SRE Metrics Intelligence: Anomaly Detection + Forecasting (AWS CloudWatch)

Project goal

In production systems, SRE and backend teams monitor time-series metrics (CPU, network, request counts, latency) to detect incidents early and forecast capacity trends. This notebook builds a small but resume-grade “metrics intelligence” pipeline using a real AWS CloudWatch metric time series.

We will:

1. Load an AWS CloudWatch CPU utilization metric (5-minute sampling).
2. Engineer simple statistical features for anomaly detection.
3. Detect anomalies using a baseline rule + an unsupervised ML model (Isolation Forest).
4. Forecast short-term metric values using a Ridge regression model with lag features.
5. Save artifacts (plots, CSVs, JSON metrics, serialized models) for reproducibility and GitHub.

> Note: This dataset does not include ground-truth anomaly labels in our local download, so anomaly detection is evaluated qualitatively (visual inspection + ranked anomaly scores) rather than precision/recall against labeled windows.

In [1]:
```
# Import + Folders
import json
from pathlib import Path

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.ensemble import IsolationForest
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_absolute_error, mean_squared_error
import joblib

RANDOM_SEED = 42
```

```
PROJECT_ROOT = Path.cwd()
DATA_DIR = PROJECT_ROOT / "data"
ARTIFACTS_DIR = PROJECT_ROOT / "artifacts"
IMAGES_DIR = PROJECT_ROOT / "images"

for d in [DATA_DIR, ARTIFACTS_DIR, IMAGES_DIR]:
    d.mkdir(parents=True, exist_ok=True)

print("Ready:", DATA_DIR, ARTIFACTS_DIR, IMAGES_DIR)
```

```
Ready: /Users/shivesh/Desktop/PythonProject/SRE Metrics Intelligence /d
ata /Users/shivesh/Desktop/PythonProject/SRE Metrics Intelligence /arti
facts /Users/shivesh/Desktop/PythonProject/SRE Metrics Intelligence /im
ages
```

1) Dataset location and reproducibility

This project assumes the dataset is stored locally inside the repository under:

- `archive/realAWSCloudwatch/realAWSCloudwatch/`

We select a single metric file:

- `ec2_cpu_utilization_53ea38.csv`

We also create standard project output folders:

- `data/` for cleaned/resampled data exports
- `artifacts/` for metrics + model files
- `images/` for saved plots

This structure makes the project easy to rerun on another machine and easy for recruiters to inspect.

In [2]:
```
# Dataset paths
NAB_ROOT = PROJECT_ROOT / "archive"
AWS_DIR = NAB_ROOT / "realAWSCloudwatch" / "realAWSCloudwatch"

assert NAB_ROOT.exists(), f"Not found: {NAB_ROOT}"
assert AWS_DIR.exists(), f"Not found: {AWS_DIR}"

SERIES_PATH = AWS_DIR / "ec2_cpu_utilization_53ea38.csv"
assert SERIES_PATH.exists(), f"Not found: {SERIES_PATH}"

print("SERIES_PATH:", SERIES_PATH)
```

```
SERIES_PATH: /Users/shivesh/Desktop/PythonProject/SRE Metrics Intellige
nce /archive/realAWSCloudwatch/realAWSCloudwatch/ec2_cpu_utilization_53
ea38.csv
```

2) Load the metric CSV

We load the CSV into a Pandas DataFrame and enforce:

- `timestamp` parsed as timezone-aware UTC datetimes
- `value` converted to numeric
- Sorted time index

This is basic but critical: many real-world data issues (string timestamps, missing rows, incorrect ordering) silently break time-series modeling.

