hi: float) -> tuple[float, float]:
    """Return winsorization bounds computed on *training* observed values."""
    x = pd.to_numeric(x, errors="coerce")
    x_obs = x.dropna()
    if len(x_obs) == 0:
        return (np.nan, np.nan)
    return (float(x_obs.quantile(lo)), float(x_obs.quantile(hi)))

def apply_bounds(x: pd.Series, lo: float, hi: float) -> pd.Series:
    x = pd.to_numeric(x, errors="coerce")
    if np.isnan(lo) or np.isnan(hi):
        return x
    return x.clip(lower=lo, upper=hi)

def compute_smd(train: pd.Series, test: pd.Series) -> float:
    """Standardized mean difference (SMD): (mu_train - mu_test)/pooled_sd."""
    a = pd.to_numeric(train, errors="coerce").dropna()
    b = pd.to_numeric(test, errors="coerce").dropna()
    if len(a) < 2 or len(b) < 2:
        return np.nan
    mu_a, mu_b = a.mean(), b.mean()
    sd_a, sd_b = a.std(ddof=1), b.std(ddof=1)
    pooled = np.sqrt(0.5*(sd_a**2 + sd_b**2))
    return float((mu_a - mu_b) / pooled) if pooled > 0 else np.nan

def logit(p: np.ndarray, eps: float = 1e-6) -> np.ndarray:
    p = np.clip(p, eps, 1-eps)
```

```
    return np.log(p/(1-p))

def sigmoid(z: np.ndarray) -> np.ndarray:
    return 1/(1+np.exp(-z))

def print_df(df: pd.DataFrame, n: int = 10, name: str = None):
    if name:
        print(f"\n{name} (top {n} rows):")
    display(df.head(n))
```

2. Data Collection & Panel Integrity

2.1 Load variable dictionary (for documentation)

We load the provided variable dictionary (`Variables.xlsx`) to:

- validate required Compustat mnemonics exist in the data file,
- generate appendix-ready variable tables.

This step **does not** transform the modeling data.

In [60]:
```
vars_path = Path(CONFIG["VARIABLES_XLSX_PATH"])
if vars_path.exists():
    var_dict = pd.read_excel(vars_path, sheet_name=0)
    var_dict.columns = [c.strip() for c in var_dict.columns]
    print(f"Loaded variable dictionary with {len(var_dict)} rows from: {vars_path}")
    display(var_dict.head(90))
else:
    var_dict = pd.DataFrame(columns=["Variable","Two-word Description","Category"])
    print(f"WARNING: variable dictionary not found at {vars_path}. Continuing without it.")
```

```
Loaded variable dictionary with 89 rows from: Variables.xlsx
```

	Variable	Two-word Description	Category
0	aco	Other Current	Balance Sheet
1	act	Current Assets	Balance Sheet
2	ao	Other Assets	Balance Sheet
3	aoloch	Asset/Liability Δ	Cash Flow
4	ap	Accounts Payable	Balance Sheet
...	...	...	...
84	txt	Total Income Taxes	Income Statement
85	xi	Extraordinary Items	Income Statement
86	xido	Extra + Discontinued	Income Statement
87	xidoc	Extraordinary Discontinued Operations	Income Statement
88	xint	Total Interest Expense	Income Statement

89 rows × 3 columns

2.2 Load raw data (no imputation or transformations)

In [61]:
```
data_path = Path(CONFIG["DATA_PATH"])
df_raw = pd.read_csv(data_path, low_memory=False)
print(f"Loaded data from {data_path} with shape {df_raw.shape}")

display(df_raw.head())
```

```
Loaded data from data.csv with shape (75005, 89)
```

	gvkey	datadate	fyear	indfmt	datafmt	consol	ismod	conm	aco	act	...	txach	txbcof	txdc	
0	1004	2015-05-31	2014	INDL	STD	C	1	AAR CORP	101.6	954.1	...	0.0	0.0	-79.8	1
1	1019	2014-12-31	2014	INDL	STD	C	1	AFA PROTECTIVE SYSTEMS INC	541.0	23369.0	...	NaN	0.0	-676.0	16
2	1045	2014-12-31	2014	INDL	STD	C	1	AMERICAN AIRLINES GROUP INC	1260.0	12112.0	...	0.0	0.0	346.0	
3	1050	2014-12-31	2014	INDL	STD	C	1	CECO ENVIRONMENTAL CORP	17424.0	142967.0	...	-1164.0	923.0	-3183.0	263
4	1072	2015-03-31	2014	INDL	STD	C	1	AVX CORP	108638.0	1744552.0	...	NaN	474.0	-58387.0	

5 rows × 89 columns

2.3 Enforce panel identifiers, types, sorting, and deduplication

In [62]:
```
df = df_raw.copy()

# Stable firm identifier
if "gvkey" not in df.columns:
    raise ValueError("Required identifier column `gvkey` not found in the dataset.")
df["firm_id"] = df["gvkey"].astype(str)

# Fiscal year
if "fyear" not in df.columns:
    raise ValueError("Required time column `fyear` not found in the dataset.")
df["fyear"] = pd.to_numeric(df["fyear"], errors="coerce").astype("Int64")

# Optional datadate parsing (kept as metadata; not used for splitting)
if "datadate" in df.columns:
    df["datadate"] = pd.to_datetime(df["datadate"], errors="coerce")

# Remove firm-year duplicates (keep-last rule, audit count)
pre_n = len(df)
dup_mask = df.duplicated(subset=["firm_id","fyear"], keep=False)
n_dups = int(dup_mask.sum())
if n_dups > 0:
    print(f"Found {n_dups} duplicated firm-year rows. Applying keep-last rule.")
    df = df.sort_values(["firm_id","fyear","datadate"] if "datadate" in df.columns else ["firm_id","fyear"])
    df = df.drop_duplicates(subset=["firm_id","fyear"], keep="last")
post_n = len(df)

# Enforce sort order for lag/lead safety
df = df.sort_values(["firm_id","fyear"]).reset_index(drop=True)

# Integrity checks
assert df[["firm_id","fyear"]].isna().sum().sum() == 0, "Missing firm_id or fyear after typing."
assert df.duplicated(subset=["firm_id","fyear"]).sum() == 0, "Duplicate firm-year keys remain after dedup."

print(f"Rows: {pre_n:,} -> {post_n:,} after deduplication.")
print("Unique firms:", df["firm_id"].nunique())
print("Year range:", int(df["fyear"].min()), "to", int(df["fyear"].max()))
```

```
Rows: 75,005 -> 75,005 after deduplication.
Unique firms: 11403
Year range: 2014 to 2024
```

2.4 Raw sample composition (no transformations)

In [63]:
```
# Minimal sample composition diagnostics (kept lightweight for large panels)

by_year = df.groupby("fyear").agg(
    n_obs=("firm_id","size"),
    n_firms=("firm_id","nunique"),
).reset_index()

display(by_year.tail(12))

# Optional: industry composition if SIC exists
if "sic" in df.columns:
    df["sic2"] = pd.to_numeric(df["sic"], errors="coerce").astype("Int64") // 100
    by_sic2 = df.groupby("sic2").size().sort_values(ascending=False).head(15).rename("n_obs").reset_index()
    display(by_sic2)
else:
    print("Note: `sic` not present; skipping industry composition.")
```

	fyear	n_obs	n_firms
0	2014	7455	7455
1	2015	7178	7178
2	2016	6970	6970
3	2017	6831	6831
4	2018	6672	6672
5	2019	6649	6649
6	2020	6703	6703
7	2021	6851	6851
8	2022	6848	6848
9	2023	6611	6611
10	2024	6237	6237

```
Note: `sic` not present; skipping industry composition.
```

3. Data Cleaning & Missingness Handling (leakage-aware)

3.1 Non-imputable identifiers and label-year setup

We drop observations missing non-imputable identifiers (firm, year).
We also define `label_year = fyear + 1` as the *outcome year* used for forecasting splits.

In [64]:
```python
# Drop rows with missing key identifiers (already asserted, but keep explicit)
df = df.dropna(subset=["firm_id","fyear"]).copy()

# label_year defines the year of the t+1 distress label
df["label_year"] = (df["fyear"] + 1).astype("Int64")

# Split masks (defined early; used for leakage-safe preprocessing throughout)
train_pool_mask = df["label_year"] <= CONFIG["TRAIN_CUTOFF_LABEL_YEAR"]
train_pool_years = sorted(df.loc[train_pool_mask, "label_year"].dropna().unique().tolist())
if len(train_pool_years) < (CONFIG["VAL_YEARS"] + 1):
    raise ValueError("Not enough label years in train pool to allocate validation years. Adjust TRAIN_CUTOFF_LA

val_years = train_pool_years[-CONFIG["VAL_YEARS"]:]
val_mask = df["label_year"].isin(val_years)
train_mask = train_pool_mask & (~val_mask)
test_mask = df["label_year"] > CONFIG["TRAIN_CUTOFF_LABEL_YEAR"]

df["split"] = np.where(test_mask, "test", np.where(val_mask, "val", "train"))

print("Split counts:")
display(df["split"].value_counts(dropna=False).to_frame("n_obs"))
print("Validation label_year(s):", val_years)
```

```
Split counts:
```

split	n_obs
train	48458
test	19696
val	6851

```
Validation label_year(s): [2022]
```

3.2 Missingness audit before intervention

In [65]:
```python
# Identify numeric columns eligible for imputation (exclude identifiers)
id_cols = {"gvkey","firm_id","fyear","label_year","datadate","split"}
numeric_cols = [c for c in df.columns if c not in id_cols and pd.api.types.is_numeric_dtype(df[c])]

missing_tbl = (df[numeric_cols].isna().mean().sort_values(ascending=False) * 100).rename("missing_%").to_frame(
missing_tbl["n_missing"] = df[numeric_cols].isna().sum().astype(int)
missing_tbl["dtype"] = [str(df[c].dtype) for c in missing_tbl.index]

display(missing_tbl.head(25))
```

	missing_%	n_missing	dtype
dlcch	44.187721	33143	float64
apalch	40.491967	30371	float64
txach	30.385974	22791	float64
ivstch	25.590294	19194	float64
recch	16.784214	12589	float64
mkvalt	16.465569	12350	float64
sppe	16.317579	12239	float64
act	14.293714	10721	float64
lct	14.259049	10695	float64
xint	14.047064	10536	float64
txditc	12.091194	9069	float64
txp	10.588627	7942	float64
esubc	9.063396	6798	float64
lco	9.038064	6779	float64
aco	9.036731	6778	float64
invch	7.086194	5315	float64
sppiv	5.288981	3967	float64
ivaeq	4.730351	3548	float64
prstkc	4.546364	3410	float64
caps	4.098393	3074	float64
aqc	3.749083	2812	float64
dp	3.629091	2722	float64
re	3.461103	2596	float64
ivch	3.386441	2540	float64
ivao	3.371775	2529	float64

3.3 Create missingness indicators (informative signals)

In [66]:
```
# Choose a focused set of inputs used for core ratios/events.
REQUIRED_RAW = [
    "at","dlc","dltt","seq","mibt","niadj",
    "oibdp","oancf","xint",
    "act","lct","che","rect","invt",
    # dividend-related (we will auto-detect among these later)
    "dv","dvc","dvt","dvp",
]
available_required = [c for c in REQUIRED_RAW if c in df.columns]

# Hard requirement for the distress proxy; fail if absent (unless synthetic mode)
HARD_REQUIRED = ["at","dlc","dltt","seq","oibdp","niadj","oancf"]
missing_hard = [c for c in HARD_REQUIRED if c not in df.columns]
if missing_hard and not USING_SYNTHETIC_DATA:
    raise ValueError(f"Missing required columns for distress proxy construction: {missing_hard}")

for c in available_required:
    df[f"fmiss_{c}"] = df[c].isna().astype("Int8")

print("Created missingness flags for:", available_required)
```

```
Created missingness flags for: ['at', 'dlc', 'dltt', 'seq', 'mibt', 'niadj', 'oibdp', 'oancf', 'xint', 'act', 'l
ct', 'che', 'rect', 'invt', 'dv', 'dvc', 'dvt', 'dvp']
```

3.4 Training-derived size deciles (used for peer imputation groups)

In [67]:
```
# Size is based on log(assets) from TRAIN only, to avoid leakage.
at_train = pd.to_numeric(df.loc[train_mask, "at"], errors="coerce")
log_at_train = np.log(at_train.where(at_train > 0)).dropna()

if len(log_at_train) < 50:
    print("WARNING: too few non-missing training `at` values for stable size deciles. Using a single size bin."
    df["size_decile"] = 5  # arbitrary mid-bin
    size_edges = None
else:
```

```
    # Use quantile cutpoints computed on training only
    qs = np.linspace(0, 1, 11)
    size_edges = log_at_train.quantile(qs).values
    size_edges[0] = -np.inf
    size_edges[-1] = np.inf

    log_at_all = np.log(pd.to_numeric(df["at"], errors="coerce").where(lambda s: s > 0))
    df["size_decile"] = pd.cut(log_at_all, bins=size_edges, labels=False, include_lowest=True).astype("Float64"

# Fill NA size_decile with training median decile for downstream stability
sd_med = float(pd.to_numeric(df.loc[train_mask, "size_decile"], errors="coerce").median())
df["size_decile"] = pd.to_numeric(df["size_decile"], errors="coerce").fillna(sd_med).astype(int)

print("Size decile distribution (train):")
display(df.loc[train_mask, "size_decile"].value_counts().sort_index().to_frame("n_obs"))
```

```
Size decile distribution (train):
```

size_decile	n_obs
0	4824
1	4823
2	4824
3	4823
4	5047
5	4823
6	4823
7	4824
8	4823
9	4824

3.5 Imputation Pipeline

In [68]:
```
# Snapshot before any imputation
df_pre_impute_snapshot = df.copy(deep=True)
```

3.5.1 KNN imputation on core structural items (train-fit; signed-log transform)

We use KNN for core balance sheet and income statement aggregates. These variables have strong multivariate dependencies (e.g., Total Assets ≈ Total Liabilities + Equity). KNN captures these relationships, allowing the imputation to respect the specific "profile" of a company.

In [69]:
```
# Core structural variables for KNN
knn_cols = [
    "at", "act", "lct", "che", "rect", "invt", "dlc", "dltt",
    "seq", "ceq", "lt", "ppent", "intan", "oibdp", "niadj",
    "oancf", "xint", "dp", "re", "capx"
]
knn_cols = [c for c in knn_cols if c in df.columns]

if len(knn_cols) >= 3:
    Z = df[knn_cols + ["fyear","size_decile"]].copy()
    # Transform magnitudes for distance stability
    for c in knn_cols:
        Z[c] = signed_log1p(Z[c])
    # Standardize fyear and size_decile to prevent distance domination by year scale (Issue 1)
    for c in ["fyear", "size_decile"]:
        Z[c] = pd.to_numeric(Z[c], errors="coerce")
        mu = Z.loc[train_mask, c].mean()
        sigma = Z.loc[train_mask, c].std()
        if sigma > 0:
            Z[c] = (Z[c] - mu) / sigma

    # --- KNN Imputation using training data ---
    imputer = KNNImputer(n_neighbors=CONFIG["KNN_K"], weights="distance")
    imputer.fit(Z.loc[train_mask, :])

    Z_imp = pd.DataFrame(imputer.transform(Z), columns=Z.columns, index=Z.index)

    # Invert signed-log transform back for magnitudes
    for c in knn_cols:
        # inverse of signed_log1p: sign(z)*(exp(|z|)-1)
        z = pd.to_numeric(Z_imp[c], errors="coerce")
```

```
        df[c] = np.sign(z) * (np.expm1(np.abs(z)))
else:
    print("Skipping KNN imputation: insufficient columns available.")
```

3.5.2 KNN Parameter Selection Audit

We evaluate the reconstruction quality for different values of K to justify the choice of `KNN_K=25`. We use a subset of fully observed training data and artificially introduce missingness to measure RMSE.

In [70]:
```
from sklearn.metrics import mean_squared_error

def evaluate_knn_k(Z_train, k_list, mask_fraction=0.1, seed=42):
    # Subset to fully observed rows for ground truth
    ground_truth_full = Z_train.dropna()
    if len(ground_truth_full) < 100:
        return None

    # Sample if too large for speed
    if len(ground_truth_full) > 2000:
        ground_truth = ground_truth_full.sample(n=2000, random_state=seed)
    else:
        ground_truth = ground_truth_full

    # Create masked version
    rng = np.random.default_rng(seed)
    masked_data = ground_truth.copy()

    # Only mask the core numeric columns (knn_cols)
    cols_to_mask = [c for c in ground_truth.columns if c not in ["fyear", "size_decile"]]

    for col in cols_to_mask:
        mask = rng.random(len(masked_data)) < mask_fraction
        masked_data.loc[mask, col] = np.nan

    results = []
    for k in k_list:
        imputer_test = KNNImputer(n_neighbors=k, weights="distance")
        # Fit on the original (possibly missing) training data
        imputer_test.fit(Z_train)
        # Transform the artificially masked data
        imputed_data = imputer_test.transform(masked_data)
        imputed_df = pd.DataFrame(imputed_data, columns=ground_truth.columns, index=ground_truth.index)

        # Calculate RMSE only on the values we masked
        mse = 0
        count = 0
        for col in cols_to_mask:
            actual_mask = masked_data[col].isna()
            if actual_mask.any():
                mse += mean_squared_error(ground_truth.loc[actual_mask, col], imputed_df.loc[actual_mask, col])
                count += actual_mask.sum()

        rmse = np.sqrt(mse / count) if count > 0 else np.nan
        results.append({"K": k, "RMSE": rmse})

    return pd.DataFrame(results)

if 'knn_cols' in locals() and len(knn_cols) >= 3:
    print("Evaluating KNN imputation performance (reconstruction RMSE)...")
    k_options = [5, 10, 25, 50, 100]
    knn_audit_df = evaluate_knn_k(Z.loc[train_mask, :], k_options)

    if knn_audit_df is not None:
        display(knn_audit_df.style.highlight_min(subset="RMSE", color="lightgreen"))

        k25_rmse = knn_audit_df.loc[knn_audit_df["K"] == 25, "RMSE"].values[0]
        best_k = knn_audit_df.loc[knn_audit_df["RMSE"].idxmin(), "K"]
        print(f"\nKNN K=25 RMSE: {k25_rmse:.4f}")
        if best_k == 25:
            print("K=25 is the optimal parameter among tested values.")
        else:
            print(f"Optimal K among tested is {best_k}. K=25 is used as a balanced choice.")
    else:
        print("Insufficient fully observed data for KNN audit.")
```

```
Evaluating KNN imputation performance (reconstruction RMSE)...
```

	K	RMSE
0	5	0.000001
1	10	0.000003
2	25	0.000006
3	50	0.000010
4	100	0.000019

```
KNN K=25 RMSE: 0.0000
Optimal K among tested is 5. K=25 is used as a balanced choice.
```

3.6 Train-only peer-median imputation (fyear × size_decile)

We use year-size median imputation for secondary items, "other" categories, and sparse flow variables (e.g., dividends, acquisitions). These variables are often missing, zero, or highly idiosyncratic. Using a multivariate model like KNN on them might introduce noise or over-impute non-zero values for sparse events. A year-size median provides a robust "typical" value for companies of similar scale in the same year, which is a safer conservative estimate for these items.

In [71]:
```python
# Secondary/incidental variables for Peer Median
peer_impute_candidates = [
    "aco", "lco", "recch", "invch", "txp", "txditc",
    "caps", "mibt", "aqc", "prstkc",
    "dv", "dvc", "dvt", "dvp"
]
peer_impute_cols = [c for c in peer_impute_candidates if c in df.columns]

group_cols = ["fyear","size_decile"]

def peer_median_impute(df_in: pd.DataFrame, cols: list[str], train_mask: pd.Series, group_cols: list[str]) -> tu
    """Impute NaNs using TRAIN-only medians by group_cols, with TRAIN (size_decile) then global median fallback
    df_out = df_in.copy()
    train = df_out.loc[train_mask, group_cols + cols].copy()
    group_meds = train.groupby(group_cols)[cols].median()
    global_meds = train[cols].median()

    # Intermediate fallback for unseen (fyear, size_decile): use TRAIN size_decile medians
    size_meds = train.groupby(["size_decile"])[cols].median()
    tmp_size = df_out[["size_decile"]].merge(size_meds.reset_index(), on="size_decile", how="left")

    # Join group medians (wide) to all rows
    tmp = df_out[group_cols].merge(group_meds.reset_index(), on=group_cols, how="left", suffixes=("", "_peer"))
    # tmp currently contains the group median columns with original names
    for c in cols:
        peer_med = tmp[c]
        df_out[c] = df_out[c].where(df_out[c].notna(), peer_med)
        size_med = tmp_size[c]
        df_out[c] = df_out[c].where(df_out[c].notna(), size_med)
        df_out[c] = df_out[c].where(df_out[c].notna(), global_meds[c])
    impact = pd.DataFrame({
        "col": cols,
        "n_imputed": [int(df_in[c].isna().sum() - df_out[c].isna().sum()) for c in cols],
        "train_global_median": [float(global_meds[c]) if pd.notna(global_meds[c]) else np.nan for c in cols],
    })
    return df_out, impact

df, peer_impact = peer_median_impute(df, peer_impute_cols, train_mask, group_cols)

display(peer_impact.sort_values("n_imputed", ascending=False).head(15))
```

	col	n_imputed	train_global_median
2	recch	12589	-11.0
5	txditc	9069	0.0
4	txp	7942	0.0
1	lco	6779	3415.5
0	aco	6778	766.0
3	invch	5315	0.0
9	prstkc	3410	0.0
6	caps	3074	21944.0
8	aqc	2812	0.0
10	dv	1137	0.0
7	mibt	842	0.0
12	dvt	706	0.0
11	dvc	703	0.0
13	dvp	36	0.0

3.7 Guardrail capping of imputed magnitudes (train quantile bands)

In [72]:
```
# Apply capping to all columns that underwent imputation (KNN and Peer Median)
cap_cols = list(set(knn_cols + peer_impute_cols))
bounds = {}

for c in cap_cols:
    lo, hi = winsorize_train_bounds(df_pre_impute_snapshot.loc[train_mask, c], CONFIG["IMPUTE_LO_Q"], CONFIG["I
    bounds[c] = {"lo": lo, "hi": hi}
    df[c] = apply_bounds(df[c], lo, hi)

bounds_df = pd.DataFrame({c: (v["lo"], v["hi"]) for c,v in bounds.items()}, index=["lo","hi"]).T
bounds_df.index.name = "col"
display(bounds_df.head(15))
```

col	lo	hi
prstkc	0.00	508786.45
rect	0.00	16780556.04
intan	0.00	7186093.64
lco	0.00	3325176.00
recch	-248055.64	105977.04
ceq	-286185.64	14146586.18
at	0.93	50746128.37
oibdp	-157336.02	4359988.88
aqc	-22209.00	589041.50
dlc	0.00	3229014.45
act	0.08	10216167.52
mibt	-1326.96	963289.56
txditc	0.00	833087.35
dvp	0.00	14797.38
dvt	0.00	821313.32

3.8 Imputation impact audit (pre vs post)

In [73]:
```
audit_cols = [c for c in ["at","dlc","dltt","seq","oibdp","oancf","act","lct"] if c in df.columns]

def dist_summary(x: pd.Series) -> dict:
    x = pd.to_numeric(x, errors="coerce")
    return {
        "n": int(x.notna().sum()),
        "mean": float(x.mean()) if x.notna().any() else np.nan,
        "p50": float(x.median()) if x.notna().any() else np.nan,
        "p10": float(x.quantile(0.10)) if x.notna().any() else np.nan,
```

