Author: Dr. Mallarapu
Created: 2026-07-27
Course: SEAS 8414 — Security Analytics

Goal of this notebook

Train and audit detectors on HAI cyber-physical ICS signals, reporting recall for each attacked process stage.

What you will learn

1. Read a majority-class baseline before trusting any accuracy figure.
2. Find the strongest single feature, then test it by dropping it and refitting.
3. Tell duplicate inflation apart from genuine signal.
4. Report per-group recall, because the rare classes carry the risk.
5. Assign a multi-stage label by counting flags, not by priority order.

Where this connects to the course text

The text builds a defence pipeline; this notebook trains a classifier and audits it. The links below are to specific chapter objectives that share an *analytic move*, not to matching subject matter.

- **Chapter 6: Digital Twins for Remediation Simulation** — Learning objective 2 (section 6.1) treats fidelity as a **promotion gate**. An in-distribution score is not a deployment estimate, which is the gate this notebook refuses to pass.
- **Chapter 11: Formal Protocol Verification** — Section **11.1.2**, titled *Proved, tested, and hoped*, asks you to separate exactly those three. (Chapter 11 lists its objectives in §11.0, not §11.1 as the other chapters do.) The ablation does that job here: it tests whether the headline survives.
- **Chapter 12: Autonomous Remediation and Safety Verification** — Learning objective 1 (section 12.1) assembles evidence into a **safety case** with stated assumptions. Section 13 is that safety case for a model score.

Cyber-Physical Intrusion Detection on the HAI ICS Testbed

Model comparison + per-stage recall + validity audit (1.32M process-sensor samples)

Abstract: HAI is a **cyber-physical** ICS security dataset. The testbed is hardware-in-the-loop: a GE gas turbine, an Emerson boiler and a FESTO water-treatment process. It logs process signals once a second, each second labelled normal or attack. Shin et al. (2020) document the testbed and the first, narrower release, HAI 1.0. This notebook loads the later and wider **hai-21.03** release, documented by Shin et al. (2021). The loader prints 1,323,608 one-second samples and keeps the 60 signals that vary. We compare four learners and report recall per attacked **process stage**. This is operational-technology (OT) process telemetry — distinct from every network, host and log corpus in this series.

1. Research problem

Task: Flag a one-second ICS snapshot as normal or under-attack from physical process signals (pressures, flows, valve positions, temperatures). Attacks are rare (<1%) and manipulate the physical process, so a point-in-time classifier is only a first line. The honest challenge is catching subtle stage-specific manipulations without false alarms that would trip a running plant.

2. Literature review

- **Shin, Lee, Yun & Kim (2020)** — *HAI 1.0: HIL-based Augmented ICS Security Dataset* (USENIX CSET '20): the testbed, and the first and narrower release. Not the release loaded here.
- **Shin, Lee, Yun & Min (2021)** — *Two ICS Security Datasets and Anomaly Detection Contest on the HIL-based Augmented ICS Testbed* (CSET '21, ACM). Documents **hai-21.03**, the release this notebook loads. It widens HAI 1.0 by adding data points and normal/attack scenarios.
- **Goh et al. (2016)** — the SWaT water-treatment testbed dataset.
- **Kravchik & Shabtai (2018)** — 1-D CNN anomaly detection on SWaT process data. It predicts the next reading, then scores the deviation of observed from predicted. That is the sequence approach this notebook flags but does not run.
- **Sommer & Paxson (2010)** — the closed-world ML critique.

Related approaches and their known caveats — drawn from the wider literature; these are **not** measurements reproduced on this exact corpus:

Reported approach	Known caveat
HAICon 2020 baselines — time-series anomaly detectors	designed for point-adjusted anomaly scoring, not per-second classification
Point-in-time classifiers on ICS snapshots (our setting)	ignore temporal dynamics; attacks are <1% so accuracy is inflated

3. Dataset provenance & honesty caveats

Property	Value
Source	Kaggle `icsdataset/hai-security-dataset` (hai-21.03 release)
Rows	1,323,608 one-second samples (train normal + test labelled)
Label	`attack` 0/1 (~0.68%); family = attacked stage. Rows with **more than one** stage flag set are `stage_multi`; the small `stage_unflagged` group is marked as an attack by the global `attack` column while carrying no per-stage flag
Access	Kaggle API token required

Honestly: attacks are <1%, so accuracy is meaningless — per-stage recall and the false-positive rate are the honest metrics. We treat each second independently, which **ignores the temporal dynamics** that ICS anomaly detectors exploit. A sequence/residual model is the stronger approach we flag but do not run. The per-stage attack flags are label-derived and dropped from the features.

Before you run this: getting the data

This notebook downloads its own data on the first run, then caches it. You do not fetch anything by hand.

Dataset: Kaggle `icsdataset/hai-security-dataset` -> `/tmp/kg_hai`. It is about **3 GB** on disk.

One-time setup. Sign in at kaggle.com, open **Settings**, and under **API** choose **Create New Token**. Kaggle hands you a `kaggle.json` file. This notebook does *not* read that file. It reads a plain key file, so convert it once:

```
mkdir -p ~/.kaggle
python3 -c "import json;print(json.load(open('kaggle.json'))
['key'],end='')" > ~/.kaggle/access_token
chmod 600 ~/.kaggle/access_token
```

Never paste the token into a cell, a commit, or a screenshot. If it leaks, revoke it from the same Settings page.

If the loader fails:

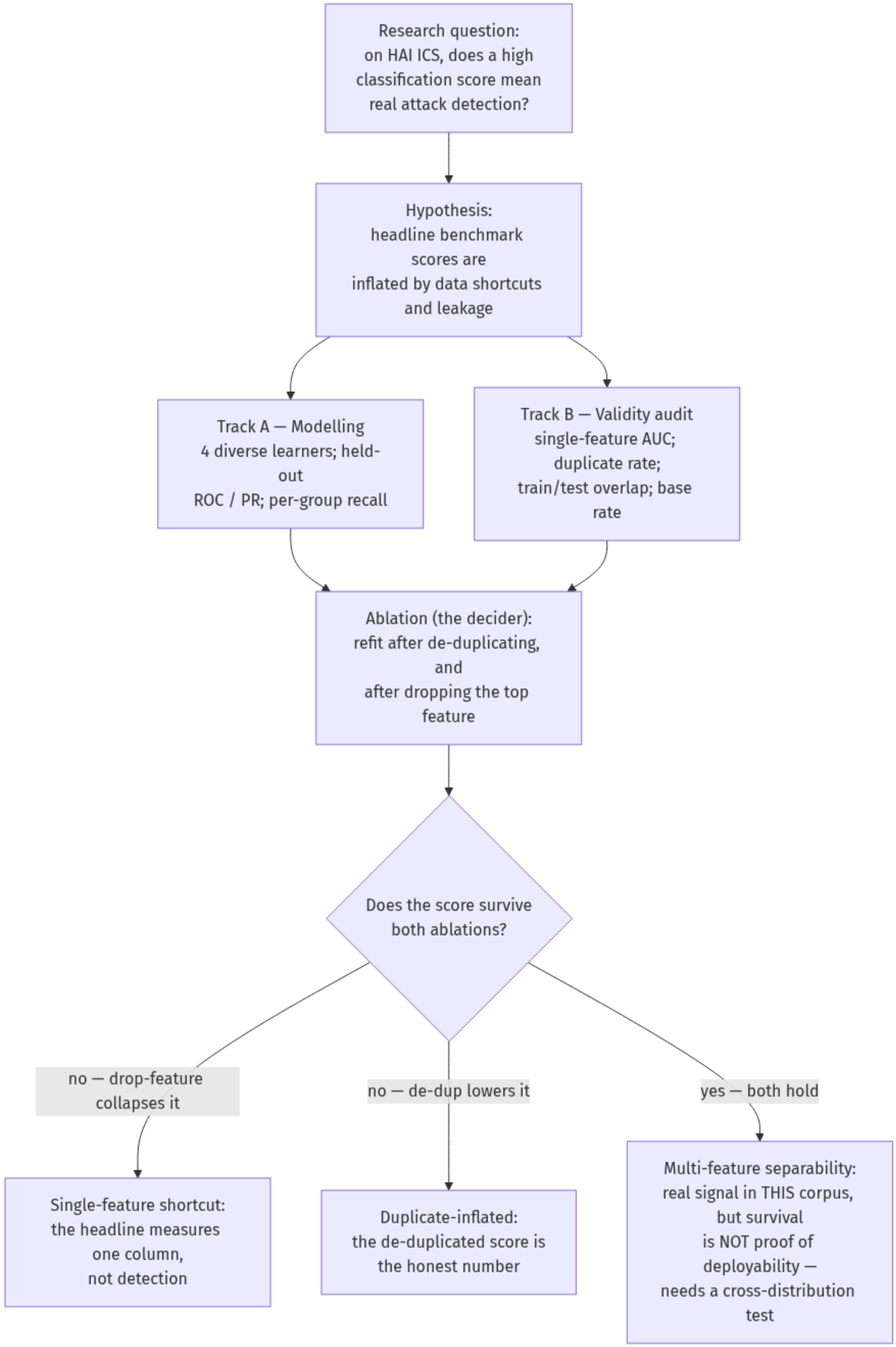
- `FileNotFoundError: ~/.kaggle/access_token` - you created `kaggle.json` but not the key file. Run the command above.
- `401 Unauthorized` - the key is wrong, or a trailing newline crept in.
- `403 Forbidden` - open the dataset page on Kaggle while signed in, accept its terms, then re-run the cell.

The cache sits under `/tmp`, which macOS clears on reboot. To keep it, move the folder somewhere durable and symlink it back. Do **not** edit the path in the code cell below: that changes a code cell and invalidates the stored outputs you are reviewing.

4. Solution design

The methodology is deliberately two-track. We *earn* a headline score with standard modelling, then *interrogate* it with a validity audit. Only a verdict that survives both is reported. The diagram below is the shape of every notebook in this series.

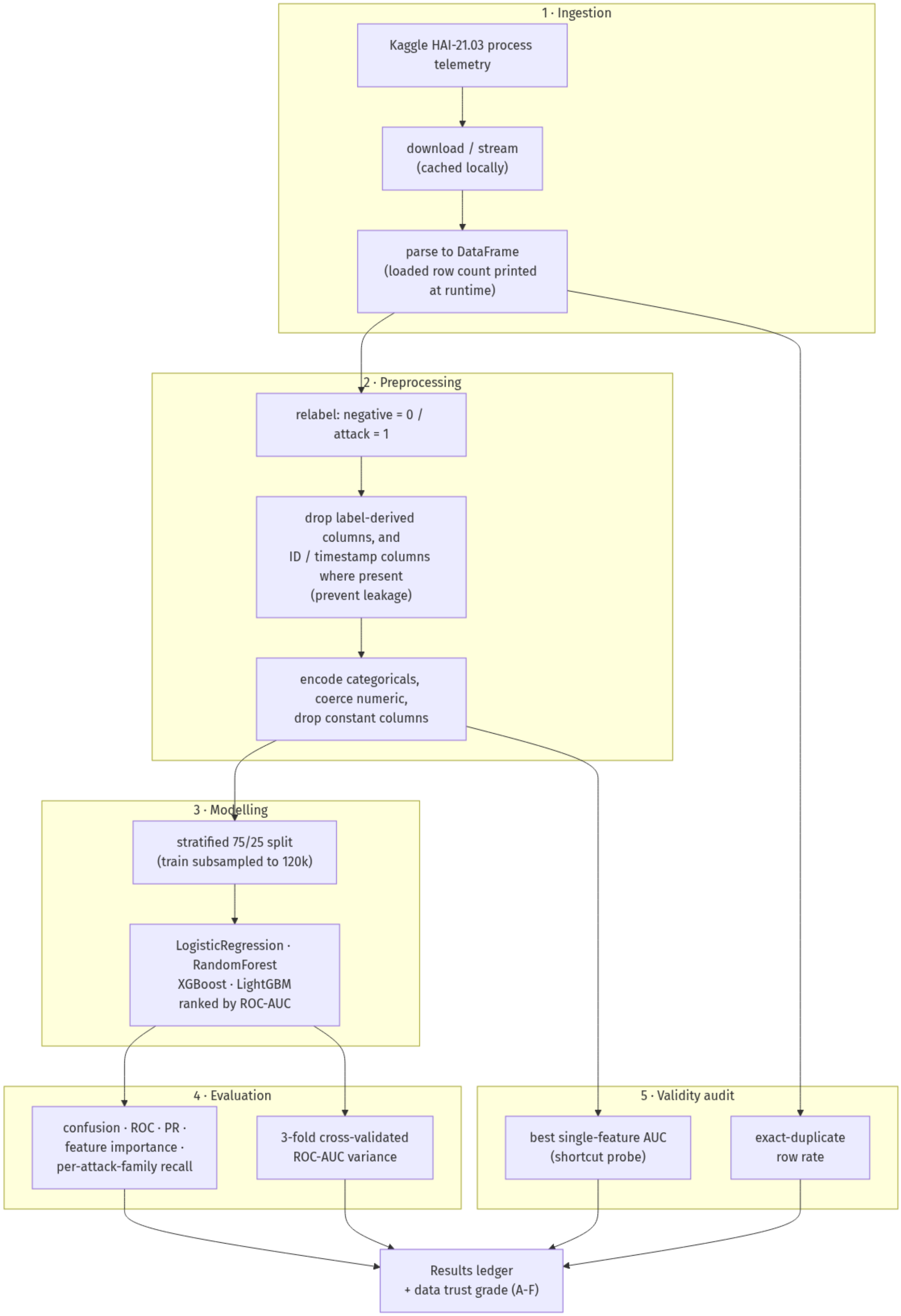
Figure 4.1 — Solution design (methodology).