In [3]:
```
# Load CSV
def load_metric_csv(path: Path) -> pd.DataFrame:
    df = pd.read_csv(path)

    # Most files are: timestamp,value
    cols = [c.lower() for c in df.columns]
    ts_col = df.columns[cols.index("timestamp")] if "timestamp" in col
    val_col = df.columns[cols.index("value")] if "value" in cols else

    out = df[[ts_col, val_col]].copy()
    out.columns = ["timestamp", "value"]

    out["timestamp"] = pd.to_datetime(out["timestamp"], errors="coerce
    out["value"] = pd.to_numeric(out["value"], errors="coerce")
    out = out.dropna().sort_values("timestamp").set_index("timestamp")
    return out

ts = load_metric_csv(SERIES_PATH)
ts.head(), ts.shape
```

Out[3]:
```
(                           value
 timestamp
 2014-02-14 14:30:00+00:00  1.732
 2014-02-14 14:35:00+00:00  1.732
 2014-02-14 14:40:00+00:00  1.960
 2014-02-14 14:45:00+00:00  1.732
 2014-02-14 14:50:00+00:00  1.706,
 (4032, 1))
```

3) Visual sanity check (raw time series)

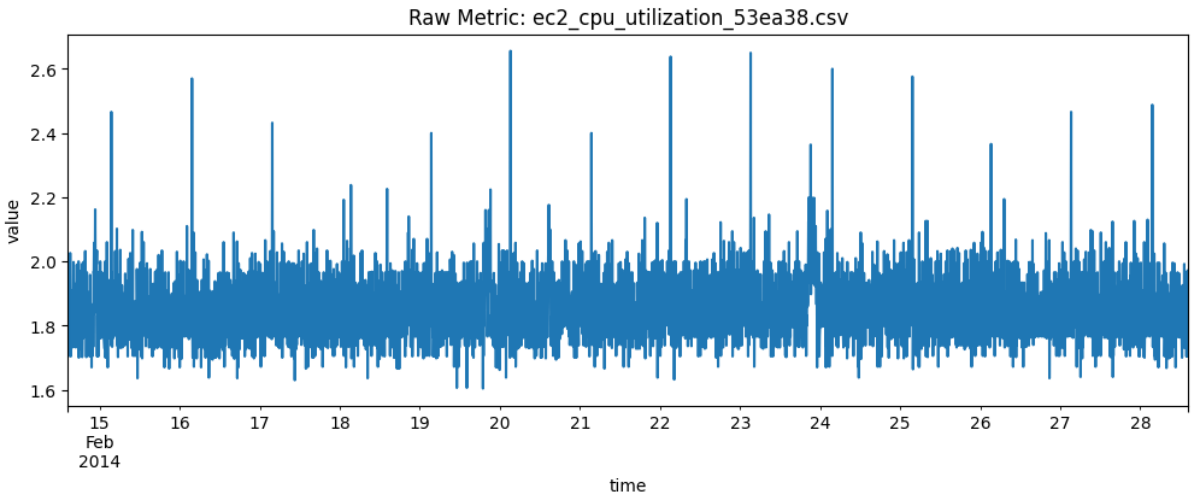
Before feature engineering, we plot the raw metric. This step helps us quickly spot:

- spikes/outliers
- missing data

- non-stationary drift
- obvious periodic patterns

This is a standard “SRE-style” step: always look at the signal before trusting models.

In [4]:
```
# Plot raw series
fig, ax = plt.subplots(figsize=(12, 4))
ts["value"].plot(ax=ax)
ax.set_title(f"Raw Metric: {SERIES_PATH.name}")
ax.set_xlabel("time")
ax.set_ylabel("value")
plt.show()
```



4) Resample to a regular frequency (5 minutes)

Time-series models generally assume a regular sampling rate. Even if the dataset is mostly regular, small gaps can exist.

We compute the median time delta and resample to a fixed interval (here, **5 minutes**), using:

- mean aggregation
- time interpolation to fill small gaps

This yields a clean, uniform series suitable for both anomaly detection and forecasting.