```
        "p90": float(x.quantile(0.90)) if x.notna().any() else np.nan,
    }

rows = []
for c in audit_cols:
    pre = dist_summary(df_pre_impute_snapshot[c])
    post = dist_summary(df[c])
    rows.append({
        "col": c,
        "n_pre": pre["n"],
        "n_post": post["n"],
        "mean_pre": pre["mean"],
        "mean_post": post["mean"],
        "p50_pre": pre["p50"],
        "p50_post": post["p50"],
    })
impact_tbl = pd.DataFrame(rows).sort_values("col")
display(impact_tbl)
```

	col	n_pre	n_post	mean_pre	mean_post	p50_pre	p50_post
6	act	64284	75005	6.440956e+05	4.210849e+05	23074.0	31617.000000
0	at	75005	75005	5.597314e+06	2.060420e+06	96379.0	96379.000000
1	dlc	74974	75005	3.666371e+05	8.781058e+04	588.0	589.000000
2	dltt	74817	75005	9.226323e+05	4.592571e+05	3416.0	3374.000000
7	lct	64310	75005	4.755088e+05	2.892094e+05	11405.0	15890.235451
5	oancf	74702	75005	1.976674e+05	1.198129e+05	644.0	638.000000
4	oibdp	72863	75005	2.472644e+05	1.487529e+05	800.4	922.000000
3	seq	75002	75005	9.623007e+05	6.320878e+05	34787.5	34784.000000

4. Exploratory Data Analysis (EDA)

EDA focuses on **signal strength and data quality**, not exhaustive plotting.
At this stage we describe the imputed-but-not-modeled input space, by split.

4.1 Summary statistics by split (key magnitudes)

In [74]:
```
eda_cols = [c for c in ["at","dlc","dltt","seq","oibdp","oancf","xint"] if c in df.columns]

def split_describe(df_in: pd.DataFrame, cols: list[str]) -> pd.DataFrame:
    out = []
    for sp in ["train","val","test"]:
        d = df_in.loc[df_in["split"]==sp, cols].describe(percentiles=[0.01,0.1,0.5,0.9,0.99]).T
        d.insert(0, "split", sp)
        d.insert(1, "col", d.index)
        out.append(d.reset_index(drop=True))
    return pd.concat(out, ignore_index=True)

desc_tbl = split_describe(df, eda_cols)
display(desc_tbl.head(20))
```

	split	col	count	mean	std	min	1%	10%	50%	90%	99%
0	train	at	48458.0	1.958101e+06	6.596407e+06	0.93	0.9300	534.985	90113.000000	3790678.7	5.067805e+07
1	train	dlc	48458.0	8.543351e+04	3.915932e+05	0.00	0.0000	0.000	304.350000	92466.8	3.229014e+06
2	train	dltt	48458.0	4.453637e+05	1.581455e+06	0.00	0.0000	0.000	2146.900000	920105.2	1.178757e+07
3	train	seq	48458.0	5.974388e+05	1.953980e+06	-155487.96	-155258.2728	-383.790	33092.500000	1252878.4	1.487620e+07
4	train	oibdp	48458.0	1.417860e+05	5.548163e+05	-157336.02	-152927.2400	-12177.500	1097.068666	272939.8	4.252946e+06
5	train	oancf	48458.0	1.157459e+05	4.347615e+05	-136407.56	-135978.6000	-10121.600	778.000000	239939.4	3.324934e+06
6	train	xint	48458.0	2.038395e+04	6.339267e+04	0.00	0.0000	0.000	439.077471	49569.6	4.226626e+05
7	val	at	6851.0	2.165063e+06	6.845755e+06	0.93	1.0000	918.000	125582.000000	4515726.0	5.049778e+07
8	val	dlc	6851.0	8.542075e+04	3.905361e+05	0.00	0.0000	0.000	1027.000000	90986.0	3.164053e+06
9	val	dltt	6851.0	4.992172e+05	1.690081e+06	0.00	0.0000	0.000	6117.000000	1073364.0	1.179806e+07
10	val	seq	6851.0	7.102562e+05	2.141541e+06	-155487.96	-126339.0000	-2.380	55807.000000	1534804.0	1.490305e+07
11	val	oibdp	6851.0	1.580163e+05	6.138716e+05	-157336.02	-157336.0200	-31901.000	600.000000	320058.0	4.359989e+06
12	val	oancf	6851.0	1.219231e+05	4.706915e+05	-136407.56	-136407.5600	-29644.000	327.380000	255903.0	3.365800e+06
13	val	xint	6851.0	1.900839e+04	6.023077e+04	0.00	0.0000	0.000	470.584026	44698.0	3.897790e+05
14	test	at	19696.0	2.275757e+06	7.193903e+06	0.93	2.9685	1162.300	103152.500000	4657672.5	5.074613e+07
15	test	dlc	19696.0	9.449016e+04	3.961110e+05	0.00	0.0000	0.000	1518.000000	125157.5	2.962161e+06
16	test	dltt	19696.0	4.795392e+05	1.637765e+06	0.00	0.0000	0.000	6424.000000	1035513.0	1.179806e+07
17	test	seq	19696.0	6.901446e+05	2.146861e+06	-155487.96	-155487.9600	-363.000	33184.000000	1484313.0	1.490305e+07
18	test	oibdp	19696.0	1.626714e+05	6.180469e+05	-157336.02	-157336.0200	-31389.000	705.800000	342716.5	4.359989e+06
19	test	oancf	19696.0	1.290849e+05	4.742946e+05	-136407.56	-136407.5600	-25528.500	469.830000	285917.0	3.365800e+06

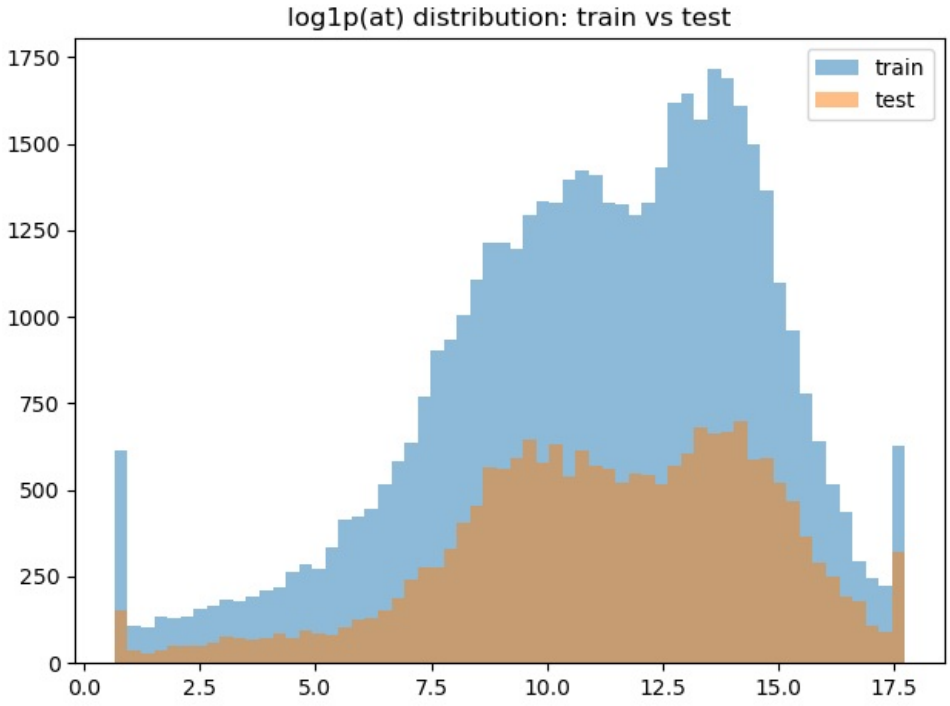
4.2 Missingness rates by split (key inputs)

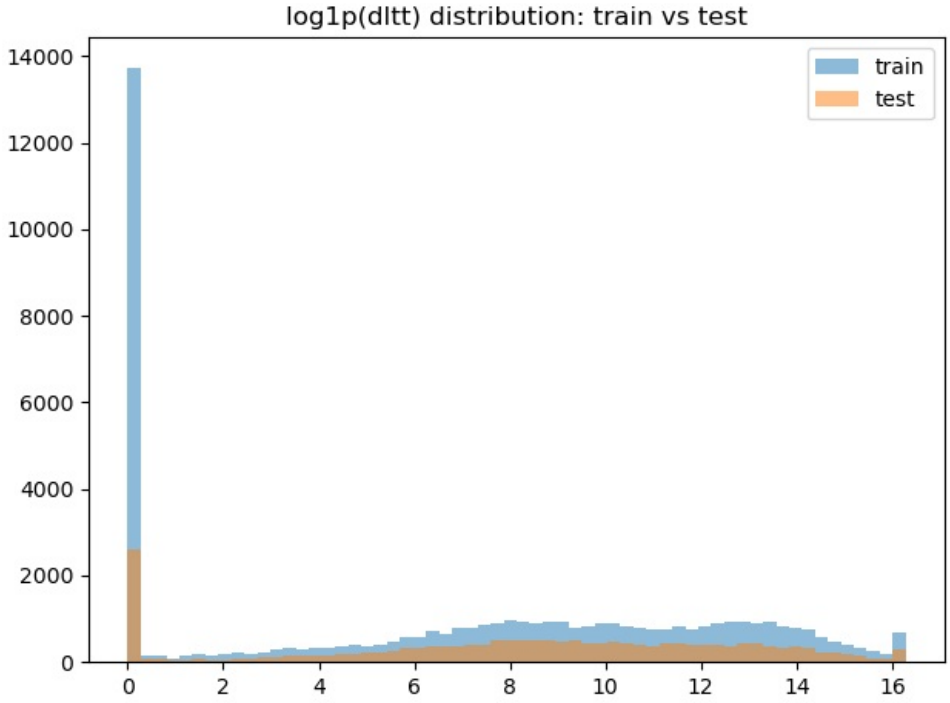
In [75]:
```
miss_cols = [c for c in available_required if f"fmiss_{c}" in df.columns]
miss_by_split = (
    df.groupby("split")[ [f"fmiss_{c}" for c in available_required if f"fmiss_{c}" in df.columns] ]
      .mean()
      .T
)
miss_by_split.index = [i.replace("fmiss_","") for i in miss_by_split.index]
miss_by_split = (miss_by_split * 100).round(2)
display(miss_by_split)
```

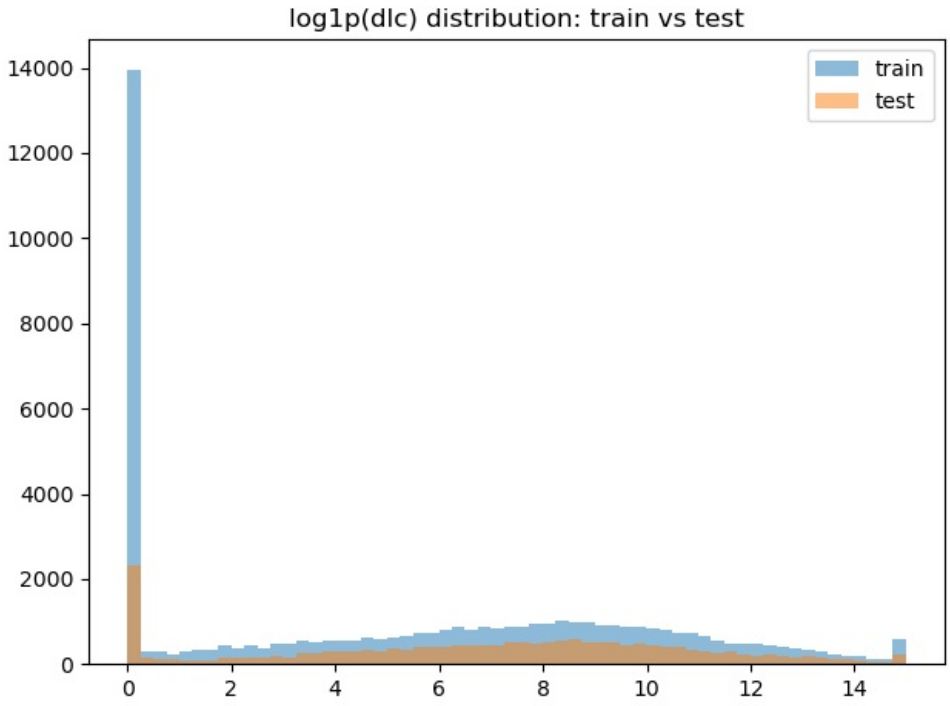
split	test	train	val
at	0.0	0.0	0.0
dlc	0.03	0.05	0.03
dltt	0.16	0.29	0.23
seq	0.0	0.01	0.0
mibt	0.76	1.3	0.89
niadj	0.0	0.0	0.0
oibdp	2.65	2.94	2.89
oancf	0.28	0.46	0.35
xint	14.57	13.92	13.43
act	13.62	14.66	13.66
lct	13.62	14.6	13.69
che	0.0	0.01	0.0
rect	1.26	1.12	1.26
invt	0.96	0.83	0.9
dv	1.36	1.59	1.45
dvc	0.55	1.14	0.66
dvt	0.55	1.14	0.66
dvp	0.05	0.05	0.04

4.3 Visual sanity-check plots (train vs test distributions)

In [76]:
```
# Lightweight plots to spot gross drift / outliers.
plot_cols = [c for c in ["at","dltt","dlc","oibdp","oancf"] if c in df.columns]

for c in plot_cols[:3]:
    a = pd.to_numeric(df.loc[df["split"]=="train", c], errors="coerce")
    b = pd.to_numeric(df.loc[df["split"]=="test", c], errors="coerce")
    plt.figure()
    plt.hist(np.log1p(a.dropna()), bins=60, alpha=0.5, label="train")
    plt.hist(np.log1p(b.dropna()), bins=60, alpha=0.5, label="test")
    plt.title(f"log1p({c}) distribution: train vs test")
    plt.legend()
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / f"eda_log1p_{c}_train_vs_test.png", dpi=140)
    plt.show()
```







5. Feature Engineering & Target Construction

This section constructs **all derived features explicitly** from Compustat-style raw items, including:

- debt aggregates and leverage ratios,
- cash-flow-to-debt ratios,
- log size and log market value,
- the NA-aware distress proxy and the next-year label.

Design choice: ratios with non-positive denominators are treated as **extreme tail states** (encoded via `+∞` then converted to `NaN` before modeling), rather than silently set to zero.

5.1 Feature list definitions (V1, V2, V3)

In [77]:
```
FEATURES_V1 = [
    "ln_at", "cash_at", "current_ratio", "nwc_at", "aco_act",
    "lco_lct", "rect_act", "invt_act", "recch_act", "invch_act",
    "txp_lct", "txditc_at", "lt_at", "dlc_at", "dltt_at",
    "debt_at", "st_debt_share", "ebitda_at", "dp_at",
    "xint_at", "interest_coverage", "debt_to_ebitda", "ebit_to_capital", "capx_at"
]

FEATURES_V2 = [
    "ln_at", "cash_at", "current_ratio", "nwc_at", "aco_act",
    "lco_lct", "rect_act", "invt_act", "ppent_at", "intan_at",
    "txp_lct", "txditc_at", "lt_at", "debt_at", "st_debt_share",
    "ebitda_at", "xint_at", "interest_coverage", "debt_to_ebitda",
    "ebit_to_capital", "ocf_to_debt", "fcf_to_debt", "capx_at", "re_at"
]

FEATURES_V3 = [
    "ln_at", "cash_at", "current_ratio", "nwc_at", "aco_act",
    "lco_lct", "rect_act", "invt_act", "recch_act", "invch_act",
    "txp_lct", "txditc_at", "lt_at", "ceq_at", "re_at",
    "caps_at", "mibt_at", "dp_at", "niadj_at", "loss_indicator",
    "xint_at", "xint_lct", "capx_at", "aqc_at", "prstkc_at"
]
```

5.2 Debt, capital, and operating aggregates

In [78]:
```
# Ensure all required raw items are numeric for safe arithmetic
raw_items = [
    "at", "che", "act", "lct", "aco", "lco", "rect", "invt", "recch", "invch",
    "txp", "txditc", "lt", "dlc", "dltt", "oibdp", "dp", "xint", "ceq", "capx",
    "ppent", "intan", "oancf", "re", "caps", "mibt", "niadj", "aqc", "prstkc", "seq"
]
for c in raw_items:
    if c in df.columns:
        df[c] = pd.to_numeric(df[c], errors="coerce")

# Debt aggregate
df["total_debt"] = df[["dlc","dltt"]].sum(axis=1, min_count=1)

# Equity plus minority interest (if available)
if "mibt" in df.columns:
    df["equity_plus_mi"] = df[["seq","mibt"]].sum(axis=1, min_count=1)
else:
    df["equity_plus_mi"] = df["seq"]

# Total capital and a non-positive capital flag
df["total_capital"] = df[["total_debt","equity_plus_mi"]].sum(axis=1, min_count=1)
df["cap_nonpos_flag"] = (df["total_capital"] <= 0).astype("Int8")

# EBITDA proxy
df["ebitda_proxy"] = df["oibdp"]
df["ebitda_nonpos_flag"] = (df["ebitda_proxy"] <= 0).astype("Int8")

# Log transforms
df["ln_at"] = np.log(df["at"].where(lambda s: s > 0))
# Legacy name if needed elsewhere
df["log_at"] = df["ln_at"]
```

5.3 Leverage, coverage, and cash-flow ratios (V1, V2, V3 features)

In [79]:
```
# --- V1/V2/V3 Shared & Specific Features ---
# (Using safe_divide which handles division by zero and returns NaN for extreme states)

# Basic Ratios
df["cash_at"] = safe_divide(df["che"], df["at"])
df["current_ratio"] = safe_divide(df["act"], df["lct"])
df["nwc_at"] = safe_divide(df["act"] - df["lct"], df["at"])
df["aco_act"] = safe_divide(df["aco"], df["act"])
df["lco_lct"] = safe_divide(df["lco"], df["lct"])
df["rect_act"] = safe_divide(df["rect"], df["act"])
df["invt_act"] = safe_divide(df["invt"], df["act"])
df["recch_act"] = safe_divide(df["recch"], df["act"])
df["invch_act"] = safe_divide(df["invch"], df["act"])
df["txp_lct"] = safe_divide(df["txp"], df["lct"])
df["txditc_at"] = safe_divide(df["txditc"], df["at"])
df["lt_at"] = safe_divide(df["lt"], df["at"])
df["dlc_at"] = safe_divide(df["dlc"], df["at"])
df["dltt_at"] = safe_divide(df["dltt"], df["at"])
df["debt_at"] = safe_divide(df["total_debt"], df["at"])
df["st_debt_share"] = safe_divide(df["dlc"], df["total_debt"])
df["ebitda_at"] = safe_divide(df["oibdp"], df["at"])
df["dp_at"] = safe_divide(df["dp"], df["at"])
df["xint_at"] = safe_divide(df["xint"], df["at"])
df["interest_coverage"] = safe_divide(df["oibdp"], df["xint"])
df["debt_to_ebitda"] = safe_divide(df["total_debt"], df["oibdp"])
df["ebit_to_capital"] = safe_divide(df["oibdp"] - df["dp"], df["total_debt"] + df["ceq"])
df["capx_at"] = safe_divide(df["capx"], df["at"])

# V2 extras
df["ppent_at"] = safe_divide(df["ppent"], df["at"])
df["intan_at"] = safe_divide(df["intan"], df["at"])
df["ocf_to_debt"] = safe_divide(df["oancf"], df["total_debt"])
df["fcf_to_debt"] = safe_divide(df["oancf"] - df["capx"], df["total_debt"])
df["re_at"] = safe_divide(df["re"], df["at"])

# V3 extras
df["ceq_at"] = safe_divide(df["ceq"], df["at"])
df["caps_at"] = safe_divide(df["caps"], df["at"])
df["mibt_at"] = safe_divide(df["mibt"], df["at"])
df["niadj_at"] = safe_divide(df["niadj"], df["at"])
df["loss_indicator"] = (df["niadj"] < 0).astype(float)
df["xint_lct"] = safe_divide(df["xint"], df["lct"])
df["aqc_at"] = safe_divide(df["aqc"], df["at"])
df["prstkc_at"] = safe_divide(df["prstkc"], df["at"])

# --- Legacy mappings for distress proxy definitions (Section 5.4) ---
# (Keeping sp_ prefix for variables used in distress proxy definition rules)
```

```
ffo_proxy = df["oancf"] + df["xint"]
if "txp" in df.columns:
    ffo_proxy = ffo_proxy - df["txp"]
df["sp_ffo_to_debt"] = safe_divide(ffo_proxy, df["total_debt"])
df["sp_debt_to_capital"] = safe_divide(df["total_debt"], df["total_capital"])
df["sp_debt_to_ebitda"] = df["debt_to_ebitda"]
df["sp_interest_coverage"] = df["interest_coverage"].clip(lower=-50, upper=50)

# Identify remaining +/-inf (though safe_divide already handles most)
ratio_cols = ["sp_debt_to_capital","sp_debt_to_ebitda","sp_ffo_to_debt","sp_interest_coverage"]
for c in ratio_cols:
    if c in df.columns:
        df[c] = df[c].replace([np.inf, -np.inf], np.nan)
```

5.4 Multiple Distress Proxies (fiscal year t) and next-year supervised labels (t+1)

In [80]:

```
# Distress proxy thresholds (frozen and documented)
DISTRESS_RULE = {
    "FFO_TO_DEBT_LT": 0.15,
    "DEBT_TO_CAPITAL_GT": 0.55,
    "DEBT_TO_EBITDA_GT": 4.5,
    "NEG_EQUITY_SEQ_LE": 0.0,
}

# --- Target construction from raw (non-imputed) data ---
# We compute distress proxies from the raw snapshot (df_pre_impute_snapshot)
# to ensure that target labels are not contaminated by the imputation process.
# Imputation is strictly reserved for predictive features.

raw_niadj = ensure_nullable_float(df_pre_impute_snapshot["niadj"])
raw_oancf = ensure_nullable_float(df_pre_impute_snapshot["oancf"])
raw_seq = ensure_nullable_float(df_pre_impute_snapshot["seq"])

# S&P components from raw items (propagate missingness - Issue 3)
raw_dlc = ensure_nullable_float(df_pre_impute_snapshot["dlc"])
raw_dltt = ensure_nullable_float(df_pre_impute_snapshot["dltt"])
raw_total_debt = raw_dlc + raw_dltt

raw_oibdp = ensure_nullable_float(df_pre_impute_snapshot["oibdp"])
raw_xint = ensure_nullable_float(df_pre_impute_snapshot["xint"])
raw_txp = ensure_nullable_float(df_pre_impute_snapshot["txp"]) if "txp" in df_pre_impute_snapshot.columns else (

raw_ffo = raw_oancf + raw_xint - raw_txp
raw_ffo_to_debt = safe_divide(raw_ffo, raw_total_debt)

raw_mibt = ensure_nullable_float(df_pre_impute_snapshot["mibt"]) if "mibt" in df_pre_impute_snapshot.columns els
raw_equity = ensure_nullable_float(df_pre_impute_snapshot[["seq","mibt"]].sum(axis=1, min_count=1)) if "mibt" i
raw_total_capital = raw_total_debt + raw_equity

raw_debt_to_cap = safe_divide(raw_total_debt, raw_total_capital)
raw_debt_to_ebitda = safe_divide(raw_total_debt, raw_oibdp)