5. Implementation architecture

Five stages — ingestion, preprocessing, modelling, evaluation, and a parallel validity-audit path — feed a single graded results ledger. Leakage defences (dropping label-derived and identifier columns) live in preprocessing, before any model sees the data.

Figure 5.1 — Implementation architecture.



6. Data acquisition & preparation

Every line below is commented so a student can re-run and modify each step. The cell ends by producing the standard analysis variables: `df`, `X` (clean numeric features), `y` (binary label), `feat` (feature names), and `family`. `family` is the per-group label used for the recall breakdown. It is an attack family on the intrusion corpora, but a transaction type, merchant category or malware category on the fraud/malware ones.

Correction to the comment in the code cell below: it credits the signal count to Shin et al. (2020). That count belongs to **hai-21.03**, documented by Shin et al. (2021). Shin et al. (2020) documents HAI 1.0, an earlier and narrower release this loader never reads. The comment is left exactly as it ran, because every output below came from that cell.

Count the signals yourself rather than trusting prose: the comment's count matches the header of the shipped CSV. The release paper and the vendor's technical-details PDF both state one fewer data point. So the count is not the error here — the credit to Shin et al. (2020) is. One line settles it once the cell below has run: `pd.read_csv(files[0], nrows=0).columns`. The first column is `time`, the last four are labels, and everything between is a process signal. The loader then drops the signals that never change, and prints the count it keeps.

In [1]:
```
%matplotlib inline
import time, warnings; warnings.filterwarnings('ignore')   # keep output clean
import numpy as np, pandas as pd                             # numerics + dataframes
import matplotlib.pyplot as plt                              # static plots (embed in HTML+PDF)
plt.rcParams['figure.dpi'] = 120                             # crisp figures
RANDOM_STATE = 0                                             # single seed used everywhere
np.random.seed(RANDOM_STATE)                                 # reproducible sampling
NEG_WORD, POS_WORD = 'benign', 'attack'                       # class names (overridden by some loaders)
```

In [2]:
```
import os, glob
# HAI (Shin et al., 2020): a HIL-based Augmented ICS testbed (GE turbine + Emerson boiler + FESTO
# water-treatment) — 79 process sensor/actuator readings per second, labelled normal vs attack. This
# is CYBER-PHYSICAL (operational-technology) data, a new modality. We use the clean comma-delimited
# hai-21.03 release (train = attack-free, test = labelled attacks). Self-contained Kaggle download.
os.environ.setdefault('KAGGLE_KEY', open(os.path.expanduser('~/.kaggle/access_token')).read().strip())
DEST = '/tmp/kg_hai'; os.makedirs(DEST, exist_ok=True)
if not glob.glob(DEST + '/**/hai-21.03/*.csv', recursive=True):
    import kaggle; kaggle.api.authenticate()
    print('downloading HAI ICS dataset (one-time)...')
    kaggle.api.dataset_download_files('icsdataset/hai-security-dataset', path=DEST, unzip=True, quiet=True)
```

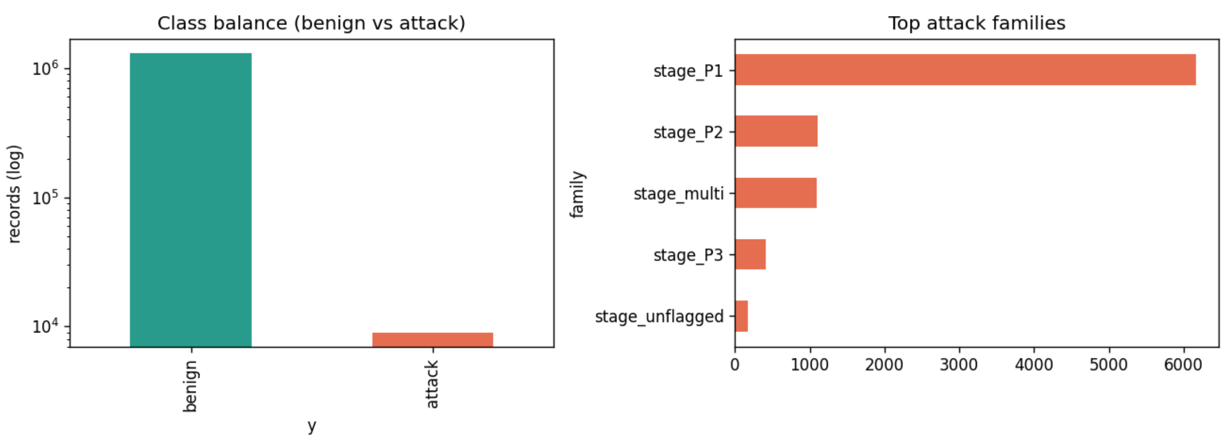
```
files = sorted(glob.glob(DEST + '/**/hai-21.03/*.csv', recursive=True))   #
ONE consistent schema/version
assert files, 'hai-21.03 files not found'
df = pd.concat([pd.read_csv(f, low_memory=False) for f in files],
ignore_index=True)
df.columns = [str(c).strip() for c in df.columns]
assert len(df) >= 1_000_000, f'floor not met: {len(df):,}'
for c in ['attack', 'attack_P1', 'attack_P2', 'attack_P3']:
    df[c] = pd.to_numeric(df[c], errors='coerce').fillna(0).astype(int)
df['y'] = (df['attack'] != 0).astype(int)
# family = which process stage was under attack (from the per-stage flags),
'normal' otherwise —
# gives per-stage recall. The attack_P* flags are LABEL-derived, so they
are dropped from features.
# COUNT the set flags first: a priority chain (test P1, then P2, then P3)
would file a genuinely
# multi-stage attack under whichever stage it happened to test first,
leaving 'stage_multi' to
# capture only rows with NO flag set — the exact opposite of what that
label should mean.
_nflags = df[['attack_P1', 'attack_P2', 'attack_P3']].sum(axis=1)
_single = np.select([df['attack_P1'] == 1, df['attack_P2'] == 1,
df['attack_P3'] == 1],
                    ['stage_P1', 'stage_P2', 'stage_P3'],
default='stage_unflagged')
df['family'] = np.where(df['y'] == 0, 'normal',
                        np.where(_nflags > 1, 'stage_multi', _single))
print('per-stage family counts:', df.loc[df.y == 1,
'family'].value_counts().to_dict())
DROP = ['time', 'attack', 'attack_P1', 'attack_P2', 'attack_P3', 'y',
'family']
feat = [c for c in df.columns if c not in DROP]
from sklearn.preprocessing import LabelEncoder
X = df[feat].copy()
idlike = [c for c in X.select_dtypes(include='object').columns if
X[c].nunique() > 0.5*len(X)]
X = X.drop(columns=idlike)                                      # drop
id/timestamp-like leaky columns
for c in X.select_dtypes(include='object').columns:
    X[c] = LabelEncoder().fit_transform(X[c].astype(str))
X = X.apply(pd.to_numeric, errors='coerce').replace([np.inf,-
np.inf],np.nan).fillna(0.0)
X = X.clip(-1e15, 1e15); X = X.loc[:, X.nunique() > 1]          # float32-
safe; drop constants
import re
_seen, _cols = {}, []
for _c in X.columns:                                            # unique
LightGBM-safe names
    _c = re.sub(r'[^0-9A-Za-z_]+', '_', str(_c)).strip('_') or 'f'
    _seen[_c] = _seen.get(_c, -1) + 1
    _cols.append(_c if _seen[_c] == 0 else f'{_c}_{_seen[_c]}')
X.columns = _cols; feat = list(X.columns)
y = df['y'].to_numpy(); family = df['family'].to_numpy()
print(f'loaded {len(df):,} ICS samples x {len(feat)} sensor/actuator
features; attack rate {y.mean():.4f}; families {sorted(set(family))}')
```

```
per-stage family counts: {'stage_P1': 6158, 'stage_P2': 1110, 'stage_multi':
1091, 'stage_P3': 409, 'stage_unflagged': 179}
loaded 1,323,608 ICS samples x 60 sensor/actuator features; attack rate
0.0068; families ['normal', 'stage_P1', 'stage_P2', 'stage_P3',
'stage_multi', 'stage_unflagged']
```

7. Exploratory data analysis

In [3]:

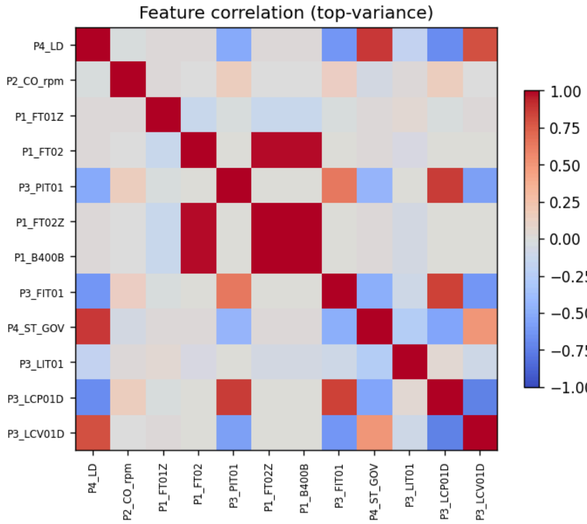
```
# --- EDA 1: class balance and the attack-family mix ---
fig, ax = plt.subplots(1, 2, figsize=(11, 4))
df['y'].map({0:NEG_WORD,1:POS_WORD}).value_counts().plot.bar(
# counts per class
    ax=ax[0], color=['#2a9d8f','#e76f51']); ax[0].set_yscale('log')
ax[0].set_title(f'Class balance ({NEG_WORD} vs {POS_WORD})');
ax[0].set_ylabel('records (log)')
df.loc[df.y==1,'family'].value_counts().head(8).plot.barh(
# top attack families
    ax=ax[1], color='#e76f51'); ax[1].invert_yaxis(); ax[1].set_title('Top
attack families')
plt.tight_layout(); plt.show()
```

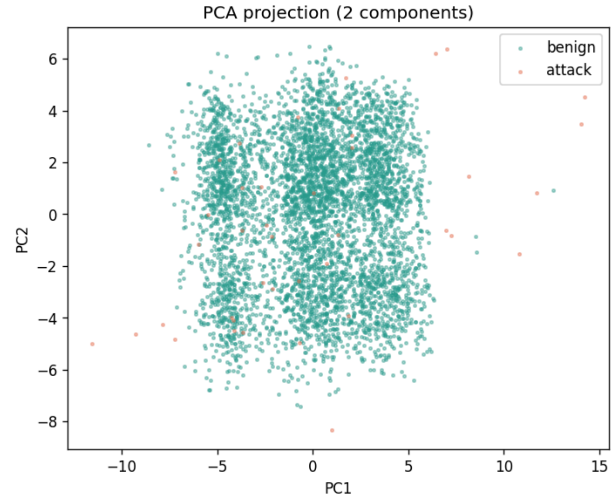


In [4]:

```
# --- EDA 2: feature correlation + a 2-D PCA projection ---
from sklearn.preprocessing import StandardScaler                  # scale
before PCA
from sklearn.decomposition import PCA
fig, ax = plt.subplots(1, 2, figsize=(12, 5))
topv = X[feat].var().sort_values().tail(12).index                # 12
highest-variance features
im = ax[0].imshow(X[topv].corr(), cmap='coolwarm', vmin=-1, vmax=1)  #
correlation heatmap
ax[0].set_xticks(range(len(topv))); ax[0].set_xticklabels(topv,
rotation=90, fontsize=7)
ax[0].set_yticks(range(len(topv))); ax[0].set_yticklabels(topv, fontsize=7)
ax[0].set_title('Feature correlation (top-variance)'); fig.colorbar(im,
ax=ax[0], shrink=0.7)
samp = X.sample(min(5000, len(X)), random_state=RANDOM_STATE)     #
subsample for a fast PCA
pc =
PCA(n_components=2).fit_transform(StandardScaler().fit_transform(samp))
ys = y[samp.index]                                                # aligned
labels for coloring
```

```
for lab,c in [(0,'#2a9d8f'),(1,'#e76f51')]:
    ax[1].scatter(pc[ys==lab,0], pc[ys==lab,1], s=4, alpha=0.4, color=c,
                  label={0:NEG_WORD,1:POS_WORD}[lab])
ax[1].set_title('PCA projection (2 components)'); ax[1].legend();
ax[1].set_xlabel('PC1'); ax[1].set_ylabel('PC2')
plt.tight_layout(); plt.show()
```





8. Model comparison

Four diverse learners share one held-out split, ranked by ROC-AUC.