# V1: Loss + NegCFO (Accounting-based)
# Beaver (1966), Ohlson (1980) logic: niadj < 0 and oancf < 0
df["distress_v1_t"] = (raw_niadj < 0) & (raw_oancf < 0)
# Fix: explicitly set to missing if inputs are missing
df.loc[raw_niadj.isna() | raw_oancf.isna(), "distress_v1_t"] = pd.NA

# V2: Negative Equity
df["distress_v2_t"] = raw_seq <= DISTRESS_RULE["NEG_EQUITY_SEQ_LE"]
# Fix: explicitly set to missing if inputs are missing
df.loc[raw_seq.isna(), "distress_v2_t"] = pd.NA

# V3: S&P High Leverage Solely (without conditioning on negative equity)
cond_ffo = raw_ffo_to_debt < DISTRESS_RULE["FFO_TO_DEBT_LT"]
cond_cap = raw_debt_to_cap > DISTRESS_RULE["DEBT_TO_CAPITAL_GT"]
cond_ebitda = raw_debt_to_ebitda > DISTRESS_RULE["DEBT_TO_EBITDA_GT"]
df["distress_v3_t"] = cond_ffo & cond_cap & cond_ebitda
# Fix: explicitly set to missing if inputs are missing
df.loc[raw_ffo_to_debt.isna() | raw_debt_to_cap.isna() | raw_debt_to_ebitda.isna(), "distress_v3_t"] = pd.NA

# Next-year targets: lead of proxies within firm
# Fix: Robust adjacency check (exactly fyear + 1) to avoid mislabeling gaps (Issue 1)
next_fyear = df.groupby("firm_id")["fyear"].shift(-1)
is_adjacent = (next_fyear == (df["fyear"] + 1))

df["target_next_v1"] = df.groupby("firm_id")["distress_v1_t"].shift(-1)
df.loc[~is_adjacent, "target_next_v1"] = pd.NA
df["target_next_v1"] = df["target_next_v1"].astype("Int8")

df["target_next_v2"] = df.groupby("firm_id")["distress_v2_t"].shift(-1)
df.loc[~is_adjacent, "target_next_v2"] = pd.NA
```

```
df["target_next_v2"] = df["target_next_v2"].astype("Int8")

df["target_next_v3"] = df.groupby("firm_id")["distress_v3_t"].shift(-1)
df.loc[~is_adjacent, "target_next_v3"] = pd.NA
df["target_next_v3"] = df["target_next_v3"].astype("Int8")

# Note: We keep v1, v2, v3 separate as requested and do not combine them.
# v2/target_next_v2 is used as the primary modeling proxy/target in the subsequent sections.
PROXY_NAME = "distress_v2_t"
TARGET_NAME = "target_next_v2"

# Label availability / attrition (fixed to check adjacency)
df["has_next_year_obs"] = is_adjacent.fillna(False).astype("Int8")

target_cols = ["target_next_v1", "target_next_v2", "target_next_v3"]
print("Distress prevalence (by split) — multiple targets:")
display(df.groupby("split")[target_cols].mean())

print("Share of observations with next-year observation (attrition diagnostic):")
display(df.groupby("split")["has_next_year_obs"].mean().rename("has_next_rate").to_frame())
```

```
Distress prevalence (by split) — multiple targets:
```

split	target_next_v1	target_next_v2	target_next_v3
test	0.333848	0.121023	0.061369
train	0.314055	0.12144	0.072889
val	0.372697	0.113658	0.067086

```
Share of observations with next-year observation (attrition diagnostic):
```

split	has_next_rate
test	0.627183
train	0.915308
val	0.929791

5.5 Target prevalence and attrition diagnostics (by year and size)

In [81]:

```
# Target prevalence by label year
target_cols = ["target_next_v1", "target_next_v2", "target_next_v3"]
agg_dict = {
    "n_obs": ("firm_id", "size"),
    "has_next_rate": ("has_next_year_obs", "mean"),
}
for c in target_cols:
    agg_dict[f"{c}_rate"] = (c, "mean")

by_label_year = df.groupby(["label_year","split"]).agg(**agg_dict).reset_index()

display(by_label_year.tail(15))

# By size decile (train pool), to assess composition effects
agg_dict_size = {"n_obs": ("firm_id", "size")}
for c in target_cols:
    agg_dict_size[f"{c}_rate"] = (c, "mean")

by_size = df.groupby(["size_decile","split"]).agg(**agg_dict_size).reset_index()

display(by_size.sort_values(["split","size_decile"]).head(30))
```

	label_year	split	n_obs	has_next_rate	target_next_v1_rate	target_next_v2_rate	target_next_v3_rate
0	2015	train	7455	0.900738	0.305734	0.135071	0.068298
1	2016	train	7178	0.905405	0.301609	0.130635	0.06846
2	2017	train	6970	0.915208	0.303875	0.118984	0.077986
3	2018	train	6831	0.915825	0.308717	0.12426	0.074519
4	2019	train	6672	0.916667	0.310809	0.126226	0.082594
5	2020	train	6649	0.927658	0.330677	0.116245	0.072257
6	2021	train	6703	0.928092	0.338496	0.097251	0.066652
7	2022	val	6851	0.929791	0.372697	0.113658	0.067086
8	2023	test	6848	0.916472	0.339779	0.119822	0.056125
9	2024	test	6611	0.919226	0.327723	0.122264	0.066845
10	2025	test	6237	0.0	<NA>	<NA>	<NA>

	size_decile	split	n_obs	target_next_v1_rate	target_next_v2_rate	target_next_v3_rate
0	0	test	1500	0.658455	0.397765	0.035385
3	1	test	1671	0.368522	0.222645	0.062183
6	2	test	2182	0.423503	0.140866	0.062977
9	3	test	2232	0.478545	0.126812	0.038925
12	4	test	2045	0.506998	0.141414	0.027344
15	5	test	1928	0.471854	0.087676	0.040161
18	6	test	1823	0.300441	0.070423	0.058427
21	7	test	1925	0.185913	0.052245	0.080132
24	8	test	2072	0.059542	0.047256	0.106667
27	9	test	2318	0.027443	0.029431	0.105856
1	0	train	4824	0.653564	0.399058	0.030826
4	1	train	4823	0.43062	0.249943	0.067511
7	2	train	4824	0.448456	0.137931	0.074203
10	3	train	4823	0.485558	0.091631	0.047093
13	4	train	5047	0.476149	0.115158	0.038877
16	5	train	4823	0.340682	0.073843	0.059382
19	6	train	4823	0.179664	0.048925	0.063952
22	7	train	4824	0.084944	0.042002	0.104371
25	8	train	4823	0.048856	0.047176	0.120606
28	9	train	4824	0.024601	0.027173	0.123229
2	0	val	555	0.689379	0.404	0.023747
5	1	val	600	0.409926	0.213504	0.054374
8	2	val	649	0.4375	0.147059	0.09205
11	3	val	652	0.47557	0.084142	0.058212
14	4	val	763	0.568182	0.143466	0.039568
17	5	val	735	0.521674	0.073244	0.043557
20	6	val	717	0.368585	0.052713	0.048096
23	7	val	676	0.237947	0.044961	0.087576
26	8	val	748	0.091038	0.042614	0.115063
29	9	val	756	0.053867	0.027586	0.110599

5.6 Event indicators (evt_*) for decision support

Events are discrete, interpretable signals designed for operational triage.
They are calibrated **using training data only** (when thresholds are estimated), and we explicitly **exclude** events mechanically tied to the distress-definition ratios (leverage/coverage) from the predictive feature set.

Events implemented here (subject to data availability):

- Dividend cut / suspension / initiation
- Liquidity squeeze (current ratio < 1.0) and quick-ratio squeeze (< 0.8)
- EBITDA drop (vs. t-1) and CFO drop (vs. t-1)

In [82]:
```
# Ensure sorting already enforced
assert df.index.is_monotonic_increasing

# Lag helpers
def lag(df_in: pd.DataFrame, col: str, n: int = 1) -> pd.Series:
    """Robust firm-level lag that enforces year adjacency (Issue 1)."""
    val = df_in.groupby("firm_id")[col].shift(n)
    year_lag = df_in.groupby("firm_id")["fyear"].shift(n)
    is_adjacent = (year_lag == (df_in["fyear"] - n))
    return val.where(is_adjacent.fillna(False), np.nan)

# Identify dividend column (prefer dvc if present; else dv / dvt / dvp)
dividend_candidates = ["dvc","dv","dvt","dvp"]
div_col = next((c for c in dividend_candidates if c in df.columns), None)

if div_col is None:
    print("Dividend column not found (looked for dvc/dv/dvt/dvp). Dividend events will be NaN.")
    df["evt_divcut"] = np.nan
    df["evt_divsusp"] = np.nan
    df["evt_divinit"] = np.nan
else:
    # Use absolute value (guard against sign conventions)
    df["dv_obs"] = pd.to_numeric(df[div_col], errors="coerce").abs()
    df["dv_obs_l1"] = lag(df, "dv_obs", 1)

# Liquidity ratios
if "act" in df.columns and "lct" in df.columns:
    df["current_ratio"] = safe_divide(df["act"], df["lct"], denom_floor=1e-6)
else:
    df["current_ratio"] = np.nan

if "act" in df.columns and "lct" in df.columns:
    if "invt" in df.columns:
        df["quick_ratio"] = safe_divide(pd.to_numeric(df["act"], errors="coerce") - pd.to_numeric(df["invt"], e
                                        df["lct"], denom_floor=1e-6)
    elif "che" in df.columns and "rect" in df.columns:
        df["quick_ratio"] = safe_divide(pd.to_numeric(df["che"], errors="coerce") + pd.to_numeric(df["rect"], e
                                        df["lct"], denom_floor=1e-6)
    else:
        df["quick_ratio"] = df["current_ratio"]
else:
    df["quick_ratio"] = np.nan

# EBITDA and CFO lags for deterioration events
if "ebitda_proxy" in df.columns:
    df["ebitda_l1"] = lag(df, "ebitda_proxy", 1)
if "oancf" in df.columns:
    df["cfo_l1"] = lag(df, "oancf", 1)
```

5.5.1 Dividend policy events (training-calibrated cut threshold)

In [83]:
```
event_params = {}

if div_col is None:
    pass
else:
    # YoY % change among observed payers with meaningful baseline
    dv_l1 = pd.to_numeric(df["dv_obs_l1"], errors="coerce")
    dv = pd.to_numeric(df["dv_obs"], errors="coerce")
    df["div_pct_change"] = np.where(dv_l1 > 1e-2, (dv - dv_l1) / dv_l1, np.nan)

    payer_train = train_mask & (dv_l1 > 0) & pd.notna(df["div_pct_change"])
    if payer_train.sum() >= 50:
        cut_thr = float(np.nanpercentile(df.loc[payer_train, "div_pct_change"], 10))
    else:
        cut_thr = -0.25

    # Bound cut threshold to avoid pathological values
    cut_thr = float(np.clip(cut_thr, -0.50, -0.10))
    event_params["DIV_CUT_THR_P10_BOUNDED"] = cut_thr

    # Dividend cut: large negative YoY change among payers
    df["evt_divcut"] = (df["div_pct_change"] <= cut_thr).astype("Int8")
    df.loc[df["div_pct_change"].isna(), "evt_divcut"] = pd.NA

    # Suspension: payer last year, ~zero dividend now
    df["evt_divsusp"] = ((dv_l1 > 0) & (dv.fillna(0) <= 1e-4)).astype("Int8")
```

```
    df.loc[dv_l1.isna() | dv.isna(), "evt_divsusp"] = pd.NA

    # Initiation: ~zero last year, dividend now positive
    df["evt_divinit"] = ((dv_l1.fillna(0) <= 1e-4) & (dv > 1e-4)).astype("Int8")
    df.loc[dv_l1.isna() | dv.isna(), "evt_divinit"] = pd.NA

    print(f"Dividend cut threshold (train P10 bounded): {cut_thr:.3f}")
    display(df[["dv_obs","dv_obs_l1","div_pct_change","evt_divcut","evt_divsusp","evt_divinit"]].head(8))
```

```
Dividend cut threshold (train P10 bounded): -0.500
```

	dv_obs	dv_obs_l1	div_pct_change	evt_divcut	evt_divsusp	evt_divinit
0	11905.0	NaN	NaN	<NA>	<NA>	<NA>
1	13697.0	11905.0	0.150525	0	0	0
2	15447.0	13697.0	0.127765	0	0	0
3	17287.0	15447.0	0.119117	0	0	0
4	18854.0	17287.0	0.090646	0	0	0
5	20593.0	18854.0	0.092235	0	0	0
6	11218.0	20593.0	-0.455252	0	0	0
7	22179.0	11218.0	0.977090	0	0	0

5.5.2 Liquidity squeeze events

In [84]:
```
cr = pd.to_numeric(df["current_ratio"], errors="coerce")
df["evt_liq_squeeze"] = (cr < 1.0).astype("Int8")
df.loc[cr.isna(), "evt_liq_squeeze"] = pd.NA

qr = pd.to_numeric(df["quick_ratio"], errors="coerce")
df["evt_quick_squeeze"] = (qr < 0.8).astype("Int8")
df.loc[qr.isna(), "evt_quick_squeeze"] = pd.NA

display(df[["current_ratio","quick_ratio","evt_liq_squeeze","evt_quick_squeeze"]].head(8))
```

	current_ratio	quick_ratio	evt_liq_squeeze	evt_quick_squeeze
0	1.857971	0.749443	0	1
1	0.001967	-1.169719	1	1
2	1.813861	0.712973	0	1
3	1.735209	0.593500	0	1
4	1.747436	0.628349	0	1
5	1.775434	0.585567	0	1
6	1.977964	0.836194	0	0
7	1.492971	0.508954	0	1

5.5.3 Operating deterioration events (vs. t-1)

In [85]:
```
# EBITDA drop: requires lagged EBITDA observed and positive
if "ebitda_proxy" in df.columns:
    e = pd.to_numeric(df["ebitda_proxy"], errors="coerce")
    e_l1 = pd.to_numeric(df["ebitda_l1"], errors="coerce")
    ratio = e / e_l1
    evt = ((e_l1 > 0) & ((ratio < 0.5) | (e <= 0))).astype("Int8")
    evt = evt.where(pd.notna(e_l1) & pd.notna(e), other=pd.NA).astype("Int8")
    df["evt_ebitdadrop"] = evt
else:
    df["evt_ebitdadrop"] = pd.NA

# CFO drop: requires lagged CFO observed and positive
if "oancf" in df.columns:
    c = pd.to_numeric(df["oancf"], errors="coerce")
    c_l1 = pd.to_numeric(df["cfo_l1"], errors="coerce")
    ratio = c / c_l1
    evt = ((c_l1 > 0) & ((ratio < 0.5) | (c <= 0))).astype("Int8")
    evt = evt.where(pd.notna(c_l1) & pd.notna(c), other=pd.NA).astype("Int8")
    df["evt_cfdrop"] = evt
else:
    df["evt_cfdrop"] = pd.NA

display(df[["ebitda_proxy","ebitda_l1","evt_ebitdadrop","oancf","cfo_l1","evt_cfdrop"]].head(10))
```

	ebitda_proxy	ebitda_l1	evt_ebitdadrop	oancf	cfo_l1	evt_cfdrop
0	113.4	NaN	<NA>	46987.00	NaN	<NA>
1	97361.0	113.4	0	65171.00	46987.0	0
2	121286.0	97361.0	0	97805.00	65171.0	0
3	126988.0	121286.0	0	64617.00	97805.0	0
4	105555.0	126988.0	0	70258.00	64617.0	0
5	122894.0	105555.0	0	76928.00	70258.0	0
6	138308.0	122894.0	0	97896.00	76928.0	0
7	158338.0	138308.0	0	85564.00	97896.0	0
8	141213.0	158338.0	0	-27533.00	85564.0	1
9	124265.0	141213.0	0	144.26	-27533.0	0

5.5.4 Event dictionary (appendix-ready)

In [86]:
```
event_dict_rows = [
    {"event":"evt_divcut", "definition":"Dividend YoY % change <= training P10 threshold (bounded [-0.50,-0.10]
    {"event":"evt_divsusp", "definition":"Dividend >0 at t-1 and ~0 at t", "inputs":div_col or "N/A", "calibrat
    {"event":"evt_divinit", "definition":"Dividend ~0 at t-1 and >0 at t", "inputs":div_col or "N/A", "calibrat
    {"event":"evt_liq_squeeze", "definition":"Current ratio < 1.0", "inputs":"act,lct", "calibration":"fixed th
    {"event":"evt_quick_squeeze", "definition":"Quick ratio < 0.8", "inputs":"act,lct,invt (or che+rect)", "cal
    {"event":"evt_ebitdadrop", "definition":"EBITDA <=0 OR EBITDA/EBITDA_{t-1}<0.5 (requires EBITDA_{t-1}>0)",
    {"event":"evt_cfdrop", "definition":"CFO <=0 OR CFO/CFO_{t-1}<0.5 (requires CFO_{t-1}>0)", "inputs":"oancf"
]
event_dict = pd.DataFrame(event_dict_rows)
event_dict["parameter"] = event_dict["event"].map(lambda e: json.dumps({k:v for k,v in event_params.items()}) i
display(event_dict)
```

	event	definition	inputs	calibration	parameter
0	evt_divcut	Dividend YoY % change <= training P10 threshol...	dvc	train-only	{"DIV_CUT_THR_P10_BOUNDED": -0.5}
1	evt_divsusp	Dividend >0 at t-1 and ~0 at t	dvc	none	
2	evt_divinit	Dividend ~0 at t-1 and >0 at t	dvc	none	
3	evt_liq_squeeze	Current ratio < 1.0	act,lct	fixed threshold	
4	evt_quick_squeeze	Quick ratio < 0.8	act,lct,invt (or che+rect)	fixed threshold	
5	evt_ebitdadrop	EBITDA <=0 OR EBITDA/EBITDA_{t-1}<0.5 (require...	oibdp	fixed threshold	
6	evt_cfdrop	CFO <=0 OR CFO/CFO_{t-1}<0.5 (requires CFO_{t-...	oancf	fixed threshold	

6. Preprocessing for Modeling (train-only fitting)

Preprocessing design principles:

- **Train-only fitting:** imputation (if needed), winsorization bounds, and scaling are all fit on *train* only.
- **Binary events stay in levels** (0/1) to preserve interpretability and prevalence.
- **Leakage audit:** variables that mechanically define the distress proxy are excluded from `MODEL_FEATS`.

6.1 Feature set definition and leakage audit

In [87]:
```
# Features that participate in the distress proxy definition (must be excluded from predictors)
# We use a dynamic set based on the chosen target to avoid definitional leakage (Issue 2).
if TARGET_NAME == "target_next_v1":
    # v1 uses niadj and oancf
    DISTRESS_DEFINITION_VARS = {"niadj", "oancf", "niadj_at", "loss_indicator", "ocf_to_debt", "fcf_to_debt"}
elif TARGET_NAME == "target_next_v2":
    # v2 uses seq
    DISTRESS_DEFINITION_VARS = {"seq"}
elif TARGET_NAME == "target_next_v3":
    # v3 uses ffo (oancf, xint, txp), debt (dlc, dltt), and equity (seq, mibt)
    DISTRESS_DEFINITION_VARS = {
        "sp_ffo_to_debt", "sp_debt_to_capital", "sp_debt_to_ebitda",
        "oancf", "xint", "txp", "dlc", "dltt", "seq", "mibt", "oibdp",
        "ocf_to_debt", "fcf_to_debt", "debt_to_ebitda", "interest_coverage"
    }
else:
```

```
    DISTRESS_DEFINITION_VARS = set()

# Candidate continuous predictors (selected based on TARGET_NAME)
if TARGET_NAME == "target_next_v1":
    continuous_feats_raw = [c for c in FEATURES_V1]
    event_feats = []
elif TARGET_NAME == "target_next_v2":
    continuous_feats_raw = [c for c in FEATURES_V2]
    event_feats = []
elif TARGET_NAME == "target_next_v3":
    # loss_indicator is binary, treat as event feature to avoid z-scoring
    continuous_feats_raw = [c for c in FEATURES_V3 if c != "loss_indicator"]
    event_feats = ["loss_indicator"]
else:
    continuous_feats_raw = [c for c in FEATURES_V2]
    event_feats = []

continuous_feats_raw = [c for c in continuous_feats_raw if c in df.columns]
event_feats = [c for c in event_feats if c in df.columns]

# Final model feature list (events in levels; continuous will be z-scored with z_ prefix)
MODEL_FEATS = [f"z_{c}" for c in continuous_feats_raw] + event_feats

# Leakage audit: ensure no distress-definition variables are included (raw or scaled variants)
leakage_hits = []
for v in DISTRESS_DEFINITION_VARS:
    if v in continuous_feats_raw or v in event_feats or f"z_{v}" in MODEL_FEATS:
        leakage_hits.append(v)

if leakage_hits:
    raise ValueError(f"Leakage audit failed: distress-definition variables present in feature set: {leakage_hits

print("Continuous (to be scaled):", continuous_feats_raw)
print("Events (kept in levels):", event_feats)
print("MODEL_FEATS (post-scaling names):", MODEL_FEATS)
```

```
Continuous (to be scaled): ['ln_at', 'cash_at', 'current_ratio', 'nwc_at', 'aco_act', 'lco_lct', 'rect_act', 'in
vt_act', 'ppent_at', 'intan_at', 'txp_lct', 'txditc_at', 'lt_at', 'debt_at', 'st_debt_share', 'ebitda_at', 'xint
_at', 'interest_coverage', 'debt_to_ebitda', 'ebit_to_capital', 'ocf_to_debt', 'fcf_to_debt', 'capx_at', 're_at'
]
Events (kept in levels): []
MODEL_FEATS (post-scaling names): ['z_ln_at', 'z_cash_at', 'z_current_ratio', 'z_nwc_at', 'z_aco_act', 'z_lco_lc
t', 'z_rect_act', 'z_invt_act', 'z_ppent_at', 'z_intan_at', 'z_txp_lct', 'z_txditc_at', 'z_lt_at', 'z_debt_at',
'z_st_debt_share', 'z_ebitda_at', 'z_xint_at', 'z_interest_coverage', 'z_debt_to_ebitda', 'z_ebit_to_capital', '
z_ocf_to_debt', 'z_fcf_to_debt', 'z_capx_at', 'z_re_at']
```

6.2 Modeling sample and target availability

In [88]:
```
# Modeling requires a defined next-year label
model_mask = df[TARGET_NAME].notna()
df_model = df.loc[model_mask].copy()

print("Modeling sample size:", df_model.shape[0])
display(df_model["split"].value_counts().to_frame("n_obs"))
```

```
Modeling sample size: 63074
```

split	n_obs
train	44351
test	12353
val	6370

6.3 Replace infinities and set up train-only median imputation for remaining NaNs

In [89]:
```
# Replace inf with NaN for preprocessing
for c in continuous_feats_raw:
    df_model[c] = pd.to_numeric(df_model[c], errors="coerce").replace([np.inf, -np.inf], np.nan)