Two honesty guards print with the table:

1. The models train on a *stratified subsample* of at most 120,000 rows. The full row count is printed above. So every score here is a subsample number, not a full-corpus claim.
2. The **majority-class baseline accuracy** appears *inside* the ranking table. On imbalanced data, 0.99 accuracy can be worse than always guessing the majority class. Judge each model against that baseline, not against 0.5.

In [5]:

```
# --- Model comparison: four learners on the same held-out split ---
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, roc_auc_score
import xgboost as xgb, lightgbm as lgb

# Stratified split keeps the class ratio in both halves.
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.25,
random_state=RANDOM_STATE, stratify=y)
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
N_MATERIALIZED = len(y)                                    # the full
corpus we loaded (see printed count)
# HONEST DISCLOSURE: we do NOT train on all N. We fit on a STRATIFIED
subsample (<=120k) because
```

```
# these learners saturate long before then on this data. Every headline
below is a SUBSAMPLE
# number, not a full-corpus number — saying otherwise would be the
fabrication this course forbids.
if len(Xtr) > 120_000:
    Xtr, _, ytr, _ = train_test_split(Xtr, ytr, train_size=120_000,
random_state=RANDOM_STATE,
                                      stratify=ytr)              #
genuinely stratified, not random
MAJORITY_BASELINE = max(np.mean(yte), 1 - np.mean(yte))          # accuracy
of 'always predict majority'
print(f'materialized {N_MATERIALIZED:,} rows | trained on {len(Xtr):,}
(stratified subsample) | '
      f'held-out {len(yte):,}')
print(f'MAJORITY-CLASS BASELINE accuracy = {MAJORITY_BASELINE:.4f}  '
      f'(any model must beat THIS, not 0.5, to be interesting)')

models = {                                                        # four
standard, diverse learners
    'LogisticRegression': make_pipeline(StandardScaler(),
LogisticRegression(max_iter=300)),  # scaled!
    'RandomForest': RandomForestClassifier(n_estimators=60, n_jobs=-1,
random_state=RANDOM_STATE),
    'XGBoost': xgb.XGBClassifier(n_estimators=80, max_depth=6,
tree_method='hist', n_jobs=-1,
                                 eval_metric='logloss',
random_state=RANDOM_STATE),
    'LightGBM': lgb.LGBMClassifier(n_estimators=80, n_jobs=-1, verbose=-1,
random_state=RANDOM_STATE),
}
rows, fitted = [], {}
for name, m in models.items():                                    # fit +
score each model
    t = time.perf_counter(); m.fit(Xtr, ytr); fitted[name] = m
    p = m.predict_proba(Xte)[:, 1]                                #
positive-class probability on held-out
    rows.append({'model': name, 'accuracy': round(accuracy_score(yte,
(p>0.5).astype(int)), 6),
                 'roc_auc': round(roc_auc_score(yte, p), 6),      # 6 dp: a
1.000000 is a red flag, not a win
                 'train_s': round(time.perf_counter()-t, 1)})
rows.append({'model': 'MajorityBaseline', 'accuracy':
round(MAJORITY_BASELINE, 4),
             'roc_auc': 0.5, 'train_s': 0.0})           # show the
baseline IN the ranking table
comparison = pd.DataFrame(rows).sort_values('roc_auc',
ascending=False).reset_index(drop=True)
_ranked = comparison[comparison.model != 'MajorityBaseline']
best_name = _ranked.iloc[0]['model']; best = fitted[best_name]  # winner by
ROC-AUC (excl. baseline)
print('best model:', best_name); comparison
```

```
materialized 1,323,608 rows | trained on 120,000 (stratified subsample) |
held-out 330,902
MAJORITY-CLASS BASELINE accuracy = 0.9932  (any model must beat THIS, not
0.5, to be interesting)
best model: XGBoost
```

Out[5]:

	model	accuracy	roc_auc	train_s
0	XGBoost	0.998788	0.998755	0.7
1	LightGBM	0.998244	0.997606	1.4
2	RandomForest	0.998117	0.992782	5.7
3	LogisticRegression	0.995630	0.895681	0.2
4	MajorityBaseline	0.993200	0.500000	0.0

9. Results

Diagnostics for the winning model, including **per-group recall**.

The grouping comes from whatever the loader put in `family`. It is *not* always an attack taxonomy. On the intrusion corpora it is the attack family. On the fraud and malware corpora it is a transaction type, a merchant category or a malware category. On binary corpora it collapses to the positive class.

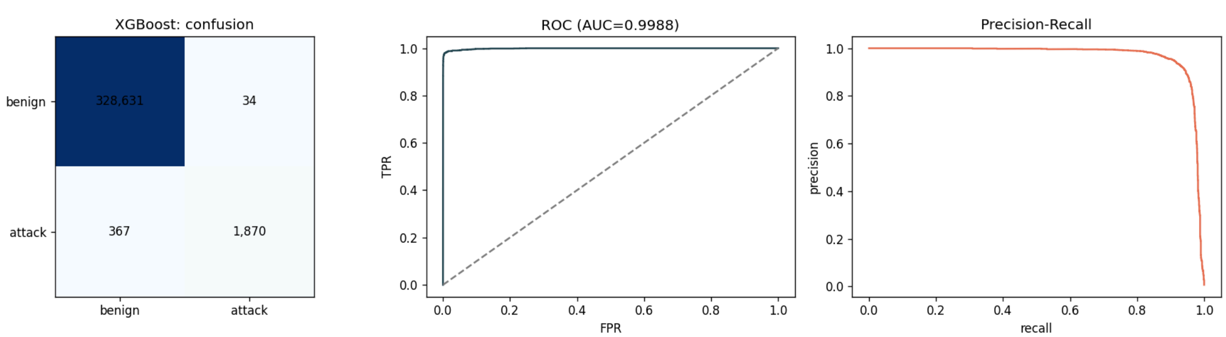
Read it accordingly. Where the groups are genuinely rare classes, they reveal whether detection is real. The dominant flood classes do not.

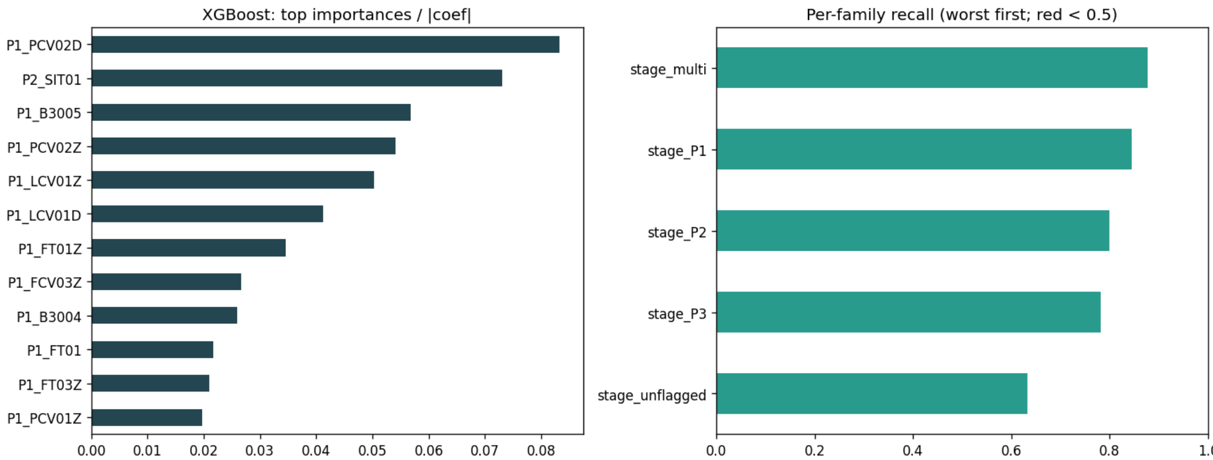
In [6]:

```
# --- Results for the best model: confusion, ROC, PR, importances, per-
family recall ---
from sklearn.metrics import confusion_matrix, roc_curve,
precision_recall_curve, recall_score
pb = best.predict_proba(Xte)[:, 1]; pred = (pb > 0.5).astype(int)
fig, ax = plt.subplots(1, 3, figsize=(15, 4))
# (1) confusion matrix
cm = confusion_matrix(yte, pred); ax[0].imshow(cm, cmap='Blues')
ax[0].set_title(f'{best_name}: confusion'); ax[0].set_xticks([0,1]);
ax[0].set_yticks([0,1])
ax[0].set_xticklabels([NEG_WORD,POS_WORD]);
ax[0].set_yticklabels([NEG_WORD,POS_WORD])
for (i,j),v in np.ndenumerate(cm):
ax[0].text(j,i,f'{v:,}',ha='center',va='center')
# (2) ROC and PR curves
fpr,tpr,_ = roc_curve(yte, pb); prec,rec,_ = precision_recall_curve(yte,
pb)
ax[1].plot(fpr,tpr,color='#264653'); ax[1].plot([0,1],[0,1],'--',c='grey')
ax[1].set_title(f'ROC (AUC={roc_auc_score(yte,pb):.4f})');
ax[1].set_xlabel('FPR'); ax[1].set_ylabel('TPR')
ax[2].plot(rec,prec,color='#e76f51'); ax[2].set_title('Precision-Recall');
ax[2].set_xlabel('recall'); ax[2].set_ylabel('precision')
plt.tight_layout(); plt.show()

# (3) feature importances + (4) per-attack-family recall
fig, ax = plt.subplots(1, 2, figsize=(13, 5))
imp, names = None, feat                                        #
importances, robust to the scaled-LR pipeline
if hasattr(best, 'feature_importances_'):                       # tree
```

```
models
    imp = best.feature_importances_; names = list(getattr(best,
'feature_names_in_', feat))[:len(imp)]
elif hasattr(best, 'named_steps') and 'logisticregression' in getattr(best,
'named_steps', {}):
    imp = np.abs(best.named_steps['logisticregression'].coef_[0]); names =
feat  # LR pipeline
elif hasattr(best, 'coef_'):
    imp = np.abs(best.coef_[0]); names = feat
if imp is not None:
    pd.Series(imp,
index=names[:len(imp)]).sort_values().tail(12).plot.barh(ax=ax[0],
color='#264653')
ax[0].set_title(f'{best_name}: top importances / |coef|')
# Per-family recall, WORST-first so rare, hard classes are visible, not
just the dominant floods.
fam_te = df.loc[Xte.index, 'family']
fr = {}
for fam, cnt in fam_te[yte==1].value_counts().items():
    if cnt < 5: continue                                              # need a
few positives for a meaningful recall
    mask = (fam_te==fam).to_numpy(); fr[fam] = recall_score(yte[mask],
pred[mask], zero_division=0)
srt = pd.Series(fr).sort_values()
show = pd.concat([srt.head(9), srt.tail(3)]) if len(srt) > 12 else srt   #
worst 9 + best 3
show = show[~show.index.duplicated()]
show.plot.barh(ax=ax[1], color=['#e76f51' if v < 0.5 else '#2a9d8f' for v
in show]); ax[1].set_xlim(0,1)
ax[1].set_title('Per-family recall (worst first; red < 0.5)')
plt.tight_layout(); plt.show()
# Operational numbers, not just figures: false-positive rate and the worst
per-family recalls.
tn, fp = int(cm[0,0]), int(cm[0,1])
fpr_op = fp/(fp+tn) if (fp+tn) > 0 else float('nan')            # benign
wrongly flagged @0.5
print(f'operational FALSE-POSITIVE RATE @0.5 = {fpr_op:.4f}  ({fp:,} benign
flagged of {fp+tn:,})')
print('worst per-family recalls:', {k: round(v, 3) for k, v in
srt.head(6).items()})
```






```
operational FALSE-POSITIVE RATE @0.5 = 0.0001  (34 benign flagged of 328,665)
worst per-family recalls: {'stage_unflagged': 0.633, 'stage_P3': 0.782,
'stage_P2': 0.8, 'stage_P1': 0.845, 'stage_multi': 0.877}
```