# Train-only medians for remaining NaNs (after earlier imputation steps)
train_medians = df_model.loc[df_model["split"]=="train", continuous_feats_raw].median()

for c in continuous_feats_raw:
    df_model[c] = df_model[c].fillna(train_medians[c])

# Event features: coerce to Int8 with missing -> 0 (conservative) but preserve missingness flags separately if
for c in event_feats:
    df_model[c] = pd.to_numeric(df_model[c], errors="coerce").fillna(0).astype("Int8")
```

```
assert df_model[continuous_feats_raw].isna().sum().sum() == 0, "NaNs remain in continuous features after train-
```

6.4 Winsorize continuous features (train quantile bounds)

In [90]:
```
winsor_bounds = {}
for c in continuous_feats_raw:
    lo, hi = winsorize_train_bounds(df_model.loc[df_model["split"]=="train", c], CONFIG["WINSOR_LO_Q"], CONFIG[
    winsor_bounds[c] = (lo, hi)
    df_model[c] = apply_bounds(df_model[c], lo, hi)

winsor_tbl = pd.DataFrame(
    [{"feature": c, "lo": winsor_bounds[c][0], "hi": winsor_bounds[c][1]} for c in continuous_feats_raw]
)
display(winsor_tbl)
```

	feature	lo	hi
0	ln_at	0.000000	17.742346
1	cash_at	0.000000	555.021352
2	current_ratio	0.000390	5094.562108
3	nwc_at	-387.346049	560.393086
4	aco_act	0.000000	141.026630
5	lco_lct	0.000000	772.119898
6	rect_act	0.000000	523.621566
7	invt_act	0.000000	377.807774
8	ppent_at	0.000000	712.386243
9	intan_at	0.000000	352.718420
10	txp_lct	0.000000	27.841158
11	txditc_at	0.000000	29.321185
12	lt_at	0.000165	1036.470525
13	debt_at	0.000000	593.503174
14	st_debt_share	0.000000	1.000000
15	ebitda_at	-569.733871	145.517787
16	xint_at	0.000000	56.074527
17	interest_coverage	-13665.092655	20479.838612
18	debt_to_ebitda	-910.774112	5620.654806
19	ebit_to_capital	-93.249770	53.298370
20	ocf_to_debt	-938.666084	951.819000
21	fcf_to_debt	-1435.111111	633.217514
22	capx_at	0.000000	89.352771
23	re_at	-10519.661228	304.100091

6.5 Standardize continuous features (train-fit scaler; z_ prefix)

In [91]:
```
from sklearn.preprocessing import StandardScaler

# Standardize continuous features (fit on TRAIN only)
scaler = StandardScaler()
df_model[continuous_feats_raw] = df_model[continuous_feats_raw].apply(lambda s: pd.to_numeric(s, errors="coerce

train_cont = df_model.loc[df_model["split"] == "train", continuous_feats_raw].astype(float)
scaler.fit(train_cont)

Z_all = scaler.transform(df_model[continuous_feats_raw].astype(float))
for j, c in enumerate(continuous_feats_raw):
    df_model[f"z_{c}"] = Z_all[:, j].astype(float)

# Final modeling matrix (events forced to clean 0/1 ints)
z_cols = [f"z_{c}" for c in continuous_feats_raw]
X = df_model[z_cols + event_feats].copy()

for c in event_feats:
    X[c] = pd.to_numeric(X[c], errors="coerce")
    X[c] = X[c].fillna(0).astype("int8")
    assert set(X[c].unique()).issubset({0, 1}), f"{c} not binary after coercion: {sorted(X[c].unique())}"
```

```
y = df_model[TARGET_NAME].astype(int)

# Split views
splits = {}
for sp in ["train", "val", "test"]:
    mask = df_model["split"] == sp
    splits[sp] = {"X": X.loc[mask, :], "y": y.loc[mask], "df": df_model.loc[mask, :]}

print({sp: (v["X"].shape[0], v["X"].shape[1]) for sp, v in splits.items()})

# Numeric-safe finiteness check
assert np.isfinite(X.astype("float64").to_numpy()).all(), "Non-finite values in modeling matrix."
```

```
{'train': (44351, 24), 'val': (6370, 24), 'test': (12353, 24)}
```

7. Model Selection & Training

7A. Logit model (primary baseline: calibrated PD with interpretable coefficients)

7A.1 Hyperparameter tuning on out-of-time validation year

In [92]:
```
train_X, train_y = splits["train"]["X"], splits["train"]["y"]
val_X, val_y = splits["val"]["X"], splits["val"]["y"]

results = []
for C in CONFIG["LOGIT_C_GRID"]:
    mdl = LogisticRegression(C=C, solver="lbfgs", max_iter=2000, random_state=SEED)
    mdl.fit(train_X, train_y)
    val_proba = mdl.predict_proba(val_X)[:, 1]
    results.append({
        "C": C,
        "val_roc_auc": roc_auc_score(val_y, val_proba),
        "val_pr_auc": average_precision_score(val_y, val_proba),
        "val_brier": brier_score_loss(val_y, val_proba),
    })

tune_tbl = pd.DataFrame(results).sort_values("val_roc_auc", ascending=False)
display(tune_tbl)

best_C = float(tune_tbl.iloc[0]["C"])
print("Selected C:", best_C)
```

	C	val_roc_auc	val_pr_auc	val_brier
0	0.01	0.785159	0.505262	0.077200
1	0.10	0.785046	0.504543	0.077066
2	1.00	0.785037	0.504432	0.077054
3	10.00	0.785035	0.504396	0.077053

```
Selected C: 0.01
```

7A.2 Fit Logit models and generate PDs

In [93]:
```
trainval_mask = df_model["split"].isin(["train","val"])
X_trainval = X.loc[trainval_mask, :]
y_trainval = y.loc[trainval_mask]

# To ensure 'val' metrics are honest out-of-sample estimates, we use the model
# trained on 'train' only for the validation split.
# For the final 'test' performance, we use the model trained on 'train+val'.

# Model trained on 'train' ONLY (for honest 'val' evaluation)
logit_train_only = LogisticRegression(C=best_C, solver="lbfgs", max_iter=3000, random_state=SEED)
logit_train_only.fit(train_X, train_y)

# Model trained on 'train+val' (for final 'test' evaluation)
logit_trainval = LogisticRegression(C=best_C, solver="lbfgs", max_iter=3000, random_state=SEED)
logit_trainval.fit(X_trainval, y_trainval)

# Assign predictions
df_model["pd_logit"] = np.nan
# val gets predictions from train-only model (honest out-of-sample)
df_model.loc[df_model["split"]=="val", "pd_logit"] = logit_train_only.predict_proba(val_X)[:, 1]
# test gets predictions from train+val model
df_model.loc[df_model["split"]=="test", "pd_logit"] = logit_trainval.predict_proba(splits["test"]["X"])[:, 1]
# train gets predictions from train+val model (in-sample)
df_model.loc[df_model["split"]=="train", "pd_logit"] = logit_trainval.predict_proba(train_X)[:, 1]

# For legacy compatibility in reporting
df_model["pd_logit_val"] = np.where(df_model["split"]=="val", df_model["pd_logit"], np.nan)
```

```
df_model["pd_logit_test"] = np.where(df_model["split"]=="test", df_model["pd_logit"], np.nan)

# Keep logit_clf as the final model for downstream use
logit_clf = logit_trainval

print("Example PDs (Logit):")
display(df_model[["firm_id","fyear","label_year","split",TARGET_NAME,"pd_logit"]].head(10))
```

Example PDs (Logit):

	firm_id	fyear	label_year	split	target_next_v2	pd_logit
0	10000	2014	2015	train	0	0.073434
1	10000	2015	2016	train	0	0.087332
2	10000	2016	2017	train	0	0.071247
3	10000	2017	2018	train	0	0.070718
4	10000	2018	2019	train	0	0.069281
5	10000	2019	2020	train	0	0.079402
6	10000	2020	2021	train	0	0.062314
7	10000	2021	2022	val	0	0.055969
8	10000	2022	2023	test	0	0.036874
9	10000	2023	2024	test	0	0.033127

7A.3 Inference audit (statsmodels Logit; clustered standard errors)

In [94]:

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
from scipy import stats
from statsmodels.stats.sandwich_covariance import cov_cluster, cov_cluster_2groups
# Statsmodels requires numpy arrays; keep column names for tables.
X_sm = sm.add_constant(X_trainval, has_constant="add")
y_sm = y_trainval.astype(float)

logit_sm = sm.Logit(y_sm, X_sm)
res_sm = logit_sm.fit(disp=False, maxiter=200)

# --- Firm cluster (numeric codes to avoid dtype issues) ---
firm_raw = df_model.loc[trainval_mask, "firm_id"]
firm_codes = pd.factorize(firm_raw, sort=True)[0].astype(np.int64)

cov_firm = cov_cluster(res_sm, firm_codes)
se_firm = np.sqrt(np.diag(cov_firm))

# --- Two-way cluster (firm + year), with feasibility + shape guards ---
year_raw = df_model.loc[trainval_mask, "label_year"]
firm_raw = df_model.loc[trainval_mask, "firm_id"]

firm_codes = pd.factorize(firm_raw, sort=True)[0].astype(np.int64)
year_codes = pd.factorize(year_raw, sort=True)[0].astype(np.int64)

if (np.unique(firm_codes).size < 2) or (np.unique(year_codes).size < 2):
    # Not enough clusters in one dimension -> two-way clustering not identified
    se_2 = se_firm.copy()
else:
    cov_2 = cov_cluster_2groups(res_sm, firm_codes, year_codes)
    cov_2 = np.asarray(cov_2)

    k = len(res_sm.params)
    if cov_2.ndim == 2 and cov_2.shape == (k, k):
        se_2 = np.sqrt(np.diag(cov_2))
    elif cov_2.ndim == 1 and cov_2.size == k:
        # Some statsmodels versions may return only the diagonal variances
        se_2 = np.sqrt(cov_2)
    else:
        # Unexpected shape -> fall back (safer than crashing)
        se_2 = se_firm.copy()

coef = res_sm.params
p_firm = 2 * (1 - stats.norm.cdf(np.abs(coef / se_firm)))
p_2 = 2 * (1 - stats.norm.cdf(np.abs(coef / se_2)))

infer_tbl = pd.DataFrame({
    "coef_logodds": coef,
    "se_firm": se_firm,
    "p_firm": p_firm,
    "se_firm_year": se_2,
```

```
    "p_firm_year": p_2,
    "odds_ratio": np.exp(coef),
})
infer_tbl.index.name = "feature"
display(infer_tbl)
```

feature	coef_logodds	se_firm	p_firm	se_firm_year	p_firm_year	odds_ratio
const	-2.493923	0.038545	0.000000e+00	0.038545	0.000000e+00	0.082585
z_ln_at	-1.175934	0.036151	0.000000e+00	0.036151	0.000000e+00	0.308531
z_cash_at	-0.271302	0.026392	0.000000e+00	0.026392	0.000000e+00	0.762386
z_current_ratio	-0.296104	0.030688	0.000000e+00	0.030688	0.000000e+00	0.743710
z_nwc_at	-0.227721	0.020300	0.000000e+00	0.020300	0.000000e+00	0.796346
z_aco_act	0.121639	0.017884	1.034706e-11	0.017884	1.034706e-11	1.129346
z_lco_lct	0.260041	0.021422	0.000000e+00	0.021422	0.000000e+00	1.296984
z_rect_act	-0.036731	0.022571	1.036606e-01	0.022571	1.036606e-01	0.963935
z_invt_act	-0.047676	0.021964	2.996086e-02	0.021964	2.996086e-02	0.953443
z_ppent_at	-0.266579	0.024454	0.000000e+00	0.024454	0.000000e+00	0.765995
z_intan_at	-0.197935	0.023464	0.000000e+00	0.023464	0.000000e+00	0.820423
z_txp_lct	-0.017683	0.024395	4.685243e-01	0.024395	4.685243e-01	0.982472
z_txditc_at	-0.103990	0.033113	1.687135e-03	0.033113	1.687135e-03	0.901235
z_lt_at	-0.150641	0.024018	3.566452e-10	0.024018	3.566452e-10	0.860157
z_debt_at	0.058223	0.030312	5.476090e-02	0.030312	5.476090e-02	1.059951
z_st_debt_share	0.253436	0.020739	0.000000e+00	0.020739	0.000000e+00	1.288444
z_ebitda_at	0.052713	0.023593	2.546222e-02	0.023593	2.546222e-02	1.054127
z_xint_at	0.221559	0.030137	1.956213e-13	0.030137	1.956213e-13	1.248021
z_interest_coverage	-0.097633	0.017619	3.001759e-08	0.017619	3.001759e-08	0.906982
z_debt_to_ebitda	-0.008094	0.026848	7.630537e-01	0.026848	7.630537e-01	0.991939
z_ebit_to_capital	0.077400	0.016009	1.331999e-06	0.016009	1.331999e-06	1.080474
z_ocf_to_debt	-0.270714	0.071701	1.596115e-04	0.071701	1.596115e-04	0.762834
z_fcf_to_debt	0.207613	0.062332	8.661457e-04	0.062332	8.661457e-04	1.230736
z_capx_at	-0.039080	0.026643	1.424279e-01	0.026643	1.424279e-01	0.961674
z_re_at	-0.057848	0.022355	9.661593e-03	0.022355	9.661593e-03	0.943794

7A.4 Economic magnitude: MEM marginal effects and IQR-scaled effects

In [95]:
```
# MEM marginal effects using statsmodels (on train+val)
try:
    me = res_sm.get_margeff(at="mean")
    me_tbl = me.summary_frame()
    # Align to inference table index (margeff typically excludes const)
    me_tbl = me_tbl.reindex(infer_tbl.index)
    display(me_tbl)
except Exception as e:
    print("Marginal effects computation failed:", e)
    me_tbl = None

# IQR-scaled effects for continuous features (using TRAIN distribution, mapped into z-space)
train_df = df_model.loc[df_model["split"]=="train", :].copy()

iqr_rows = []
for j, raw_c in enumerate(continuous_feats_raw):
    q25 = float(train_df[raw_c].quantile(0.25))
    q75 = float(train_df[raw_c].quantile(0.75))
    iqr = q75 - q25
    sd = float(scaler.scale_[j]) if scaler.scale_[j] > 0 else np.nan
    delta_z = iqr / sd if sd and not np.isnan(sd) else np.nan
    beta = float(infer_tbl.loc[f"z_{raw_c}", "coef_logodds"]) if f"z_{raw_c}" in infer_tbl.index else np.nan
    logodds_delta = beta * delta_z if not np.isnan(beta) and not np.isnan(delta_z) else np.nan
    iqr_rows.append({
        "raw_feature": raw_c,
        "IQR_raw": iqr,
        "delta_z_equiv": delta_z,
        "logodds_change_IQR": logodds_delta,
        "odds_ratio_IQR": float(np.exp(logodds_delta)) if not np.isnan(logodds_delta) else np.nan,
```

```
    })

iqr_tbl = pd.DataFrame(iqr_rows)
display(iqr_tbl)
```

feature	dy/dx	Std. Err.	z	Pr(>|z|)	Conf. Int. Low	Cont. Int. Hi.
const	NaN	NaN	NaN	NaN	NaN	NaN
z_ln_at	-0.081905	0.001397	-58.620606	0.000000e+00	-0.084643	-0.079166
z_cash_at	-0.018896	0.001610	-11.737660	8.171584e-32	-0.022052	-0.015741
z_current_ratio	-0.020624	0.002025	-10.184803	2.318293e-24	-0.024593	-0.016655
z_nwc_at	-0.015861	0.001332	-11.909616	1.054620e-32	-0.018471	-0.013251
z_aco_act	0.008472	0.001124	7.540346	4.687262e-14	0.006270	0.010674
z_lco_lct	0.018112	0.001349	13.424494	4.345417e-41	0.015468	0.020756
z_rect_act	-0.002558	0.001392	-1.838318	6.601563e-02	-0.005286	0.000169
z_invt_act	-0.003321	0.001321	-2.513136	1.196632e-02	-0.005910	-0.000731
z_ppent_at	-0.018567	0.001631	-11.382756	5.096309e-30	-0.021765	-0.015370
z_intan_at	-0.013786	0.001464	-9.418288	4.585010e-21	-0.016655	-0.010917
z_txp_lct	-0.001232	0.001524	-0.808153	4.190026e-01	-0.004219	0.001755
z_txditc_at	-0.007243	0.001835	-3.947860	7.885293e-05	-0.010839	-0.003647
z_lt_at	-0.010492	0.001631	-6.432845	1.252374e-10	-0.013689	-0.007295
z_debt_at	0.004055	0.001757	2.307956	2.100157e-02	0.000611	0.007499
z_st_debt_share	0.017652	0.001008	17.512644	1.147378e-68	0.015676	0.019628
z_ebitda_at	0.003672	0.001341	2.738586	6.170393e-03	0.001044	0.006299
z_xint_at	0.015432	0.001597	9.660337	4.443983e-22	0.012301	0.018563
z_interest_coverage	-0.006800	0.001288	-5.280701	1.286907e-07	-0.009324	-0.004276
z_debt_to_ebitda	-0.000564	0.001419	-0.397226	6.912005e-01	-0.003345	0.002218
z_ebit_to_capital	0.005391	0.001038	5.193738	2.061128e-07	0.003357	0.007425
z_ocf_to_debt	-0.018855	0.003557	-5.300908	1.152279e-07	-0.025827	-0.011884
z_fcf_to_debt	0.014460	0.003083	4.689869	2.733796e-06	0.008417	0.020504
z_capx_at	-0.002722	0.001669	-1.631305	1.028261e-01	-0.005992	0.000548
z_re_at	-0.004029	0.001336	-3.015717	2.563725e-03	-0.006648	-0.001411

	raw_feature	IQR_raw	delta_z_equiv	logodds_change_IQR	odds_ratio_IQR
0	ln_at	4.932657	1.372432	-1.613890	0.199112
1	cash_at	0.343684	0.004668	-0.001267	0.998734
2	current_ratio	2.568065	0.003674	-0.001088	0.998913
3	nwc_at	0.381575	0.004176	-0.000951	0.999049
4	aco_act	0.122986	0.006400	0.000779	1.000779
5	lco_lct	0.607599	0.004910	0.001277	1.001278
6	rect_act	0.556480	0.007340	-0.000270	0.999730
7	invt_act	0.269088	0.005438	-0.000259	0.999741
8	ppent_at	0.464446	0.004902	-0.001307	0.998694
9	intan_at	0.237122	0.005352	-0.001059	0.998941
10	txp_lct	0.008860	0.002810	-0.000050	0.999950
11	txditc_at	0.013609	0.004013	-0.000417	0.999583
12	lt_at	0.613917	0.003391	-0.000511	0.999489
13	debt_at	0.435036	0.005100	0.000297	1.000297
14	st_debt_share	0.425525	1.191924	0.302076	1.352664
15	ebitda_at	0.199426	0.002807	0.000148	1.000148
16	xint_at	0.025467	0.003559	0.000788	1.000789
17	interest_coverage	13.766536	0.004660	-0.000455	0.999545
18	debt_to_ebitda	3.551778	0.004789	-0.000039	0.999961
19	ebit_to_capital	0.232939	0.019608	0.001518	1.001519
20	ocf_to_debt	0.341191	0.002176	-0.000589	0.999411
21	fcf_to_debt	0.342544	0.001878	0.000390	1.000390
22	capx_at	0.053429	0.004608	-0.000180	0.999820
23	re_at	1.454782	0.001189	-0.000069	0.999931

7A.5 Average Partial Effects (APEs) in probability units with cluster-robust uncertainty

In [96]:

```
# APEs (Average Partial Effects) for logit model, using delta-method SEs with cluster-robust covariances
# Notes:
# - For logit, dP/dx_j = p*(1-p)*beta_j. The APE is the sample average of this derivative.
# - We compute APEs on the TRAIN+VAL estimation sample used in statsmodels (X_sm, res_sm).
# - SEs use the same cluster-robust covariance matrices already computed above (cov_firm and cov_2).

import numpy as np
import pandas as pd
from scipy import stats
def _coerce_cov(V, names):
    """Return numeric (k x k) covariance aligned to names. Fallback logic handles common statsmodels outputs.""
    k = len(names)

    if isinstance(V, pd.DataFrame):
        V = V.reindex(index=names, columns=names).to_numpy(dtype=float)
        return V

    V = np.asarray(V)
    V = np.squeeze(V)

    # Handle 3D objects (e.g., (3,k,k) or (k,k,3)): take first covariance slice
    if V.ndim == 3:
        if V.shape[1:] == (k, k):
            V = V[0]
        elif V.shape[:2] == (k, k):
            V = V[:, :, 0]
        else:
            V = V.reshape(-1, k, k)[0]

    # Handle diagonal-only variances
    if V.ndim == 1:
        if V.size != k:
            raise ValueError(f"Unexpected 1D covariance length: {V.size} (expected {k})")
        V = np.diag(V.astype(float))

    if V.ndim != 2 or V.shape != (k, k):
        raise ValueError(f"Unexpected covariance shape: {V.shape} (expected {(k, k)})")
```

```
    V = V.astype(float)
    V[~np.isfinite(V)] = 0.0
    return V

# ---- Align design matrix to params order ----
X_df = X_sm if isinstance(X_sm, pd.DataFrame) else pd.DataFrame(X_sm)
b_ser = res_sm.params

names = list(b_ser.index)
X_df = X_df.loc[:, names]                 # enforce same column order
X_audit_np = X_df.to_numpy(dtype=float)

b = b_ser.to_numpy(dtype=float)
k = len(names)

# Predicted probabilities on estimation sample
eta = X_audit_np @ b
p = 1.0 / (1.0 + np.exp(-eta))
w = p * (1.0 - p)
mw = float(np.mean(w))

# APE_j = beta_j * mean(w)
ape = b * mw
if "const" in names:
    ape[names.index("const")] = np.nan

# Delta-method gradient
t = (w * (1.0 - 2.0 * p))[:, None] * X_audit_np
dmw_db = np.mean(t, axis=0)

G = np.full((k, k), np.nan, dtype=float)
for j in range(k):
    if names[j] == "const":
        continue
    g = dmw_db * b[j]
    g[j] += mw
    G[j, :] = g

# Covariances (coerce 2-way; fallback to firm)
V_firm = _coerce_cov(cov_firm, names)
if "cov_2" in globals():
    try:
        V_2 = _coerce_cov(cov_2, names)
    except Exception:
        V_2 = V_firm
else:
    V_2 = V_firm

def _se_from_V(V):
    se = np.full(k, np.nan, dtype=float)
    for j in range(k):
        if not np.all(np.isfinite(G[j, :])):
            continue
        g = G[j, :].astype(float)
        v = (g @ V @ g).item()  # scalar quadratic form
        se[j] = np.sqrt(v) if v >= 0 else np.nan
    return se

se_ape_firm = _se_from_V(V_firm)
se_ape_2 = _se_from_V(V_2)

# p-values (normal approximation)
z_firm = ape / se_ape_firm
p_ape_firm = 2 * (1 - stats.norm.cdf(np.abs(z_firm)))

z_2 = ape / se_ape_2
p_ape_2 = 2 * (1 - stats.norm.cdf(np.abs(z_2)))

ape_tbl = pd.DataFrame({
    "APE": ape,
    "se_APE_firm": se_ape_firm,
    "p_APE_firm": p_ape_firm,
    "se_APE_firm_year": se_ape_2,
    "p_APE_firm_year": p_ape_2,
}, index=pd.Index(names, name="feature"))

display(ape_tbl)

infer_tbl_with_ape = infer_tbl.join(ape_tbl, how="left")
display(infer_tbl_with_ape)
```