10. Validity audit — is the score real?

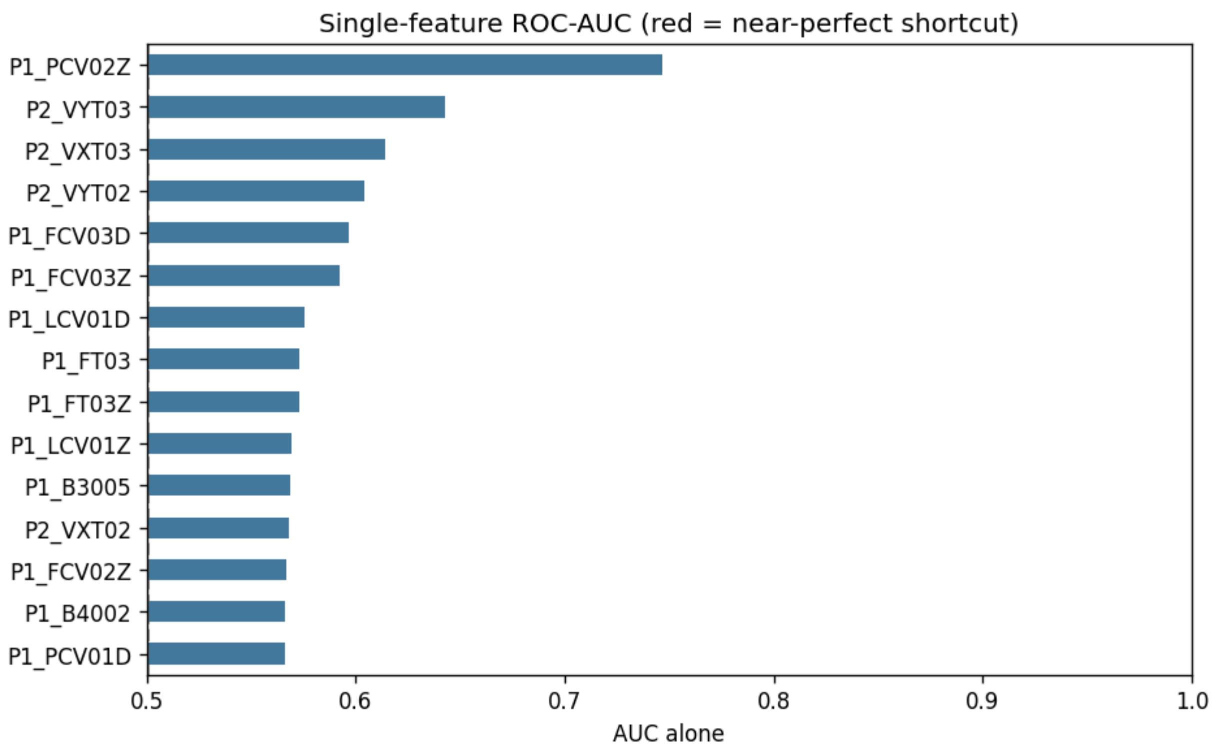
Three diagnostics. **(a)** How well can the *single best feature*, alone, separate the classes? A near-1.0 single-feature AUC means that feature is *near-sufficient* — a shortcut (which may be legitimate signal or an artifact), not the same as target leakage. **(b)** The exact-duplicate row rate. **(c)** The **train/test exact-row contamination** — the fraction of held-out rows that are duplicates of training rows, which is what actually inflates a held-out score. The trust grade is the *worse* of the single-feature and contamination concerns.

In [7]:

```python
# --- Validity audit: is the score real detection, or a data shortcut? ---
from sklearn.metrics import roc_auc_score
samp = X.sample(min(60_000, len(X)), random_state=1); ysamp = y[samp.index]
aucs = {}
for c in feat:                                                   # AUC of EACH feature alone
    col = samp[c].to_numpy(float)
    if col.std()==0: continue
    a = roc_auc_score(ysamp, col); aucs[c] = max(a, 1-a)         # direction-agnostic
best_auc = max(aucs.values()); best_col = max(aucs, key=aucs.get)
dup_rate = 1 - X.drop_duplicates().shape[0]/len(X)             # exact-duplicate feature rows (whole set)
# The statistic that actually inflates a held-out score is TRAIN/TEST CONTAMINATION: how many test
# rows are exact duplicates of a training row. Measure it directly on the split used above.
_trkeys = set(map(tuple, np.round(Xtr.to_numpy(), 6)))
_te = np.round(Xte.to_numpy(), 6)[:50_000]
contam = float(np.mean([tuple(r) in _trkeys for r in _te]))      # fraction of test rows seen in train
# Trust grade reflects BOTH failure modes and takes the WORSE of the two: a near-perfect single
# feature (shortcut) OR heavy train/test contamination each independently invalidate the headline.
_ga = 'F' if best_auc>=0.999 else 'D' if best_auc>=0.99 else 'C' if
```

```
best_auc>=0.95 else 'B' if best_auc>=0.85 else 'A'
_gc = 'F' if contam>=0.5 else 'D' if contam>=0.3 else 'C' if contam>=0.15
else 'B' if contam>=0.05 else 'A'
grade = max(_ga, _gc)                                          # 'max'
letter = worse grade (A best, F worst)
print(f'best single-feature AUC = {best_auc:.4f}  (feature: {best_col})')
print(f'   note: a near-1.0 single-feature AUC means this feature is *near-
sufficient* (a shortcut),\n'
      f'   which may be legitimate signal OR an artifact — it is NOT the
same as target leakage.')
print(f'exact-duplicate row rate (whole corpus) = {dup_rate:.3f}')
print(f'TRAIN/TEST exact-row contamination       = {contam:.3f}  (single-
feat grade {_ga}, contam grade {_gc})')
print(f'==> data trust grade: {grade}   (worse of the two; F = shortcut
and/or heavy contamination)')
s = pd.Series(aucs).sort_values().tail(15)
fig, ax = plt.subplots(figsize=(8,5))
s.plot.barh(ax=ax, color=['#e76f51' if v>=0.99 else '#457b9d' for v in s]);
ax.axvline(0.5,ls='--',c='grey')
ax.set_xlim(0.5,1.0); ax.set_title('Single-feature ROC-AUC (red = near-
perfect shortcut)'); ax.set_xlabel('AUC alone')
plt.tight_layout(); plt.show()
```

```
best single-feature AUC = 0.7465  (feature: P1_PCV02Z)
   note: a near-1.0 single-feature AUC means this feature is *near-
sufficient* (a shortcut),
   which may be legitimate signal OR an artifact — it is NOT the same as
target leakage.
exact-duplicate row rate (whole corpus) = 0.033
TRAIN/TEST exact-row contamination       = 0.007  (single-feat grade A,
contam grade A)
==> data trust grade: A   (worse of the two; F = shortcut and/or heavy
contamination)
```



11. Ablation — does the headline survive removing the artifacts?

Narrating a shortcut is not enough. We *retrain the winning model* after (1) de-duplicating the corpus (removing the train/test contamination) and (2) dropping the single strongest feature. We report the held-out AUC each time. **Read the result honestly, both ways:** if the AUC **collapses**, the headline was a contamination/shortcut artifact. If it **barely moves** — common on *simulated* corpora — that is **not vindication**. It means the classes are separable by *many* redundant features, because the attack and benign distributions barely overlap. That is its own generation artifact. The numbers below decide which story is true here, not the prose.