feature	APE	se_APE_firm	p_APE_firm	se_APE_firm_year	p_APE_firm_year
const	NaN	NaN	NaN	NaN	NaN
z_ln_at	-0.096864	0.002571	0.000000e+00	0.004060	0.000000e+00
z_cash_at	-0.022348	0.002165	0.000000e+00	0.002227	0.000000e+00
z_current_ratio	-0.024391	0.002569	0.000000e+00	0.002808	0.000000e+00
z_nwc_at	-0.018758	0.001687	0.000000e+00	0.002092	0.000000e+00
z_aco_act	0.010020	0.001485	1.510791e-11	0.001484	1.448019e-11
z_lco_lct	0.021420	0.001801	0.000000e+00	0.001507	0.000000e+00
z_rect_act	-0.003026	0.001859	1.036772e-01	0.001954	1.214788e-01
z_invt_act	-0.003927	0.001813	3.026096e-02	0.001764	2.600688e-02
z_ppent_at	-0.021959	0.002023	0.000000e+00	0.001931	0.000000e+00
z_intan_at	-0.016304	0.001924	0.000000e+00	0.001414	0.000000e+00
z_txp_lct	-0.001457	0.002010	4.685677e-01	0.002844	6.084895e-01
z_txditc_at	-0.008566	0.002713	1.589990e-03	0.002004	1.909269e-05
z_lt_at	-0.012409	0.001961	2.494838e-10	0.002315	8.337291e-08
z_debt_at	0.004796	0.002501	5.517965e-02	0.002829	8.997608e-02
z_st_debt_share	0.020876	0.001635	0.000000e+00	0.001412	0.000000e+00
z_ebitda_at	0.004342	0.001948	2.579986e-02	0.002233	5.185844e-02
z_xint_at	0.018250	0.002510	3.577139e-13	0.002189	0.000000e+00
z_interest_coverage	-0.008042	0.001450	2.900621e-08	0.002117	1.453192e-04
z_debt_to_ebitda	-0.000667	0.002210	7.629281e-01	0.002613	7.986203e-01
z_ebit_to_capital	0.006376	0.001327	1.539101e-06	0.001213	1.484618e-07
z_ocf_to_debt	-0.022299	0.005908	1.603525e-04	0.005496	4.967616e-05
z_fcf_to_debt	0.017102	0.005135	8.679309e-04	0.005090	7.795196e-04
z_capx_at	-0.003219	0.002194	1.422590e-01	0.001653	5.143419e-02
z_re_at	-0.004765	0.001846	9.859352e-03	0.002288	3.725700e-02

feature	coef_logodds	se_firm	p_firm	se_firm_year	p_firm_year	odds_ratio	APE	se_APE_firm	p_APE
const	-2.493923	0.038545	0.000000e+00	0.038545	0.000000e+00	0.082585	NaN	NaN	
z_ln_at	-1.175934	0.036151	0.000000e+00	0.036151	0.000000e+00	0.308531	-0.096864	0.002571	0.000000
z_cash_at	-0.271302	0.026392	0.000000e+00	0.026392	0.000000e+00	0.762386	-0.022348	0.002165	0.000000
z_current_ratio	-0.296104	0.030688	0.000000e+00	0.030688	0.000000e+00	0.743710	-0.024391	0.002569	0.000000
z_nwc_at	-0.227721	0.020300	0.000000e+00	0.020300	0.000000e+00	0.796346	-0.018758	0.001687	0.000000
z_aco_act	0.121639	0.017884	1.034706e-11	0.017884	1.034706e-11	1.129346	0.010020	0.001485	1.51079
z_lco_lct	0.260041	0.021422	0.000000e+00	0.021422	0.000000e+00	1.296984	0.021420	0.001801	0.000000
z_rect_act	-0.036731	0.022571	1.036606e-01	0.022571	1.036606e-01	0.963935	-0.003026	0.001859	1.03677
z_invt_act	-0.047676	0.021964	2.996086e-02	0.021964	2.996086e-02	0.953443	-0.003927	0.001813	3.02609
z_ppent_at	-0.266579	0.024454	0.000000e+00	0.024454	0.000000e+00	0.765995	-0.021959	0.002023	0.000000
z_intan_at	-0.197935	0.023464	0.000000e+00	0.023464	0.000000e+00	0.820423	-0.016304	0.001924	0.000000
z_txp_lct	-0.017683	0.024395	4.685243e-01	0.024395	4.685243e-01	0.982472	-0.001457	0.002010	4.68567
z_txditc_at	-0.103990	0.033113	1.687135e-03	0.033113	1.687135e-03	0.901235	-0.008566	0.002713	1.58999
z_lt_at	-0.150641	0.024018	3.566452e-10	0.024018	3.566452e-10	0.860157	-0.012409	0.001961	2.49483
z_debt_at	0.058223	0.030312	5.476090e-02	0.030312	5.476090e-02	1.059951	0.004796	0.002501	5.51796
z_st_debt_share	0.253436	0.020739	0.000000e+00	0.020739	0.000000e+00	1.288444	0.020876	0.001635	0.000000
z_ebitda_at	0.052713	0.023593	2.546222e-02	0.023593	2.546222e-02	1.054127	0.004342	0.001948	2.57998
z_xint_at	0.221559	0.030137	1.956213e-13	0.030137	1.956213e-13	1.248021	0.018250	0.002510	3.57713
z_interest_coverage	-0.097633	0.017619	3.001759e-08	0.017619	3.001759e-08	0.906982	-0.008042	0.001450	2.90062
z_debt_to_ebitda	-0.008094	0.026848	7.630537e-01	0.026848	7.630537e-01	0.991939	-0.000667	0.002210	7.62928
z_ebit_to_capital	0.077400	0.016009	1.331999e-06	0.016009	1.331999e-06	1.080474	0.006376	0.001327	1.53910
z_ocf_to_debt	-0.270714	0.071701	1.596115e-04	0.071701	1.596115e-04	0.762834	-0.022299	0.005908	1.60352
z_fcf_to_debt	0.207613	0.062332	8.661457e-04	0.062332	8.661457e-04	1.230736	0.017102	0.005135	8.67930
z_capx_at	-0.039080	0.026643	1.424279e-01	0.026643	1.424279e-01	0.961674	-0.003219	0.002194	1.42259
z_re_at	-0.057848	0.022355	9.661593e-03	0.022355	9.661593e-03	0.943794	-0.004765	0.001846	9.85935

7A.5 Walk-forward validation (expanding window)

In [97]:
```
trainpool_df = df_model.loc[df_model["split"].isin(["train","val"]), :].copy()
years = sorted(trainpool_df["label_year"].unique().tolist())
years = [int(y) for y in years if pd.notna(y)]

N_SPLITS = 4
if len(years) < (N_SPLITS + 2):
    print("Not enough years for walk-forward validation; skipping.")
    wf_tbl = pd.DataFrame()
else:
    # Choose split points evenly across the year range (excluding last year to keep a holdout)
    split_idx = np.linspace(2, len(years)-1, N_SPLITS, dtype=int)
    wf_rows = []
    for k in split_idx:
        train_years = years[:k]
        val_year = years[k]
        tr = trainpool_df["label_year"].isin(train_years)
        va = trainpool_df["label_year"].isin([val_year])

        X_tr = trainpool_df.loc[tr, [f"z_{c}" for c in continuous_feats_raw] + event_feats]
        y_tr = trainpool_df.loc[tr, TARGET_NAME].astype(int)
        X_va = trainpool_df.loc[va, [f"z_{c}" for c in continuous_feats_raw] + event_feats]
        y_va = trainpool_df.loc[va, TARGET_NAME].astype(int)

        mdl = LogisticRegression(C=best_C, solver="lbfgs", max_iter=2000, random_state=SEED)
        mdl.fit(X_tr, y_tr)
        p_va = mdl.predict_proba(X_va)[:, 1]

        wf_rows.append({
            "train_years_min": min(train_years),
            "train_years_max": max(train_years),
            "val_year": val_year,
            "n_train": int(len(y_tr)),
            "n_val": int(len(y_va)),
            "roc_auc": roc_auc_score(y_va, p_va),
```

```
            "pr_auc": average_precision_score(y_va, p_va),
            "brier": brier_score_loss(y_va, p_va),
        })
    wf_tbl = pd.DataFrame(wf_rows)
    display(wf_tbl)
```

	train_years_min	train_years_max	val_year	n_train	n_val	roc_auc	pr_auc	brier
0	2015	2016	2017	13214	6379	0.799745	0.491441	0.081091
1	2015	2017	2018	19593	6253	0.812571	0.535082	0.080945
2	2015	2019	2020	31962	6168	0.777843	0.482729	0.080091
3	2015	2021	2022	44351	6370	0.785159	0.505262	0.077200

7B. Tree-based model (XGBoost; nonlinear)

Tree models capture interactions and nonlinearities that logit cannot, but they require stronger regularization and calibration discipline.

Implementation details:

- Early stopping on **PR-AUC** using validation split.
- Conservative depth and regularization parameters.
- Cost-sensitive weighting to reflect class imbalance and FN/FP asymmetry.
- **Isotonic calibration** fit on validation predictions (train-only model remains unchanged).

7B.1 Train XGBoost with early stopping (validation PR-AUC)

In [98]:
```
# Build DMatrix objects
X_tr = splits["train"]["X"]
y_tr = splits["train"]["y"].astype(int)
X_va = splits["val"]["X"]
y_va = splits["val"]["y"].astype(int)
X_te = splits["test"]["X"]
y_te = splits["test"]["y"].astype(int)

n_pos = int(y_tr.sum())
n_neg = int((y_tr==0).sum())
imbalance = (n_neg / max(n_pos, 1))

w_pos = CONFIG["COST_FN"] * imbalance
w_neg = CONFIG["COST_FP"]

w_tr = np.where(y_tr.values==1, w_pos, w_neg).astype(float)
w_va = np.where(y_va.values==1, w_pos, w_neg).astype(float)

dtrain = xgb.DMatrix(X_tr, label=y_tr, weight=w_tr, feature_names=X_tr.columns.tolist())
dval   = xgb.DMatrix(X_va, label=y_va, weight=w_va, feature_names=X_tr.columns.tolist())
dall   = xgb.DMatrix(X, label=y.astype(int), feature_names=X_tr.columns.tolist())

evals = [(dtrain, "train"), (dval, "val")]

xgb_model = xgb.train(
    params=CONFIG["XGB_PARAMS"],
    dtrain=dtrain,
    num_boost_round=CONFIG["XGB_NUM_BOOST_ROUND"],
    evals=evals,
    early_stopping_rounds=CONFIG["XGB_EARLY_STOPPING"],
    verbose_eval=False,
)

print("Best iteration:", xgb_model.best_iteration)
```

```
Best iteration: 217
```

7B.2 Isotonic calibration on validation set (probability calibration)

In [99]:
```
# Raw probabilities (uncalibrated)
p_val_raw = xgb_model.predict(dval)
p_all_raw = xgb_model.predict(dall)

# Fit isotonic on validation only
iso = IsotonicRegression(out_of_bounds="clip")
iso.fit(p_val_raw, y_va.values.astype(int))

df_model["pd_tree"] = iso.transform(p_all_raw)

print("Calibration fitted on validation only.")
display(df_model[["split","pd_tree"]].groupby("split").mean())
```

```
Calibration fitted on validation only.
```

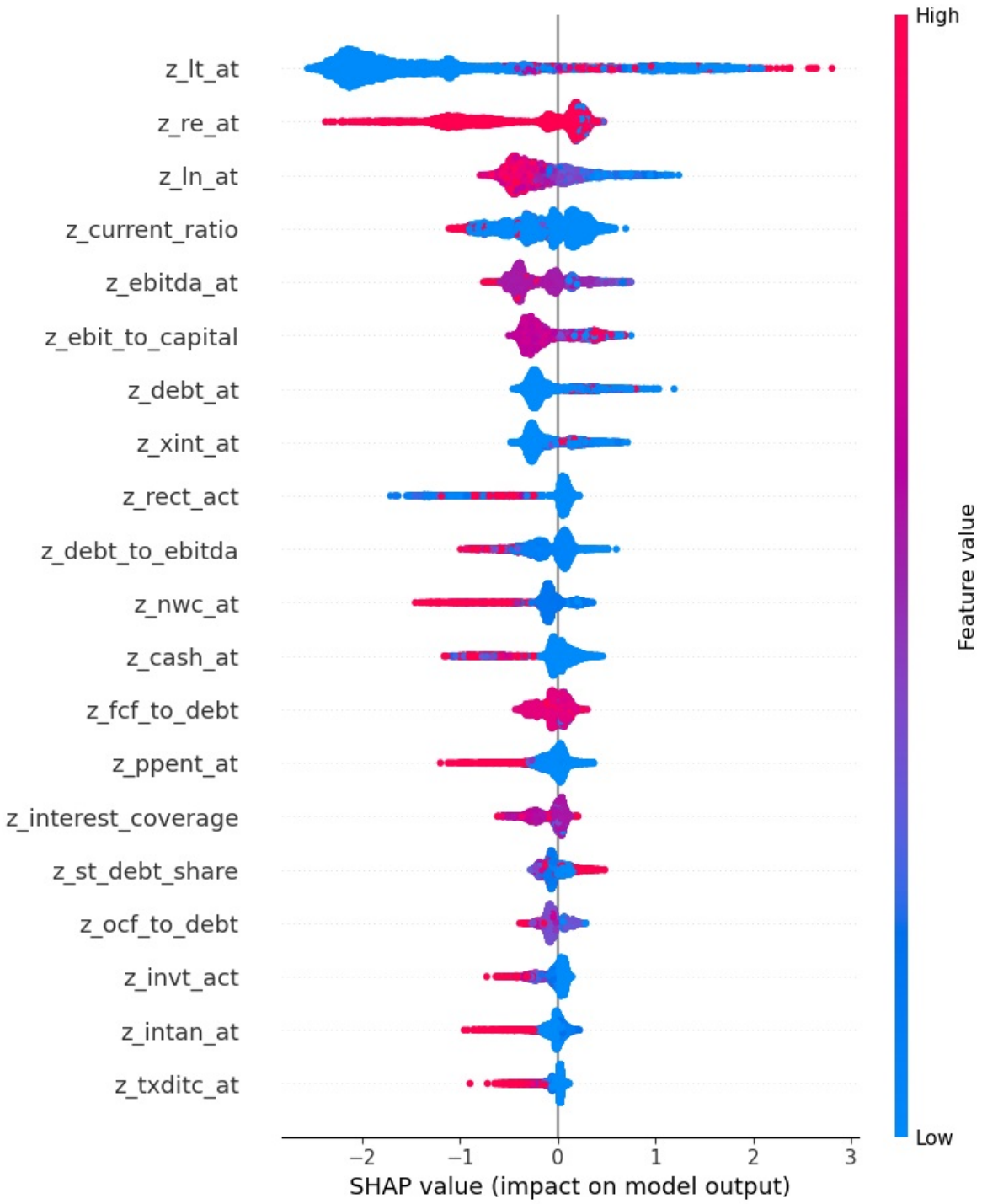
split	pd_tree
test	0.132238
train	0.142517
val	0.113658

7B.3 Feature importance and SHAP (optional explainability)

In [100…

```
# Gain-based feature importance
importance = xgb_model.get_score(importance_type="gain")
imp_tbl = (pd.DataFrame({"feature": list(importance.keys()), "gain": list(importance.values())})
             .sort_values("gain", ascending=False))
display(imp_tbl.head(20))

# Optional: SHAP summary for a subsample (can be expensive on large panels)
try:
    import shap
    shap.initjs()
    sample_n = min(5000, X_tr.shape[0])
    X_sample = X_tr.sample(sample_n, random_state=SEED)
    explainer = shap.TreeExplainer(xgb_model)
    shap_values = explainer.shap_values(X_sample)
    plt.figure()
    shap.summary_plot(shap_values, X_sample, show=False)
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / "shap_summary_tree.png", dpi=160)
    plt.show()
except Exception as e:
    print("SHAP skipped:", e)
```

	feature	gain
12	z_lt_at	945.924561
23	z_re_at	859.991943
15	z_ebitda_at	758.710999
16	z_xint_at	517.698303
13	z_debt_at	341.961884
3	z_nwc_at	316.666687
0	z_ln_at	250.306305
2	z_current_ratio	245.419266
19	z_ebit_to_capital	215.952332
6	z_rect_act	168.950378
1	z_cash_at	157.996155
20	z_ocf_to_debt	153.153809
18	z_debt_to_ebitda	146.621063
21	z_fcf_to_debt	144.956848
8	z_ppent_at	127.000687
14	z_st_debt_share	125.681755
4	z_aco_act	114.477135
7	z_invt_act	114.379890
22	z_capx_at	113.727875
11	z_txditc_at	103.934830



8. Model Evaluation & Diagnostic Monitoring

All evaluation in this section treats the **test split as untouchable**: no tuning based on test results.

Clean Evaluation Protocol Note:

- For **Logit**: The validation (`val`) performance reported below is an honest out-of-sample estimate because it uses a model trained on `train` only. The test performance uses a model trained on `train+val` .
- For **Tree (XGBoost)**: The validation (`val`) performance is **in-sample** relative to early stopping and isotonic calibration, both of which use the validation set. Therefore, `val` performance for trees will appear optimistic.
- **Unbiased Performance**: The **test split** results are the only strictly unbiased final performance metrics.

We report:

- ROC-AUC, PR-AUC, Brier score,
- calibration curve and calibration slope (reliability),
- persistence benchmark,
- collinearity and drift diagnostics.

8.1 Out-of-sample metrics (val and test) + persistence benchmark

In [101…

```
def eval_metrics(y_true: pd.Series, p: np.ndarray) -> dict:
    y_true = y_true.astype(int).values
    return {
        "roc_auc": roc_auc_score(y_true, p),
        "pr_auc": average_precision_score(y_true, p),
        "brier": brier_score_loss(y_true, p),
        "mean_p": float(np.mean(p)),
        "event_rate": float(np.mean(y_true)),
    }

rows = []
for sp in ["val","test"]:
    mask = df_model["split"] == sp
    y_sp = df_model.loc[mask, TARGET_NAME].astype(int)

    rows.append({"split": sp, "model":"logit", **eval_metrics(y_sp, df_model.loc[mask, "pd_logit"].values)})
    rows.append({"split": sp, "model":"tree_calibrated", **eval_metrics(y_sp, df_model.loc[mask, "pd_tree"].valu

    # Persistence benchmark: predict next-year distress = current-year PROXY_NAME
    pers = pd.to_numeric(df_model.loc[mask, PROXY_NAME], errors="coerce").fillna(0).astype(int).values
    rows.append({"split": sp, "model":"persistence", **eval_metrics(y_sp, pers)})

metrics_tbl = pd.DataFrame(rows).sort_values(["split","model"])
display(metrics_tbl)
```

	split	model	roc_auc	pr_auc	brier	mean_p	event_rate
3	test	logit	0.756942	0.429605	0.087748	0.112667	0.121023
5	test	persistence	0.848276	0.604476	0.056828	0.110823	0.121023
4	test	tree_calibrated	0.959206	0.750602	0.047786	0.132238	0.121023
0	val	logit	0.785159	0.505262	0.077200	0.108871	0.113658
2	val	persistence	0.829223	0.577720	0.056201	0.097331	0.113658
1	val	tree_calibrated	0.959934	0.758923	0.044605	0.113658	0.113658

8.1b Early-warning vs Surveillance decomposition (state-conditional evaluation)

In [102…

```
# Early-warning vs surveillance evaluation:
#   - Early warning: subset with PROXY_NAME == 0 (not currently distressed)
#   - Surveillance: subset with PROXY_NAME == 1 (currently distressed)
# Also add a state-only baseline: predict next-year distress using current state only.

import numpy as np
import pandas as pd

def safe_eval_metrics(y_true: pd.Series, p: np.ndarray) -> dict:
    y = y_true.astype(int).values
    out = {
        "roc_auc": np.nan,
        "pr_auc": np.nan,
        "brier": brier_score_loss(y, p),
        "mean_p": float(np.mean(p)),
        "event_rate": float(np.mean(y)),
        "n": int(len(y)),
    }
    if np.unique(y).size >= 2:
        out["roc_auc"] = roc_auc_score(y, p)
        out["pr_auc"] = average_precision_score(y, p)
    return out

def eval_segment(df_seg: pd.DataFrame, split_name: str, segment_name: str) -> list:
    rows = []
    if df_seg.empty:
        return rows

    y = df_seg[TARGET_NAME].astype(int)

    # State-only baseline (uses current distress state only)
```

```
    state = pd.to_numeric(df_seg[PROXY_NAME], errors="coerce").fillna(0).astype(int).values
    base = safe_eval_metrics(y, state)

    # Models
    for col, mdl in [("pd_logit","logit"),
                     ("pd_tree","tree_calibrated")]:
        met = safe_eval_metrics(y, df_seg[col].values)

        rows.append({
            "split": split_name,
            "segment": segment_name,
            "model": mdl,
            **met,
            "baseline_roc_auc": base["roc_auc"],
            "baseline_pr_auc": base["pr_auc"],
            "baseline_brier": base["brier"],
            "delta_roc_auc": (met["roc_auc"] - base["roc_auc"]) if (met["roc_auc"]==met["roc_auc"] and base["ro
            "delta_pr_auc": (met["pr_auc"] - base["pr_auc"]) if (met["pr_auc"]==met["pr_auc"] and base["pr_auc"
            "delta_brier": met["brier"] - base["brier"],  # negative is improvement
        })

    # Add baseline as a row for reference
    rows.append({
        "split": split_name,
        "segment": segment_name,
        "model": "state_only",
        **base,
        "baseline_roc_auc": np.nan,
        "baseline_pr_auc": np.nan,
        "baseline_brier": np.nan,
        "delta_roc_auc": 0.0,
        "delta_pr_auc": 0.0,
        "delta_brier": 0.0,
    })
    return rows

seg_rows = []
for sp in ["val", "test"]:
    df_sp = df_model.loc[df_model["split"]==sp, :].copy()

    # Only evaluate segments where current distress state is observed.
    dcur = pd.to_numeric(df_sp[PROXY_NAME], errors="coerce")
    df_sp = df_sp.loc[dcur.notna(), :].copy()
    df_sp["distress_t_int"] = dcur.loc[dcur.notna()].astype(int)

    seg_rows += eval_segment(df_sp.loc[df_sp["distress_t_int"]==0, :], sp, f"early_warning ({PROXY_NAME}=0)")
    seg_rows += eval_segment(df_sp.loc[df_sp["distress_t_int"]==1, :], sp, f"surveillance ({PROXY_NAME}=1)")

seg_metrics_tbl = pd.DataFrame(seg_rows)

if not seg_metrics_tbl.empty:
    seg_metrics_tbl = seg_metrics_tbl.sort_values(["split","segment","model"])
    display(seg_metrics_tbl)
else:
    print("No segment metrics computed (empty segments).")
```

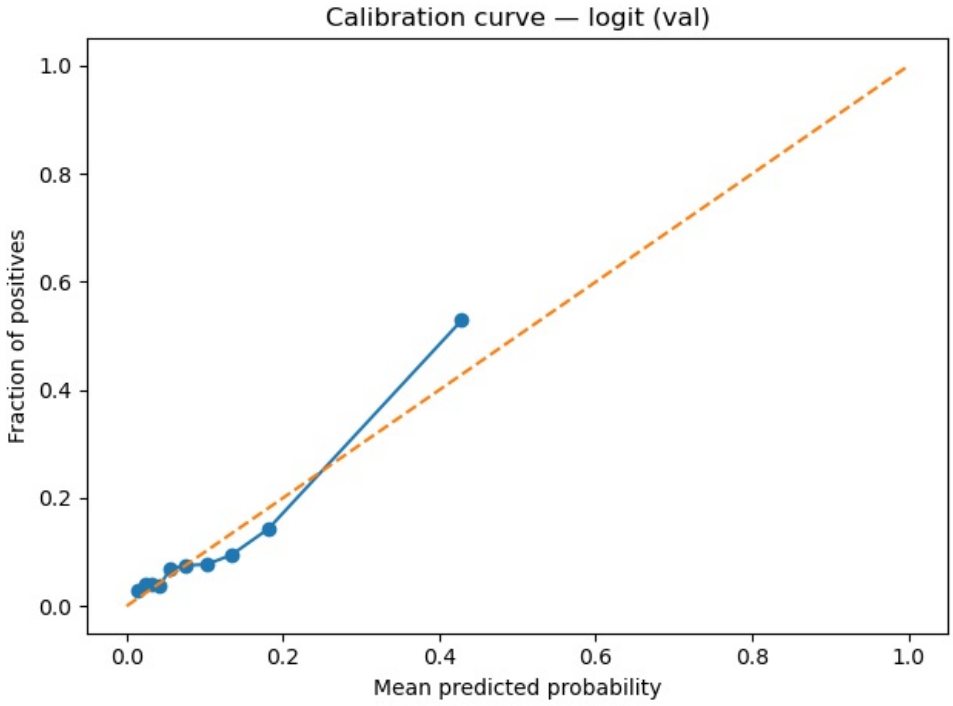
	split	segment	model	roc_auc	pr_auc	brier	mean_p	event_rate	n	baseline_roc_auc	baseline_pr_a
6	test	early_warning (distress_v2_t=0)	logit	0.704411	0.097092	0.040803	0.089903	0.037691	10984	0.5	0.0376
8	test	early_warning (distress_v2_t=0)	state_only	0.500000	0.037691	0.037691	0.000000	0.037691	10984	NaN	N
7	test	early_warning (distress_v2_t=0)	tree_calibrated	0.918341	0.290356	0.031368	0.054960	0.037691	10984	0.5	0.0376
9	test	surveillance (distress_v2_t=1)	logit	0.518778	0.817431	0.464406	0.295316	0.789627	1369	0.5	0.7896
11	test	surveillance (distress_v2_t=1)	state_only	0.500000	0.789627	0.210373	1.000000	0.789627	1369	NaN	N
10	test	surveillance (distress_v2_t=1)	tree_calibrated	0.590136	0.835140	0.179512	0.752266	0.789627	1369	0.5	0.7896
0	val	early_warning (distress_v2_t=0)	logit	0.708317	0.122533	0.040986	0.084632	0.040174	5750	0.5	0.0401
2	val	early_warning (distress_v2_t=0)	state_only	0.500000	0.040174	0.040174	0.000000	0.040174	5750	NaN	N
1	val	early_warning (distress_v2_t=0)	tree_calibrated	0.920356	0.321564	0.030991	0.045986	0.040174	5750	0.5	0.0401
3	val	surveillance (distress_v2_t=1)	logit	0.608088	0.873682	0.413057	0.333665	0.795161	620	0.5	0.7951
5	val	surveillance (distress_v2_t=1)	state_only	0.500000	0.795161	0.204839	1.000000	0.795161	620	NaN	N
4	val	surveillance (distress_v2_t=1)	tree_calibrated	0.644120	0.861654	0.170864	0.741259	0.795161	620	0.5	0.7951

8.2 Calibration diagnostics (curve + calibration-in-the-large + slope)

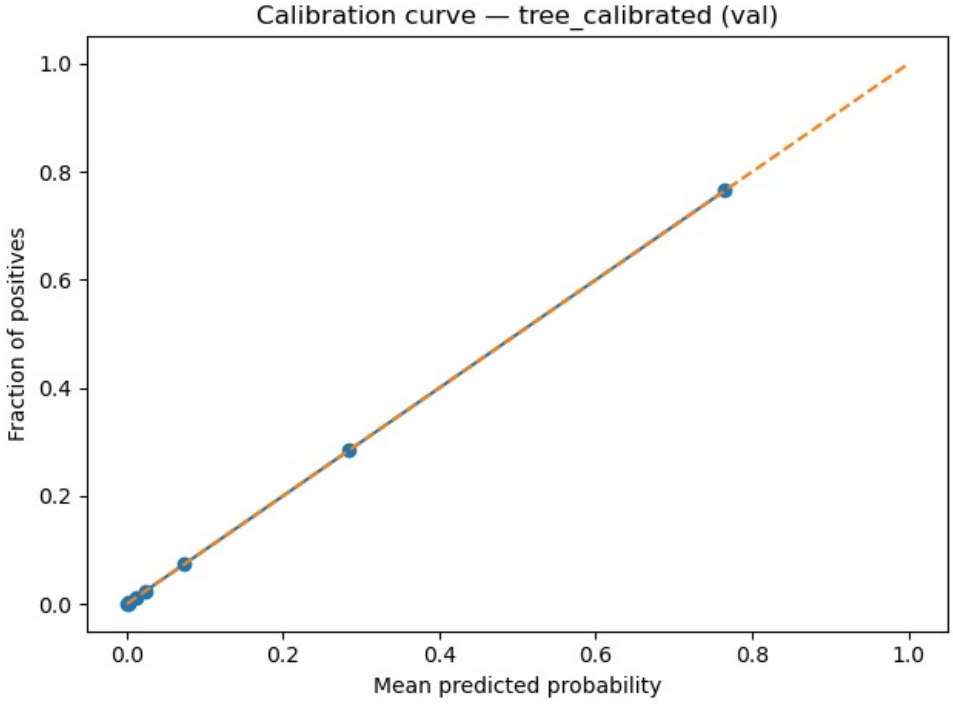
In [103…

```
def calibration_slope_intercept(y_true: np.ndarray, p: np.ndarray) -> tuple[float,float]:
    z = logit(p)
    Xc = sm.add_constant(z, has_constant="add")
    mdl = sm.GLM(y_true, Xc, family=sm.families.Binomial())
    res = mdl.fit()
    intercept, slope = res.params[0], res.params[1]
    return float(intercept), float(slope)

def plot_calibration(y_true: np.ndarray, p: np.ndarray, title: str, fname: str):
    frac_pos, mean_pred = calibration_curve(y_true, p, n_bins=10, strategy="quantile")
    plt.figure()
    plt.plot(mean_pred, frac_pos, marker="o")
    plt.plot([0,1],[0,1], linestyle="--")
    plt.xlabel("Mean predicted probability")
    plt.ylabel("Fraction of positives")
    plt.title(title)
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / fname, dpi=160)
    plt.show()

for sp in ["val","test"]:
    mask = df_model["split"] == sp
    y_sp = df_model.loc[mask, TARGET_NAME].astype(int).values

    for model_name, pcol in [("logit","pd_logit"), ("tree_calibrated","pd_tree")]:
        p = df_model.loc[mask, pcol].values
        icpt, slope = calibration_slope_intercept(y_sp, p)
        print(f"{sp} | {model_name}: calibration intercept={icpt:.3f}, slope={slope:.3f}")
        plot_calibration(y_sp, p, f"Calibration curve — {model_name} ({sp})", f"cal_curve_{model_name}_{sp}.png
```

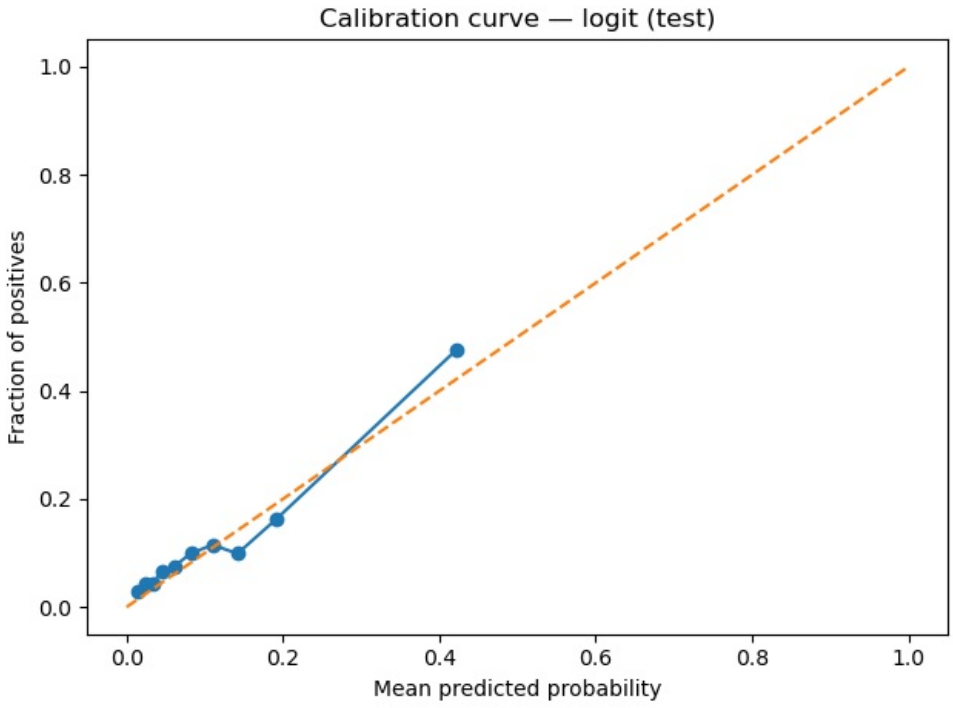
```
val | logit: calibration intercept=0.170, slope=1.060
```



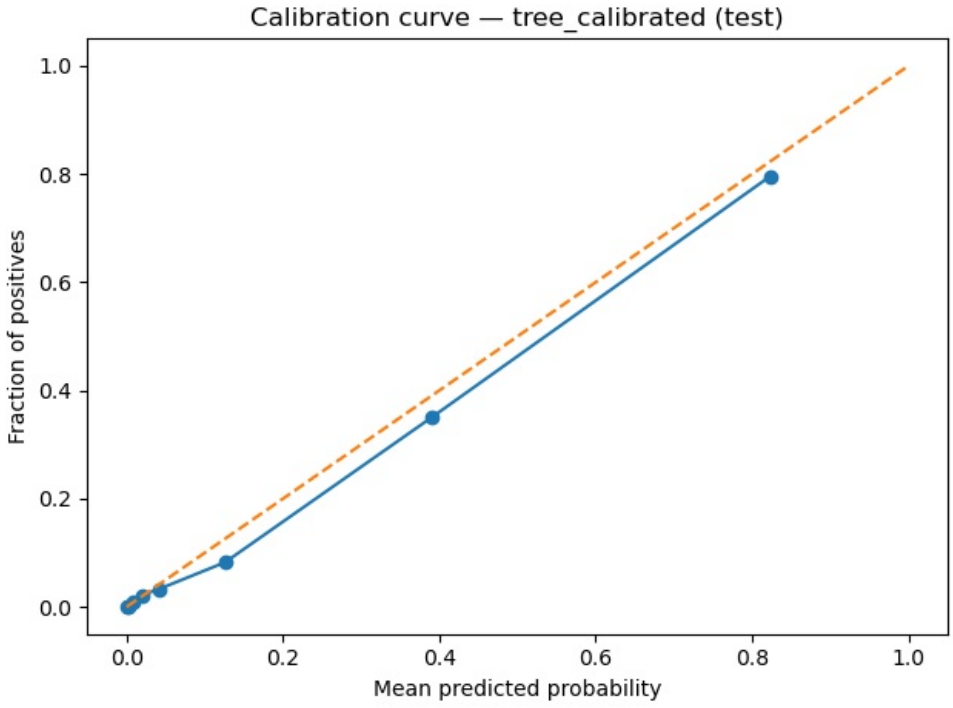

```
val | tree_calibrated: calibration intercept=0.000, slope=1.000
```




```
test | logit: calibration intercept=-0.053, slope=0.916
```




```
test | tree_calibrated: calibration intercept=-0.215, slope=1.016
```

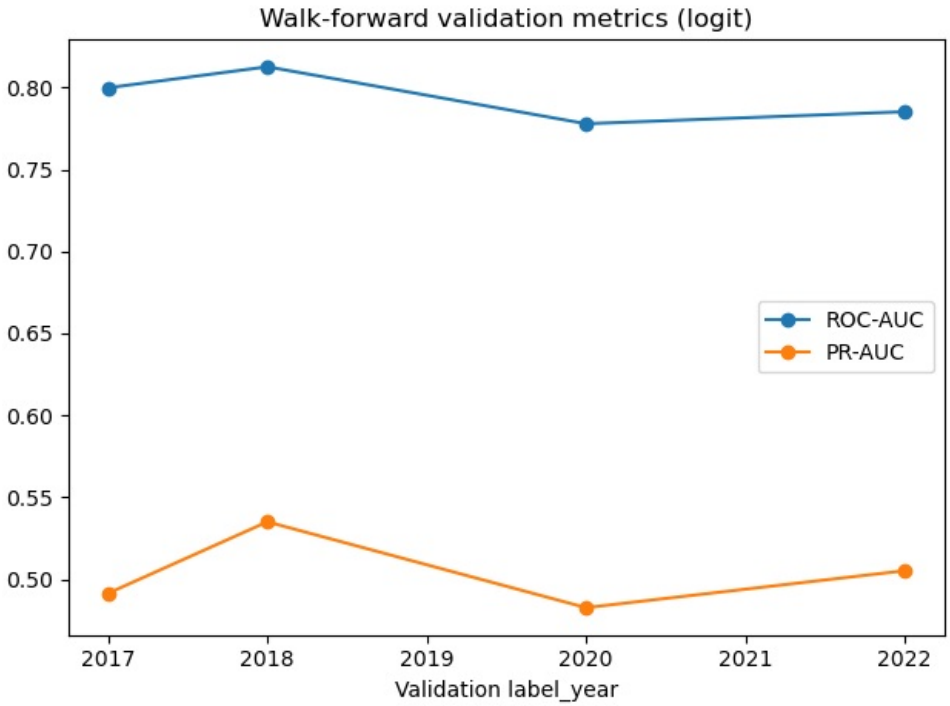


8.3 Temporal stability (walk-forward fold metrics)

In [104…

```python
if 'wf_tbl' in globals() and len(wf_tbl) > 0:
    display(wf_tbl)
    plt.figure()
    plt.plot(wf_tbl["val_year"], wf_tbl["roc_auc"], marker="o", label="ROC-AUC")
    plt.plot(wf_tbl["val_year"], wf_tbl["pr_auc"], marker="o", label="PR-AUC")
    plt.title("Walk-forward validation metrics (logit)")
    plt.xlabel("Validation label_year")
    plt.legend()
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / "walkforward_metrics_logit.png", dpi=160)
    plt.show()
```

	train_years_min	train_years_max	val_year	n_train	n_val	roc_auc	pr_auc	brier
0	2015	2016	2017	13214	6379	0.799745	0.491441	0.081091
1	2015	2017	2018	19593	6253	0.812571	0.535082	0.080945
2	2015	2019	2020	31962	6168	0.777843	0.482729	0.080091
3	2015	2021	2022	44351	6370	0.785159	0.505262	0.077200



8.4 Collinearity checks (VIF + high-correlation pairs)

In [105…

```
# VIF on continuous z-features (train only)
X_vif = splits["train"]["X"][[f"z_{c}" for c in continuous_feats_raw]].copy()
X_vif = sm.add_constant(X_vif, has_constant="add")

vif_rows = []
for i, col in enumerate(X_vif.columns):
    if col == "const":
        continue
    vif_rows.append({"feature": col, "VIF": float(variance_inflation_factor(X_vif.values, i))})

vif_tbl = pd.DataFrame(vif_rows).sort_values("VIF", ascending=False)
display(vif_tbl)

# Correlation screen (continuous only)
corr = splits["train"]["X"][[f"z_{c}" for c in continuous_feats_raw]].corr()
high_pairs = []
for i in range(len(corr.columns)):
    for j in range(i+1, len(corr.columns)):
        v = corr.iloc[i,j]
        if abs(v) >= 0.85:
            high_pairs.append((corr.columns[i], corr.columns[j], float(v)))
high_pairs_tbl = pd.DataFrame(high_pairs, columns=["feat1","feat2","corr"]).sort_values("corr", key=np.abs, asc
display(high_pairs_tbl)
```

	feature	VIF
13	z_debt_at	3.254520
16	z_xint_at	2.782242
12	z_lt_at	2.736720
20	z_ocf_to_debt	2.546488
21	z_fcf_to_debt	2.531441
8	z_ppent_at	2.347146
22	z_capx_at	2.324664
1	z_cash_at	2.285567
23	z_re_at	2.052515
15	z_ebitda_at	1.959064
3	z_nwc_at	1.817332
5	z_lco_lct	1.774798
2	z_current_ratio	1.744278
6	z_rect_act	1.723576
4	z_aco_act	1.581921
9	z_intan_at	1.566156
11	z_txditc_at	1.553209
0	z_ln_at	1.550161
7	z_invt_act	1.369634
10	z_txp_lct	1.254853
14	z_st_debt_share	1.051779
17	z_interest_coverage	1.033069
18	z_debt_to_ebitda	1.023999
19	z_ebit_to_capital	1.012181

feat1	feat2	corr

8.5 Drift diagnostics (standardized mean difference: train vs test)

In [106…

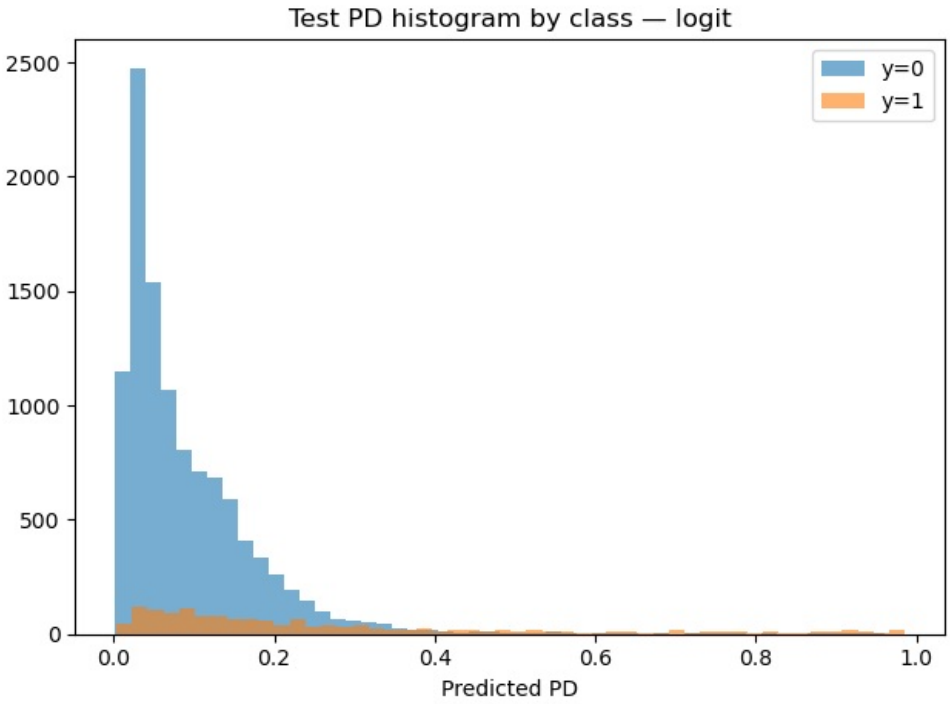
```
feat_cols = [f"z_{c}" for c in continuous_feats_raw] + event_feats
drift_rows = []
for c in feat_cols:
    smd = compute_smd(df_model.loc[df_model["split"]=="train", c], df_model.loc[df_model["split"]=="test", c])
    drift_rows.append({"feature": c, "SMD_train_vs_test": smd})
drift_tbl = pd.DataFrame(drift_rows).sort_values("SMD_train_vs_test", key=lambda s: s.abs(), ascending=False)
display(drift_tbl.head(25))
```

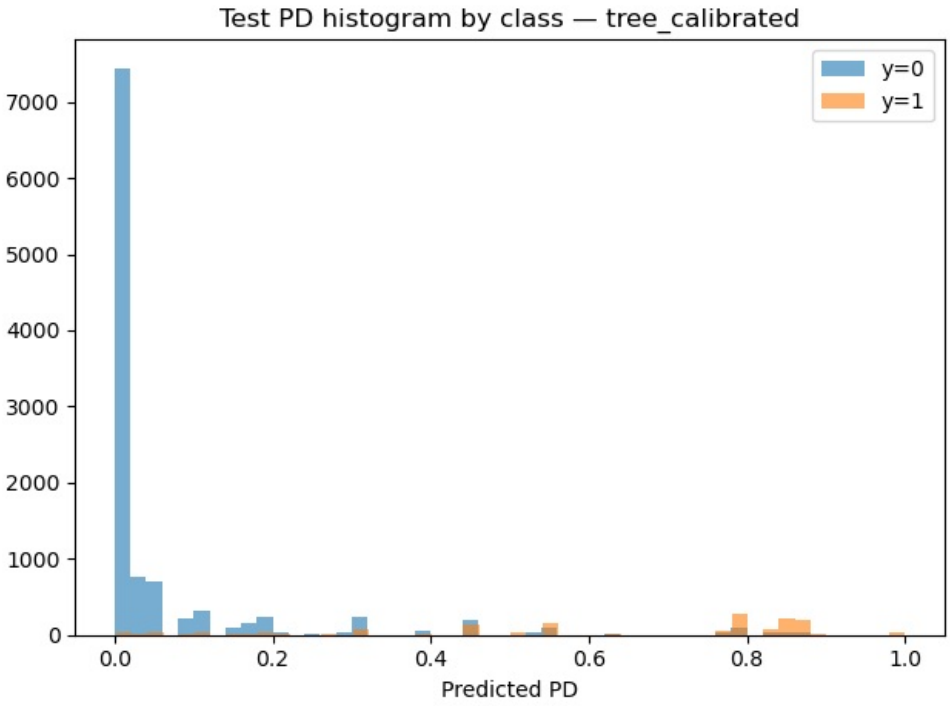
	feature	SMD_train_vs_test
20	z_ocf_to_debt	0.091882
0	z_ln_at	-0.079524
14	z_st_debt_share	-0.062415
22	z_capx_at	0.036250
17	z_interest_coverage	0.034499
11	z_txditc_at	0.031830
16	z_xint_at	0.029446
21	z_fcf_to_debt	0.029084
23	z_re_at	-0.025731
8	z_ppent_at	0.022198
6	z_rect_act	0.019723
10	z_txp_lct	-0.018629
12	z_lt_at	0.017416
5	z_lco_lct	-0.013873
19	z_ebit_to_capital	-0.012974
1	z_cash_at	-0.011300
9	z_intan_at	0.011047
3	z_nwc_at	-0.010260
2	z_current_ratio	-0.009573
13	z_debt_at	-0.006061
7	z_invt_act	0.005669
15	z_ebitda_at	0.005416
4	z_aco_act	0.000731
18	z_debt_to_ebitda	-0.000531

8.6 Probability distributions by class (test split)

In [107…

```
mask = df_model["split"]=="test"
y_true = df_model.loc[mask, TARGET_NAME].astype(int)

for model_name, pcol in [("logit","pd_logit"), ("tree_calibrated","pd_tree")]:
    p = df_model.loc[mask, pcol]
    plt.figure()
    plt.hist(p[y_true==0], bins=50, alpha=0.6, label="y=0")
    plt.hist(p[y_true==1], bins=50, alpha=0.6, label="y=1")
    plt.title(f"Test PD histogram by class — {model_name}")
    plt.xlabel("Predicted PD")
    plt.legend()
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / f"pd_hist_{model_name}_test.png", dpi=160)
    plt.show()
```





9. Operational Risk Management Layer (Events + PDs)

This section uses a **two-layer** design:

- **Model layer (prediction):** calibrated probability of next-year distress (**PD**) from accounting ratios and structural predictors.
- **Indicator layer (events):** discrete `evt_*` early-warning indicators **not used as predictors**. They serve governance, monitoring, and decision support.