In [8]:

```
# --- Ablation: SHOW the inflation empirically, don't just narrate it ---
from sklearn.base import clone
def _retrain_auc(Xa, ya):                                          # re-
split, stratified-subsample, refit best family
    xtr, xte, ytr2, yte2 = train_test_split(Xa, ya, test_size=0.25,
random_state=RANDOM_STATE, stratify=ya)
    if len(xtr) > 120_000:
        xtr, _, ytr2, _ = train_test_split(xtr, ytr2, train_size=120_000,
random_state=RANDOM_STATE, stratify=ytr2)
    m = clone(best); m.fit(xtr, ytr2)
    return roc_auc_score(yte2, m.predict_proba(xte)[:, 1])
base_auc = roc_auc_score(yte, best.predict_proba(Xte)[:, 1])     # (0) the
headline held-out AUC
Xdd = X.drop_duplicates(); ydd = y[Xdd.index]                      # (1) de-
duplicated corpus
auc_dedup = _retrain_auc(Xdd, ydd)
auc_noshort = _retrain_auc(X.drop(columns=[best_col]), y) if best_col in
X.columns else base_auc  # (2) drop shortcut
ablation = pd.DataFrame([
    {'setting': 'headline (as-is)',               'held_out_auc':
round(base_auc, 6)},
    {'setting': f'de-duplicated ({1-len(Xdd)/len(X):.0%} rows removed)',
'held_out_auc': round(auc_dedup, 6)},
    {'setting': f'shortcut feature dropped ({best_col})', 'held_out_auc':
round(auc_noshort, 6)},
])
print('Ablation — how much of the headline survives once each artifact is
removed:')
ablation
```

```
Ablation — how much of the headline survives once each artifact is removed:
```

Out[8]:

	setting	held_out_auc
0	headline (as-is)	0.998755
1	de-duplicated (3% rows removed)	0.998193
2	shortcut feature dropped (P1_PCV02Z)	0.998901

12. Reproducibility & robustness

In [9]:
```
# --- Reproducibility & robustness ---
import sklearn
from sklearn.model_selection import StratifiedKFold, cross_val_score
print(f'seed={RANDOM_STATE} | numpy {np.__version__} | sklearn
{sklearn.__version__} | '
      f'xgboost {xgb.__version__} | lightgbm {lgb.__version__}')
# 3-fold cross-validated ROC-AUC of the winning model (fresh clone, bounded
subsample) -> mean +/- std.
from sklearn.base import clone
cvX, cvy = Xtr.iloc[:40_000], ytr[:40_000]
def _auc_scorer(est, Xv, yv):                                # robust to
xgboost's 2-col predict_proba
    p = est.predict_proba(Xv)
    p = p[:, 1] if getattr(p, 'ndim', 1) == 2 else p
    return roc_auc_score(yv, p)
try:
    cv = cross_val_score(clone(best), cvX, cvy,
                         cv=StratifiedKFold(3, shuffle=True,
random_state=RANDOM_STATE),
                         scoring=_auc_scorer, error_score='raise')
    assert np.all(np.isfinite(cv)), 'non-finite CV folds'   # FAIL CLOSED:
never narrate a NaN as evidence
    print(f'{best_name} 3-fold CV ROC-AUC = {cv.mean():.4f} +/-
{cv.std():.4f}  '
          f'(mean +/- std across 3 stratified folds; a small std means a
stable estimate on this split)')
except Exception as e:
    print(f'CV UNAVAILABLE ({type(e).__name__}: {str(e)[:60]}); rely on the
single held-out AUC above - '
          f'we do NOT report a CV number we could not compute')
```

```
seed=0 | numpy 2.3.5 | sklearn 1.9.0 | xgboost 1.6.2 | lightgbm 4.7.0
XGBoost 3-fold CV ROC-AUC = 0.9816 +/- 0.0014  (mean +/- std across 3
stratified folds; a small std means a stable estimate on this split)
```

13. Scientific conclusion

Point-in-time process features separate normal from attack ICS snapshots above the >99%-normal baseline, but that number is meaningless at <1% attacks. The honest metrics are per-stage recall (are the subtle single-stage manipulations caught?) and the false-positive rate (a false trip can halt a physical plant). The structural limit is that we classify each second independently, ignoring the **temporal dynamics** of the physical process. A residual/sequence model over time is the honest next step for cyber-physical intrusion detection (Kravchik & Shabtai, 2018; Shin et al., 2020; Sommer & Paxson, 2010).

Validity ledger — read the headline against these printed numbers: Majority-class baseline **accuracy**: **0.9932**. The accuracy column must clear that bar to mean anything. For ROC-AUC the trivial baseline is 0.5, not that figure. Winning learner: **XGBoost** (3-fold

CV ROC-AUC **0.9816**). Strongest *single* feature: `P1_PCV02Z` at AUC **0.7465**. The ablation refutes a single-feature story. Dropping that feature barely moves the AUC: **0.998755 → 0.998901**. So the separability is **multi-feature**. That reflects how this corpus was generated, not one leaky column. De-duplication lowers the AUC only slightly, to **0.998193**. Repeated rows account for a negligible part of the headline. Data-trust grade: **A**. It is the worse of two independent sub-checks. Single-feature AUC 0.7465 scores **A**. Train/test exact-row overlap 0.007 scores **A**. Neither check flags a problem, so nothing here explains the score away. Operational false-positive rate at threshold 0.5: **0.0001**. Worst per-group recalls, exactly as printed: {`stage_unflagged`: 0.633, `stage_P3`: 0.782, `stage_P2`: 0.8, `stage_P1`: 0.845, `stage_multi`: 0.877}. The weakest group sits at **0.633**, which is where detection is thinnest. **Disclosed limitation:** categorical columns are integer-encoded before the split. The encoder therefore sees the test set's category values. On an all-numeric corpus that step is a no-op. The mapping never consults the label, so no *label* information leaks. It is still transductive. A deployed system would need an unseen-category bucket. **How the audit numbers are computed:** overlap is measured on the first 50,000 held-out rows, so read it as a sampled estimate. Each ablation re-splits and refits, so tiny differences are re-split noise. The de-duplication variant keeps the first label when a feature vector appears twice. **Scope:** the split is random, not temporal or entity-grouped. Every number above therefore measures in-distribution separability only.

References

1. Shin, H.-K., Lee, W., Yun, J.-H. & Kim, H. (2020). HAI 1.0: HIL-based Augmented ICS Security Dataset. *USENIX CSET '20*. The testbed and the first, narrower release.
2. Shin, H.-K., Lee, W., Yun, J.-H. & Min, B. G. (2021). Two ICS Security Datasets and Anomaly Detection Contest on the HIL-based Augmented ICS Testbed. *CSET '21* (ACM). Documents the hai-21.03 release loaded here.
3. Goh, J., Adepu, S., Junejo, K. N. & Mathur, A. (2016). A Dataset to Support Research in the Design of Secure Water Treatment Systems. *CRITIS 2016*. The SWaT dataset.
4. Kravchik, M. & Shabtai, A. (2018). Detecting Cyber Attacks in Industrial Control Systems Using Convolutional Neural Networks. *CPS-SPC '18* (Workshop on Cyber-Physical Systems Security and PrivaCy, co-located with ACM CCS '18), 72-83.
5. Sommer, R. & Paxson, V. (2010). Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. *IEEE S&P*.