This structure matches four operational functions commonly discussed in risk-management systems:

1. **Risk awareness:** documented prior knowledge of which indicators flag trouble (event dictionary + empirical lift).
2. **Monitoring and warning:** continuous tracking of event activation/persistence and PD levels over the panel.
3. **Communication:** translating signals into decision-maker-friendly views (risk tiers/deciles, transitions, reason codes).
4. **Response capability:** predefined action rules (screen / monitor / no action) based on PDs and events under explicit costs and capacity constraints.

9.1 Risk awareness — event dictionary and conditional risk (lift)

In [108…

```
EVT_COLS = event_dict["event"].tolist()
print("Decision-support events:", EVT_COLS)

# Optional: enrich the event dictionary with a simple mechanism taxonomy (appendix-ready)
```

```python
event_dict_enriched = event_dict.copy()
mech_map = {
    "evt_divcut": "Payout policy",
    "evt_divsusp": "Payout policy",
    "evt_divinit": "Payout policy",
    "evt_liq_squeeze": "Liquidity",
    "evt_quick_squeeze": "Liquidity",
    "evt_ebitdadrop": "Operating deterioration",
    "evt_cfdrop": "Operating deterioration",
}
event_dict_enriched["mechanism"] = event_dict_enriched["event"].map(mech_map).fillna("Other/unspecified")
display(event_dict_enriched)

def event_lift_table(df_in: pd.DataFrame, events: list[str], y_col: str) -> pd.DataFrame:
    # Event lift with explicit missingness handling.
    # - prevalence_obs: among observations where the event is observed (not NA)
    # - missing_rate: definitional missingness (insufficient inputs)
    # - cond_distress_rate: P(y=1 | evt=1, evt observed)

    base = df_in[y_col].astype(float).mean()
    rows = []
    for e in events:
        if e not in df_in.columns:
            continue

        s = pd.to_numeric(df_in[e], errors="coerce")  # may contain NA by construction
        miss = float(s.isna().mean())
        obs = s.notna()
        if obs.sum() == 0:
            continue

        s_obs = s[obs].astype(int)
        n_event = int((s_obs == 1).sum())
        if n_event == 0:
            continue

        rate = df_in.loc[obs & (s == 1), y_col].astype(float).mean()
        prev = float((s_obs == 1).mean())

        rows.append({
            "event": e,
            "mechanism": mech_map.get(e, "Other/unspecified"),
            "n_obs_event": int(obs.sum()),
            "n_event": n_event,
            "missing_rate": miss,
            "prevalence_obs": prev,
            "cond_distress_rate": float(rate),
            "base_rate": float(base),
            "lift_vs_base": float(rate/base) if base > 0 else np.nan,
        })

    out = pd.DataFrame(rows)
    if not out.empty:
        out = out.sort_values(["lift_vs_base", "n_event"], ascending=[False, False])
    return out

for sp in ["train", "test"]:
    df_sp = df_model.loc[df_model["split"] == sp, :].copy()
    print(f"\nEvent lift (labels: {TARGET_NAME}) — {sp}")
    display(event_lift_table(df_sp, EVT_COLS, TARGET_NAME).head(25))
```

```
Decision-support events: ['evt_divcut', 'evt_divsusp', 'evt_divinit', 'evt_liq_squeeze', 'evt_quick_squeeze', 'e
vt_ebitdadrop', 'evt_cfdrop']
```

	event	definition	inputs	calibration	parameter	mechanism
0	evt_divcut	Dividend YoY % change <= training P10 threshol...	dvc	train-only	{"DIV_CUT_THR_P10_BOUNDED": -0.5}	Payout policy
1	evt_divsusp	Dividend >0 at t-1 and ~0 at t	dvc	none		Payout policy
2	evt_divinit	Dividend ~0 at t-1 and >0 at t	dvc	none		Payout policy
3	evt_liq_squeeze	Current ratio < 1.0	act,lct	fixed threshold		Liquidity
4	evt_quick_squeeze	Quick ratio < 0.8	act,lct,invt (or che+rect)	fixed threshold		Liquidity
5	evt_ebitdadrop	EBITDA <=0 OR EBITDA/EBITDA_{t-1}<0.5 (require...	oibdp	fixed threshold		Operating deterioration
6	evt_cfdrop	CFO <=0 OR CFO/CFO_{t-1}<0.5 (requires CFO_{t-...	oancf	fixed threshold		Operating deterioration

```
Event lift (labels: target_next_v2) — train
```

	event	mechanism	n_obs_event	n_event	missing_rate	prevalence_obs	cond_distress_rate	base_rate	lift_vs_base
3	evt_liq_squeeze	Liquidity	44351	12287	0.000000	0.277040	0.284773	0.12144	2.344958
4	evt_quick_squeeze	Liquidity	44351	13078	0.000000	0.294875	0.264414	0.12144	2.177312
1	evt_divsusp	Payout policy	35052	728	0.209668	0.020769	0.094780	0.12144	0.780467
6	evt_cfdrop	Operating deterioration	35052	5311	0.209668	0.151518	0.087178	0.12144	0.717863
5	evt_ebitdadrop	Operating deterioration	35052	4279	0.209668	0.122076	0.071278	0.12144	0.586941
0	evt_divcut	Payout policy	13292	2366	0.700300	0.178002	0.063821	0.12144	0.525532
2	evt_divinit	Payout policy	35052	700	0.209668	0.019970	0.062857	0.12144	0.517597

```
Event lift (labels: target_next_v2) — test
```

	event	mechanism	n_obs_event	n_event	missing_rate	prevalence_obs	cond_distress_rate	base_rate	lift_vs_base
3	evt_liq_squeeze	Liquidity	12353	3293	0.000000	0.266575	0.280292	0.121023	2.316014
4	evt_quick_squeeze	Liquidity	12353	3508	0.000000	0.283980	0.251140	0.121023	2.075141
1	evt_divsusp	Payout policy	11613	191	0.059904	0.016447	0.073298	0.121023	0.605656
2	evt_divinit	Payout policy	11613	289	0.059904	0.024886	0.072664	0.121023	0.600417
6	evt_cfdrop	Operating deterioration	11613	1831	0.059904	0.157668	0.063900	0.121023	0.527994
5	evt_ebitdadrop	Operating deterioration	11613	1372	0.059904	0.118143	0.063411	0.121023	0.523958
0	evt_divcut	Payout policy	4061	576	0.671254	0.141837	0.039931	0.121023	0.329941

9.2 Monitoring and warning — event dynamics, persistence, and PD×event risk grids

Monitoring should reflect (i) **activation** (0→1), (ii) **persistence** (1→1), and (iii) how event regimes interact with PDs. We treat events as *operational indicators* (not predictors) and monitor them jointly with calibrated PDs.

In [109…

```python
# --- 9.2A Event activation and persistence (adjacency-safe) ---
def transition_stats(df_in: pd.DataFrame, event: str) -> dict:
    # Robust transition stats that enforce year adjacency and handle NaNs.
    s = pd.to_numeric(df_in[event], errors="coerce")
    s_l1 = lag(df_in, event, 1)

    valid = s.notna() & s_l1.notna()
    if valid.sum() == 0:
        return {"event": event, "activation_01_rate": np.nan, "persistence_11_rate": np.nan, "n_transitions": 0

    s0 = s_l1[valid].astype(int)
    s1 = s[valid].astype(int)

    act_01 = ((s0 == 0) & (s1 == 1)).mean()
    pers_11 = ((s0 == 1) & (s1 == 1)).mean()
    return {
        "event": event,
        "activation_01_rate": float(act_01),
        "persistence_11_rate": float(pers_11),
        "n_transitions": int(valid.sum()),
    }

rows = [transition_stats(df_model, e) for e in EVT_COLS]
trans_tbl = pd.DataFrame(rows)
if not trans_tbl.empty:
    trans_tbl = trans_tbl.sort_values("activation_01_rate", ascending=False)
display(trans_tbl)

# --- 9.2B Monitoring summary by fiscal year (panel-level tracking) ---
def monitoring_by_year(df_in: pd.DataFrame, p_col: str, y_col: str, evt_cols: list[str]) -> pd.DataFrame:
    d = df_in[["fyear", "split", p_col, y_col] + evt_cols].copy()

    # Event aggregation: triggered count; missingness summarized separately.
    evt_mat = d[evt_cols].apply(pd.to_numeric, errors="coerce")
    d["evt_missing_rate_mean"] = evt_mat.isna().mean(axis=1)
    d["evt_count"] = (evt_mat.fillna(0) == 1).sum(axis=1)
    d["evt_any"] = (d["evt_count"] > 0).astype(int)
```

```
    out = (d.groupby(["split", "fyear"])
              .agg(
                  n=("fyear","size"),
                  mean_pd=(p_col,"mean"),
                  realized_rate=(y_col,"mean"),
                  evt_any_rate=("evt_any","mean"),
                  mean_evt_count=("evt_count","mean"),
                  mean_evt_missing=("evt_missing_rate_mean","mean"),
              )
              .reset_index()
              .sort_values(["split","fyear"]))
    return out

print("\nMonitoring by year — calibrated tree PD (pd_tree)")
display(monitoring_by_year(df_model, "pd_tree", TARGET_NAME, EVT_COLS).head(40))

# --- 9.2C PD × Event risk grid (operational triangulation) ---
def pd_event_grid(df_in: pd.DataFrame, p_col: str, y_col: str, evt_cols: list[str], n_bins: int = 10) -> pd.Data
    d = df_in[[p_col, y_col] + evt_cols].dropna(subset=[p_col, y_col]).copy()
    evt_mat = d[evt_cols].apply(pd.to_numeric, errors="coerce")
    d["evt_count"] = (evt_mat.fillna(0) == 1).sum(axis=1)

    d["evt_bucket"] = pd.cut(d["evt_count"], bins=[-0.1, 0.5, 1.5, 10**6], labels=["0", "1", "2+"])
    d["pd_decile"] = pd.qcut(d[p_col], q=n_bins, labels=False, duplicates="drop") + 1

    g = (d.groupby(["pd_decile","evt_bucket"])
          .agg(
              n=("pd_decile","size"),
              mean_pd=(p_col,"mean"),
              realized_rate=(y_col,"mean"),
          )
          .reset_index())
    return g.sort_values(["pd_decile","evt_bucket"])

print("\nTest split PD × Events grid — calibrated tree")
display(pd_event_grid(df_model.loc[df_model["split"]=="test", :], "pd_tree", TARGET_NAME, EVT_COLS, n_bins=10))
```

	event	activation_01_rate	persistence_11_rate	n_transitions
6	evt_cfdrop	0.136671	0.021039	43206
0	evt_divcut	0.127247	0.026002	15576
4	evt_quick_squeeze	0.120782	0.165027	52458
3	evt_liq_squeeze	0.113329	0.151893	52458
5	evt_ebitdadrop	0.104800	0.014350	43206
2	evt_divinit	0.020715	0.000000	43206
1	evt_divsusp	0.020553	0.000000	43206

```
Monitoring by year — calibrated tree PD (pd_tree)
```

	split	fyear	n	mean_pd	realized_rate	evt_any_rate	mean_evt_count	mean_evt_missing
0	test	2022	6276	0.128536	0.119822	0.506692	0.906469	0.136984
1	test	2023	6077	0.136061	0.122264	0.497285	0.883824	0.123040
2	train	2014	6715	0.139430	0.135071	0.324348	0.563366	0.714286
3	train	2015	6499	0.147182	0.130635	0.531928	0.959532	0.128767
4	train	2016	6379	0.148024	0.118984	0.515912	0.934943	0.128816
5	train	2017	6253	0.141551	0.12426	0.499920	0.887574	0.130155
6	train	2018	6116	0.137494	0.126226	0.493787	0.873774	0.127114
7	train	2019	6168	0.145190	0.116245	0.521887	0.93904	0.136627
8	train	2020	6221	0.138587	0.097251	0.512940	0.977335	0.138517
9	val	2021	6370	0.113658	0.113658	0.486185	0.863108	0.149765

```
Test split PD × Events grid — calibrated tree
```

	pd_decile	evt_bucket	n	mean_pd	realized_rate
0	1	0	1595	0.000626	0.0
1	1	1	844	0.000609	0.0
2	1	2+	576	0.000664	0.0
3	2	0	1127	0.001706	0.003549
4	2	1	489	0.001705	0.0
5	2	2+	556	0.001706	0.0
6	3	0	1080	0.008629	0.010185
7	3	1	330	0.008390	0.006061
8	3	2+	458	0.009148	0.004367
9	4	0	252	0.019845	0.02381
10	4	1	62	0.019843	0.0
11	4	2+	103	0.019805	0.029126
12	5	0	852	0.041457	0.038732
13	5	1	232	0.041328	0.021552
14	5	2+	437	0.041218	0.02746
15	6	0	506	0.125458	0.086957
16	6	1	141	0.122985	0.049645
17	6	2+	325	0.127622	0.089231
18	7	0	524	0.378627	0.354962
19	7	1	194	0.392929	0.304124
20	7	2+	548	0.401336	0.363139
21	8	0	215	0.803865	0.832558
22	8	1	87	0.792037	0.770115
23	8	2+	820	0.831543	0.789024

9.3 Communication — risk tiers, transitions, and reason codes

Communication should translate model outputs and indicator triggers into decision-maker-friendly artifacts:

- **Risk tiers/deciles:** expected vs realized risk by PD bucket.
- **Transitions:** PD movements and event activations/persistence.
- **Reason codes:** simple, interpretable attributions for material PD jumps (based on newly triggered events).

In [110…

```
def decile_table(df_in: pd.DataFrame, p_col: str, y_col: str) -> pd.DataFrame:
    d = df_in[[p_col, y_col]].dropna().copy()
    d["decile"] = pd.qcut(d[p_col], 10, labels=False, duplicates="drop") + 1
    out = d.groupby("decile").agg(
        n=("decile","size"),
        mean_pd=(p_col,"mean"),
        realized_rate=(y_col,"mean"),
    ).reset_index()
    out["calibration_gap"] = out["realized_rate"] - out["mean_pd"]
    return out

for model_name, pcol in [("logit","pd_logit"), ("tree_calibrated","pd_tree")]:
    print(f"\nTest deciles — {model_name}")
    dt = decile_table(df_model.loc[df_model["split"]=="test", :], pcol, TARGET_NAME)
    display(dt)
    # --- 9.3A Reason codes for large PD jumps (event-based) ---
# When PD decile increases materially year-over-year, summarize which events newly activated.

def reason_codes_for_pd_jumps(df_in: pd.DataFrame, p_col: str, evt_cols: list[str], min_decile_jump: int = 3, sp
    d = df_in.loc[df_in["split"] == split, ["firm_id","fyear", p_col] + evt_cols].copy()
    d = d.sort_values(["firm_id","fyear"])

    # PD deciles within the split (communication tiering)
    d["pd_decile"] = pd.qcut(d[p_col], 10, labels=False, duplicates="drop") + 1
    d["pd_decile_l1"] = lag(d, "pd_decile", 1)
    d["decile_jump"] = d["pd_decile"] - d["pd_decile_l1"]

    jump_mask = d["decile_jump"].notna() & (d["decile_jump"] >= min_decile_jump)
    if int(jump_mask.sum()) == 0:
        return pd.DataFrame(columns=["event","n_new_activation_in_jumps","share_of_jumps"])
```

```
    n_jumps = int(jump_mask.sum())
    rows = []
    for e in evt_cols:
        s = pd.to_numeric(d[e], errors="coerce")
        s_l1 = lag(d, e, 1)
        valid = jump_mask & s.notna() & s_l1.notna()
        if int(valid.sum()) == 0:
            continue
        new_act = int(((s_l1[valid].astype(int) == 0) & (s[valid].astype(int) == 1)).sum())
        if new_act > 0:
            rows.append({
                "event": e,
                "n_new_activation_in_jumps": new_act,
                "share_of_jumps": float(new_act / n_jumps),
            })

    out = pd.DataFrame(rows).sort_values("n_new_activation_in_jumps", ascending=False)
    return out

print("\nReason codes — events newly activating during large PD jumps (test split)")
display(reason_codes_for_pd_jumps(df_model, "pd_tree", EVT_COLS, min_decile_jump=3, split="test").head(20))
```

```
Test deciles — logit
```

	decile	n	mean_pd	realized_rate	calibration_gap
0	1	1236	0.014578	0.027508	0.012931
1	2	1235	0.024647	0.044534	0.019888
2	3	1235	0.033747	0.043725	0.009978
3	4	1235	0.044841	0.066397	0.021556
4	5	1236	0.060686	0.075243	0.014556
5	6	1235	0.082499	0.099595	0.017096
6	7	1235	0.110113	0.11498	0.004867
7	8	1235	0.141743	0.098785	-0.042958
8	9	1235	0.191468	0.162753	-0.028715
9	10	1236	0.422223	0.476537	0.054314

```
Test deciles — tree_calibrated
```

	decile	n	mean_pd	realized_rate	calibration_gap
0	1	3015	0.000628	0.0	-0.000628
1	2	2172	0.001706	0.001842	0.000136
2	3	1868	0.008714	0.00803	-0.000684
3	4	417	0.019835	0.021583	0.001748
4	5	1521	0.041369	0.032873	-0.008496
5	6	972	0.125823	0.082305	-0.043518
6	7	1266	0.390648	0.350711	-0.039937
7	8	1122	0.823176	0.7959	-0.027276

```
Reason codes — events newly activating during large PD jumps (test split)
```

	event	n_new_activation_in_jumps	share_of_jumps
3	evt_liq_squeeze	55	0.253456
4	evt_quick_squeeze	51	0.235023
5	evt_ebitdadrop	37	0.170507
6	evt_cfdrop	26	0.119816
0	evt_divcut	8	0.036866
1	evt_divsusp	6	0.027650
2	evt_divinit	2	0.009217

9.4 Response capability — predefined action rules under costs and capacity

We translate PDs and `evt_*` indicators into an operational policy with three actions:

- **Screen / Review** (capacity-limited): highest-risk firms warrant immediate attention.
- **Monitor more closely**: elevated risk, but not high enough for immediate screening.
- **No action**: routine monitoring only.

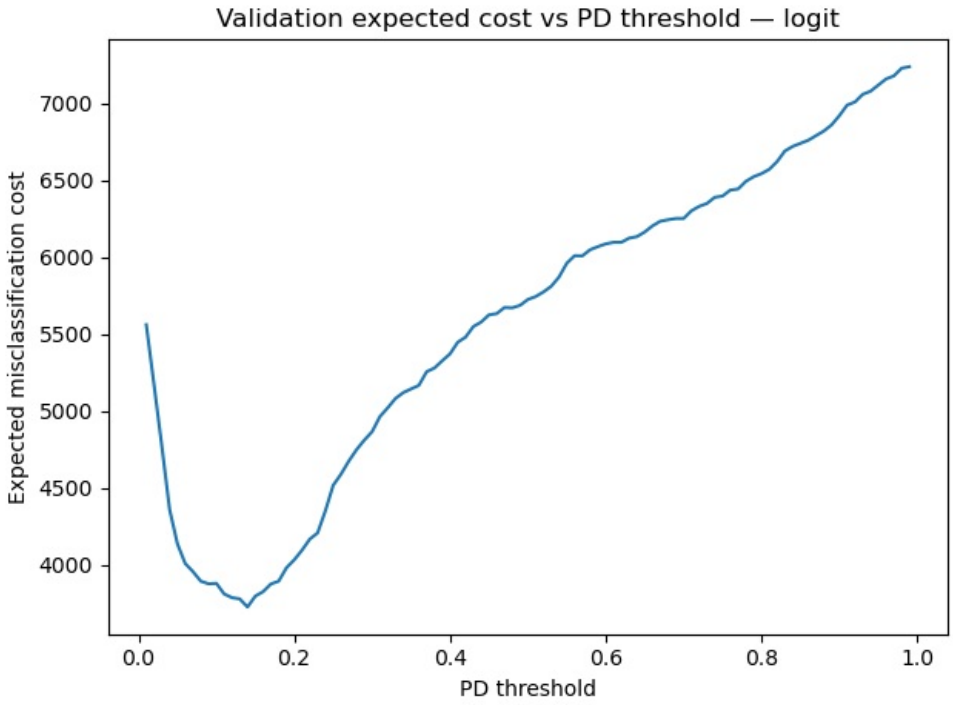
We compare:

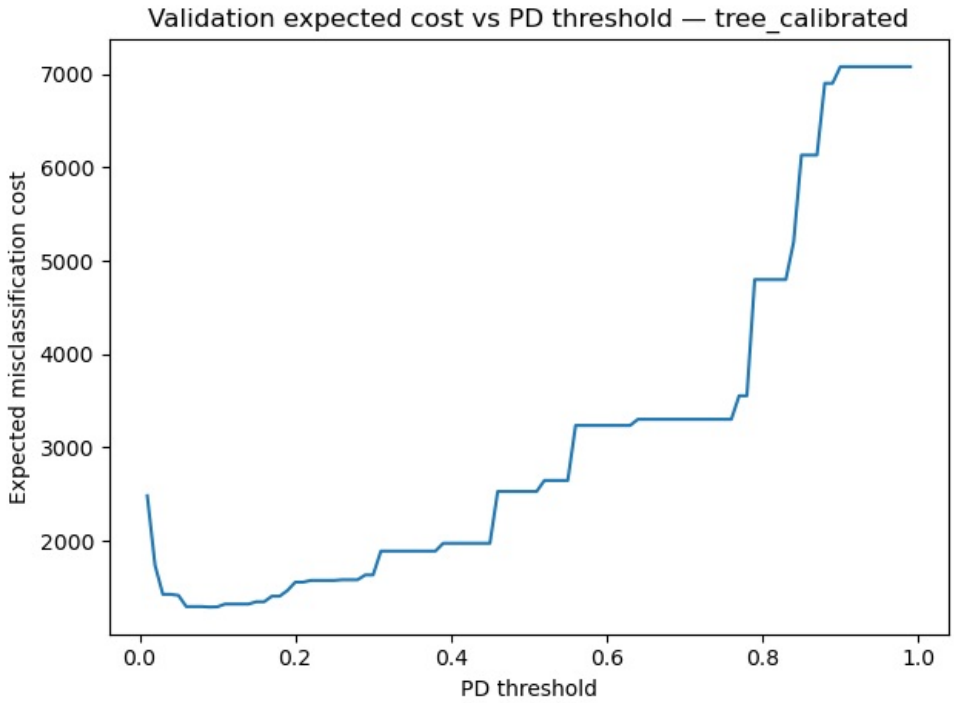
- **PD-only policy** (threshold on PD),
- **Hybrid policy** (PD + event burden) that can prioritize “indicator-led” cases without retraining the model.

In [111…

```
COST_FN = float(CONFIG["COST_FN"])
COST_FP = float(CONFIG["COST_FP"])
CAPACITY_PCT = float(CONFIG["CAPACITY_PCT"])
MONITOR_PCT = float(CONFIG.get("MONITOR_PCT", min(0.20, 2*CAPACITY_PCT)))  # fallback: monitor top 20% or 2x ca

def expected_cost(y_true: np.ndarray, y_hat: np.ndarray) -> float:
    tn, fp, fn, tp = confusion_matrix(y_true, y_hat).ravel()
    return COST_FN*fn + COST_FP*fp

def apply_pd_only_policy(p: np.ndarray, thr_screen: float, thr_monitor: float) -> dict:
    screen = (p >= thr_screen).astype(int)
    monitor = ((p >= thr_monitor) & (p < thr_screen)).astype(int)
    return {"screen": screen, "monitor": monitor}

def apply_hybrid_policy(p: np.ndarray, evt_count: np.ndarray, alpha: float = 0.05, beta: float = 0.10) -> dict:
    """Hybrid prioritization score:
      score = p + alpha*1{evt_any} + beta*1{evt_count>=2}
    Screening/monitoring are then capacity-based on the score."""
    evt_any = (evt_count > 0).astype(int)
    score = p + alpha*evt_any + beta*(evt_count >= 2).astype(int)
    thr_score_screen = float(np.quantile(score, 1-CAPACITY_PCT))
    thr_score_monitor = float(np.quantile(score, 1-MONITOR_PCT))
    screen = (score >= thr_score_screen).astype(int)
    monitor = ((score >= thr_score_monitor) & (score < thr_score_screen)).astype(int)
    return {"screen": screen, "monitor": monitor, "score": score, "thr_score_screen": thr_score_screen, "thr_sc

def build_evt_count(df_in: pd.DataFrame, evt_cols: list[str]) -> np.ndarray:
    evt_mat = df_in[evt_cols].apply(pd.to_numeric, errors="coerce")
    return (evt_mat.fillna(0) == 1).sum(axis=1).values

# --- Threshold selection (validation only) for PD-only policy ---
grid = np.linspace(0.01, 0.99, 99)
mask_val = df_model["split"] == "val"
y_val = df_model.loc[mask_val, TARGET_NAME].astype(int).values

thr_tbls = {}
for model_name, pcol in [("logit","pd_logit"), ("tree_calibrated","pd_tree")]:
    p_val = df_model.loc[mask_val, pcol].values

    # Cost-opt threshold
    costs = []
    for thr in grid:
        costs.append(expected_cost(y_val, (p_val >= thr).astype(int)))
    thr_cost_opt = float(grid[int(np.argmin(costs))])

    # Capacity and monitoring thresholds (operational)
    thr_capacity = float(np.quantile(p_val, 1-CAPACITY_PCT))
    thr_monitor = float(np.quantile(p_val, 1-MONITOR_PCT))

    thr_tbls[model_name] = {"thr_cost_opt": thr_cost_opt, "thr_capacity": thr_capacity, "thr_monitor": thr_moni

    plt.figure()
    plt.plot(grid, costs)
    plt.title(f"Validation expected cost vs PD threshold — {model_name}")
    plt.xlabel("PD threshold")
    plt.ylabel("Expected misclassification cost")
    plt.tight_layout()
    plt.savefig(Path(CONFIG["FIG_DIR"]) / f"cost_curve_{model_name}_val.png", dpi=160)
    plt.show()

display(pd.DataFrame(thr_tbls).T)

# --- Policy comparison on TEST: PD-only vs Hybrid (PD + events) ---
mask_test = df_model["split"] == "test"
y_test = df_model.loc[mask_test, TARGET_NAME].astype(int).values
evt_count_test = build_evt_count(df_model.loc[mask_test, :], EVT_COLS)

rows = []
for model_name, pcol in [("logit","pd_logit"), ("tree_calibrated","pd_tree")]:
    p_test = df_model.loc[mask_test, pcol].values

    thr_screen = thr_tbls[model_name]["thr_capacity"]
    thr_monitor = thr_tbls[model_name]["thr_monitor"]

    # PD-only (screen decision)
    polA = apply_pd_only_policy(p_test, thr_screen, thr_monitor)
    costA = expected_cost(y_test, polA["screen"])
```

```
    capA = float(polA["screen"].mean())
    tprA = float(((polA["screen"]==1) & (y_test==1)).sum() / max(1, (y_test==1).sum()))
    ppvA = float(((polA["screen"]==1) & (y_test==1)).sum() / max(1, (polA["screen"]==1).sum()))

    # Hybrid (screen decision derived from capacity on composite score)
    polB = apply_hybrid_policy(p_test, evt_count_test, alpha=0.05, beta=0.10)
    costB = expected_cost(y_test, polB["screen"])
    capB = float(polB["screen"].mean())
    tprB = float(((polB["screen"]==1) & (y_test==1)).sum() / max(1, (y_test==1).sum()))
    ppvB = float(((polB["screen"]==1) & (y_test==1)).sum() / max(1, (polB["screen"]==1).sum()))

    rows.append({
        "model": model_name,
        "policy": "PD-only",
        "screen_rate": capA,
        "monitor_rate": float(polA["monitor"].mean()),
        "tpr_screen": tprA,
        "ppv_screen": ppvA,
        "expected_cost": costA,
        "thr_screen_pd": thr_screen,
        "thr_monitor_pd": thr_monitor,
    })
    rows.append({
        "model": model_name,
        "policy": "Hybrid (PD + events)",
        "screen_rate": capB,
        "monitor_rate": float(polB["monitor"].mean()),
        "tpr_screen": tprB,
        "ppv_screen": ppvB,
        "expected_cost": costB,
        "thr_screen_score": polB["thr_score_screen"],
        "thr_monitor_score": polB["thr_score_monitor"],
        "alpha_evt_any": 0.05,
        "beta_evt_2plus": 0.10,
    })

policy_cmp = pd.DataFrame(rows).sort_values(["model","policy"])
print("\nPolicy comparison on TEST (screen decision):")
display(policy_cmp)
```





	thr_cost_opt	thr_capacity	thr_monitor
logit	0.14	0.153335	0.153335
tree_calibrated	0.09	0.106145	0.106145

```
Policy comparison on TEST (screen decision):
```

	model	policy	screen_rate	monitor_rate	tpr_screen	ppv_screen	expected_cost	thr_screen_pd	thr_monitor_pd	thr_scre
1	logit	Hybrid (PD + events)	0.200032	0.0	0.575251	0.348037	7961.0	NaN	NaN	
0	logit	PD-only	0.217923	0.0	0.543813	0.302006	8699.0	0.153335	0.153335	
3	tree_calibrated	Hybrid (PD + events)	0.203270	0.0	0.896321	0.533652	2721.0	NaN	NaN	
2	tree_calibrated	PD-only	0.253056	0.0	0.935786	0.447537	2687.0	0.106145	0.106145	

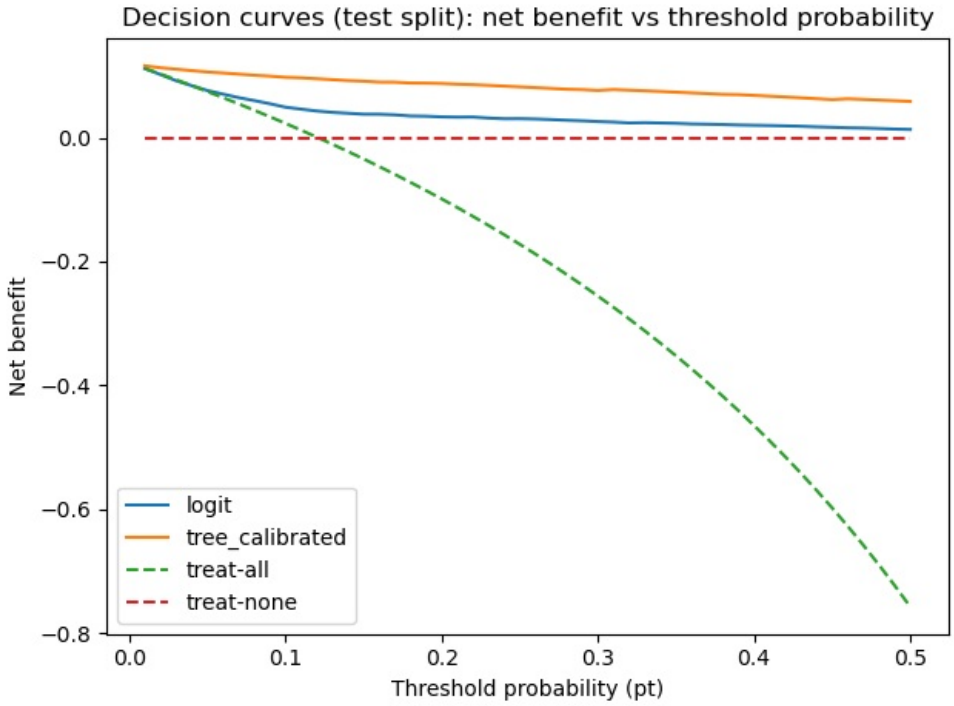
9.4.1 Decision curve analysis (net benefit)

Decision curves provide an alternative view of “response capability”: the net benefit of acting at different PD thresholds (treat-all vs treat-none baselines).

In [112…

```
def net_benefit(y_true: np.ndarray, p: np.ndarray, pt: float) -> float:
    y_hat = (p >= pt).astype(int)
    tn, fp, fn, tp = confusion_matrix(y_true, y_hat).ravel()
    n = len(y_true)
    w = pt/(1-pt)
    return (tp/n) - (fp/n)*w

mask = df_model["split"]=="test"
y_test_np = df_model.loc[mask, TARGET_NAME].astype(int).values
```

```
pts = np.linspace(0.01, 0.50, 50)
plt.figure()
for model_name, pcol in [("logit","pd_logit"), ("tree_calibrated","pd_tree")]:
    p = df_model.loc[mask, pcol].values
    nb = [net_benefit(y_test_np, p, pt) for pt in pts]
    plt.plot(pts, nb, label=model_name)

# Treat-all and treat-none baselines
event_rate = y_test_np.mean()
nb_all = [event_rate - (1-event_rate)*(pt/(1-pt)) for pt in pts]
nb_none = [0 for _ in pts]
plt.plot(pts, nb_all, linestyle="--", label="treat-all")
plt.plot(pts, nb_none, linestyle="--", label="treat-none")

plt.title("Decision curves (test split): net benefit vs threshold probability")
plt.xlabel("Threshold probability (pt)")
plt.ylabel("Net benefit")
plt.legend()
plt.tight_layout()
plt.savefig(Path(CONFIG["FIG_DIR"]) / "decision_curves_test.png", dpi=160)
plt.show()
```



9.5 Scenario analysis (accounting-consistent pro-forma adjustments; illustrative)

Scenario analysis is a **communication and response** aid: it shows how the PD changes under internally consistent accounting adjustments (illustrative; not causal).

In [113…

```
# Scenario engine: recompute a single-row feature vector using the same rules as the main pipeline.

def build_model_features_from_row(row: pd.Series) -> pd.DataFrame:
    # --- Helper for safe ratios in single row ---
    def safe_div(n, d):
        try:
            n_f = float(n)
            d_f = float(d)
            if pd.isna(n_f) or pd.isna(d_f) or d_f == 0:
                return np.nan
            return n_f / d_f
        except:
            return np.nan

    # Raw items from row
    at = row.get("at", np.nan)
    che = row.get("che", np.nan)
    act = row.get("act", np.nan)
    lct = row.get("lct", np.nan)
    aco = row.get("aco", np.nan)
    lco = row.get("lco", np.nan)
    rect = row.get("rect", np.nan)
    invt = row.get("invt", np.nan)
    recch = row.get("recch", np.nan)
    invch = row.get("invch", np.nan)
    txp = row.get("txp", np.nan)
```

```
    txditc = row.get("txditc", np.nan)
    lt = row.get("lt", np.nan)
    dlc = row.get("dlc", np.nan)
    dltt = row.get("dltt", np.nan)
    oibdp = row.get("oibdp", np.nan)
    dp = row.get("dp", np.nan)
    xint = row.get("xint", np.nan)
    ceq = row.get("ceq", np.nan)
    capx = row.get("capx", np.nan)
    ppent = row.get("ppent", np.nan)
    intan = row.get("intan", np.nan)
    oancf = row.get("oancf", np.nan)
    re = row.get("re", np.nan)
    caps = row.get("caps", np.nan)
    mibt = row.get("mibt", np.nan)
    niadj = row.get("niadj", np.nan)
    aqc = row.get("aqc", np.nan)
    prstkc = row.get("prstkc", np.nan)

    # Debt aggregate matching df logic
    d_vals = [v for v in [dlc, dltt] if pd.notna(v)]
    total_debt = sum(d_vals) if d_vals else np.nan

    # Map of all potential continuous features
    feat_map = {}
    feat_map["ln_at"] = np.log(at) if pd.notna(at) and at > 0 else np.nan
    feat_map["cash_at"] = safe_div(che, at)
    feat_map["current_ratio"] = safe_div(act, lct)
    feat_map["nwc_at"] = safe_div(act - lct, at)
    feat_map["aco_act"] = safe_div(aco, act)
    feat_map["lco_lct"] = safe_div(lco, lct)
    feat_map["rect_act"] = safe_div(rect, act)
    feat_map["invt_act"] = safe_div(invt, act)
    feat_map["recch_act"] = safe_div(recch, act)
    feat_map["invch_act"] = safe_div(invch, act)
    feat_map["txp_lct"] = safe_div(txp, lct)
    feat_map["txditc_at"] = safe_div(txditc, at)
    feat_map["lt_at"] = safe_div(lt, at)
    feat_map["dlc_at"] = safe_div(dlc, at)
    feat_map["dltt_at"] = safe_div(dltt, at)
    feat_map["debt_at"] = safe_div(total_debt, at)
    feat_map["st_debt_share"] = safe_div(dlc, total_debt)
    feat_map["ebitda_at"] = safe_div(oibdp, at)
    feat_map["dp_at"] = safe_div(dp, at)
    feat_map["xint_at"] = safe_div(xint, at)
    feat_map["interest_coverage"] = safe_div(oibdp, xint)
    feat_map["debt_to_ebitda"] = safe_div(total_debt, oibdp)
    feat_map["ebit_to_capital"] = safe_div(oibdp - dp, total_debt + ceq)
    feat_map["capx_at"] = safe_div(capx, at)

    # V2 extras
    feat_map["ppent_at"] = safe_div(ppent, at)
    feat_map["intan_at"] = safe_div(intan, at)
    feat_map["ocf_to_debt"] = safe_div(oancf, total_debt)
    feat_map["fcf_to_debt"] = safe_div(oancf - capx, total_debt)
    feat_map["re_at"] = safe_div(re, at)

    # V3 extras
    feat_map["ceq_at"] = safe_div(ceq, at)
    feat_map["caps_at"] = safe_div(caps, at)
    feat_map["mibt_at"] = safe_div(mibt, at)
    feat_map["niadj_at"] = safe_div(niadj, at)
    feat_map["xint_lct"] = safe_div(xint, lct)
    feat_map["aqc_at"] = safe_div(aqc, at)
    feat_map["prstkc_at"] = safe_div(prstkc, at)

    # Event map
    event_map = {}
    event_map["loss_indicator"] = 1.0 if pd.notna(niadj) and niadj < 0 else 0.0

    # Assemble raw feature vector
    out_dict = {}
    for c in continuous_feats_raw:
        out_dict[c] = feat_map.get(c, np.nan)
    for e in event_feats:
        out_dict[e] = event_map.get(e, 0)

    out = pd.DataFrame([out_dict])

    # Preprocessing: train medians, winsor, scaler -> z_
    for c in continuous_feats_raw:
        v = out[c].replace([np.inf, -np.inf], np.nan)
        v = v.fillna(train_medians[c])
```

```
        lo, hi = winsor_bounds[c]
        v = apply_bounds(v, lo, hi)
        out[c] = v

    Z = scaler.transform(out[continuous_feats_raw].astype(float))
    for j, c in enumerate(continuous_feats_raw):
        out[f"z_{c}"] = Z[:, j]

    # Final feature vector in MODEL_FEATS order
    return out[[f"z_{c}" for c in continuous_feats_raw] + event_feats]

def predict_pd_from_features(X_row: pd.DataFrame) -> dict:
    pd_logit = float(logit_clf.predict_proba(X_row)[:, 1][0])
    drow = xgb.DMatrix(X_row, feature_names=X_row.columns.tolist())
    pd_tree_raw = float(xgb_model.predict(drow)[0])
    pd_tree = float(iso.transform([pd_tree_raw])[0])
    return {"pd_logit": pd_logit, "pd_tree": pd_tree}

# Select a representative high-risk test observation
test_df = df_model.loc[df_model["split"]=="test", :].copy()
rep_idx = test_df["pd_logit"].idxmax()
row0 = df.loc[rep_idx, :]  # use df (feature-engineered, imputed), not df_model
base_X = build_model_features_from_row(row0)
base_pd = predict_pd_from_features(base_X)

print("Representative observation (highest logit PD in test):")
display(df_model.loc[rep_idx, ["firm_id","fyear","label_year","pd_logit","pd_tree","target_next_v1","target_nex
print("Base PDs:", base_pd)

# Scenario 1: Liquidity buffer to current ratio = 1.2 (increase current assets; illustrative)
row1 = row0.copy()
if "act" in row1.index and "lct" in row1.index and pd.notna(row1["act"]) and pd.notna(row1["lct"]) and row1["lc
    target_cr = 1.2
    add_act = max(0.0, target_cr*row1["lct"] - row1["act"])
    row1["act"] = row1["act"] + add_act
    if "che" in row1.index and pd.notna(row1.get("che", np.nan)):
        row1["che"] = row1["che"] + add_act  # assume added liquidity goes to cash
X1 = build_model_features_from_row(row1)
pd1 = predict_pd_from_features(X1)

# Scenario 2: CFO improvement of +10% of assets (accounting-consistent in the short-run is debatable; treat as
row2 = row0.copy()
if "oancf" in row2.index and "at" in row2.index and pd.notna(row2["at"]):
    delta = 0.10 * row2["at"]
    row2["oancf"] = (row2["oancf"] if pd.notna(row2.get("oancf", np.nan)) else 0.0) + delta
X2 = build_model_features_from_row(row2)
pd2 = predict_pd_from_features(X2)

scenario_tbl = pd.DataFrame([
    {"scenario":"base", **base_pd},
    {"scenario":"liquidity_buffer_CR_1.2", **pd1},
    {"scenario":"CFO_plus_10pct_assets", **pd2},
])
display(scenario_tbl)
```

```
Representative observation (highest logit PD in test):
firm_id              24730
fyear                 2022
label_year            2023
pd_logit          0.986111
pd_tree           0.876405
target_next_v1           1
target_next_v2           1
target_next_v3           0
Name: 42447, dtype: object
Base PDs: {'pd_logit': 0.9861114565643048, 'pd_tree': 0.8764045238494873}
```

	scenario	pd_logit	pd_tree
0	base	0.986111	0.876405
1	liquidity_buffer_CR_1.2	0.753698	0.556522
2	CFO_plus_10pct_assets	0.986111	0.876405

10. Results Summary & Interpretation Guardrails

10.1 Interpretation guardrails (publication-ready language)

- The label is a **constructed proxy** for balance-sheet/coverage stress; it is not a legal default outcome.
- Coefficients and SHAP values are **associational and predictive**, not causal effects.
- Even with leakage controls, residual mechanical endogeneity may remain because accounting choices jointly affect both predictors

and the proxy label.
- Attrition (missing next-year observations) can create sample-selection distortions; diagnostics are reported via `has_next_year_obs`.

10.2 Replication artifacts

The following tables/exports are written to `outputs/` for downstream paper workflow:

- `config_summary.json`
- `distress_rule.json`
- `event_dictionary.csv`
- `logit_inference_table.csv`
- `metrics_table.csv`
- `predictions.csv`

10.3 Export tables, thresholds, and predictions

In [114…

```
out_dir = Path(CONFIG["OUTPUT_DIR"])

# Config + distress rule
(out_dir / "config_summary.json").write_text(json.dumps(CONFIG, indent=2))
(out_dir / "distress_rule.json").write_text(json.dumps(DISTRESS_RULE, indent=2))

# Event dictionary
event_dict.to_csv(out_dir / "event_dictionary.csv", index=False)

# Logit inference table
infer_tbl.reset_index().to_csv(out_dir / "logit_inference_table.csv", index=False)

# Metrics table
metrics_tbl.to_csv(out_dir / "metrics_table.csv", index=False)

# Predictions export (replication-friendly)
export_cols = ["firm_id","gvkey","fyear","label_year","split","target_next_v1","target_next_v2","target_next_v3
export_cols = [c for c in export_cols if c in df_model.columns]
export_cols += [c for c in event_feats if c in df_model.columns]
pred_export = df_model[export_cols].copy()
pred_export.to_csv(out_dir / "predictions.csv", index=False)

print("Wrote artifacts to:", out_dir.resolve())
print_df(pred_export, n=10, name="predictions.csv preview")
```

```
Wrote artifacts to: /Users/test/Desktop/Test Models/AIinFinance/outputs

predictions.csv preview (top 10 rows):
```

	firm_id	gvkey	fyear	label_year	split	target_next_v1	target_next_v2	target_next_v3	pd_logit	pd_tree
0	10000	10000	2014	2015	train	0	0	0	0.073434	0.000754
1	10000	10000	2015	2016	train	0	0	0	0.087332	0.000754
2	10000	10000	2016	2017	train	0	0	0	0.071247	0.002415
3	10000	10000	2017	2018	train	0	0	0	0.070718	0.001706
4	10000	10000	2018	2019	train	0	0	0	0.069281	0.000754
5	10000	10000	2019	2020	train	0	0	0	0.079402	0.000754
6	10000	10000	2020	2021	train	0	0	0	0.062314	0.000754
7	10000	10000	2021	2022	val	0	0	0	0.055969	0.001706
8	10000	10000	2022	2023	test	0	0	0	0.036874	0.000754
9	10000	10000	2023	2024	test	0	0	0	0.033127	0.000754

10.4 Deployment and maintenance (future work)

This notebook produces a research-grade replication pipeline. For production use (not required for journal replication), a minimal MLOps extension would include:

- scheduled re-scoring and monitoring for drift in feature distributions and target prevalence,
- retraining triggers and versioned model registry,
- data validation contracts (schema + unit tests) for the upstream Compustat extraction process.