In [5]:
```
# Resample to regular frequency
delta = ts.index.to_series().diff().median()
minutes = max(1, int(round(delta.total_seconds() / 60))) if pd.notna(d
rule = f"{minutes}min"
print("Median delta:", delta, "| Resample:", rule)
```

```
series = ts["value"].resample(rule).mean().interpolate("time")
df = pd.DataFrame({"value": series}, index=series.index)

df.head(), df.shape
```

```
Median delta: 0 days 00:05:00 | Resample: 5min
```

Out[5]:

```
(                           value
 timestamp
 2014-02-14 14:30:00+00:00  1.732
 2014-02-14 14:35:00+00:00  1.732
 2014-02-14 14:40:00+00:00  1.960
 2014-02-14 14:45:00+00:00  1.732
 2014-02-14 14:50:00+00:00  1.706,
 (4032, 1))
```

5) Feature engineering for anomaly detection

We build lightweight features commonly used in monitoring pipelines:

- `value` : the metric itself
- `diff` : change since the previous timestamp (captures sudden jumps)
- `roll_mean` : rolling mean (local baseline)
- `roll_std` : rolling standard deviation (local volatility)

These features are deliberately simple but effective for many operational metrics.

In [6]:

```
# Feature engineering (for anomaly detection)
def build_features(series: pd.Series, window=12) -> pd.DataFrame:
    X = pd.DataFrame({"value": series})
    X["diff"] = X["value"].diff()
    X["roll_mean"] = X["value"].rolling(window).mean()
    X["roll_std"] = X["value"].rolling(window).std()
    X = X.dropna()
    return X

X = build_features(df["value"], window=12)
X.head(), X.shape
```

Out[6]:

```
(                           value   diff  roll_mean  roll_std
 timestamp
 2014-02-14 15:25:00+00:00  1.834  0.130   1.787833  0.102732
 2014-02-14 15:30:00+00:00  1.832 -0.002   1.796167  0.101843
 2014-02-14 15:35:00+00:00  1.826 -0.006   1.804000  0.100058
 2014-02-14 15:40:00+00:00  1.908  0.082   1.799667  0.093606
 2014-02-14 15:45:00+00:00  1.798 -0.110   1.805167  0.091176,
 (4021, 4))
```

6) Baseline anomaly detector (z-score residual)

As a baseline, we compute a rolling z-score:

[z = \frac{value - rolling_mean}{rolling_std}]

Then we flag anomalies where `|z| > 3.0`.

This is a classic monitoring rule (simple, explainable, and fast). It sets a benchmark for comparison against the ML approach.

Observed result (your run):

- Baseline anomalies flagged: **5**

In [7]:
```
# Baseline anomaly score (z-score residual)
resid = (X["value"] - X["roll_mean"]) / (X["roll_std"] + 1e-8)
Z_THRESHOLD = 3.0
baseline_anom = (np.abs(resid.values) > Z_THRESHOLD).astype(int)

print("Baseline anomalies flagged:", int(baseline_anom.sum()))
```

```
Baseline anomalies flagged: 5
```

7) Unsupervised anomaly detection with Isolation Forest

We train an `IsolationForest` model on engineered features:

- `[value, diff, roll_mean, roll_std]`

Isolation Forest works well for anomaly detection because it isolates rare points faster than normal points when building random decision trees.

Key settings:

- `n_estimators=300`: more trees for stability
- `contamination=0.03`: expected anomaly fraction (tunable)
- `random_state=42`: reproducibility

We also compute an **anomaly score** using the decision function. Higher scores indicate more “anomalous” points.

Observed result (your run):

- IsolationForest anomalies flagged: **121**

Interpretation:

- This model is more sensitive than the baseline rule and flags more “unusual” behavior. That is expected for an unsupervised detector with a contamination assumption.

In [8]:
```
# IsolationForest anomalies
iso = IsolationForest(
    n_estimators=300,
    contamination=0.03,
    random_state=RANDOM_SEED
)
iso.fit(X[["value", "diff", "roll_mean", "roll_std"]])

# -1 anomaly, +1 normal
iso_anom = (iso.predict(X[["value", "diff", "roll_mean", "roll_std"]])
scores = -iso.decision_function(X[["value", "diff", "roll_mean", "roll

print("IsolationForest anomalies flagged:", int(iso_anom.sum()))
```

```
IsolationForest anomalies flagged: 121
```

8) Visual inspection of detected anomalies

We overlay detected anomalies on the original metric plot.

This is important because (without labeled ground truth) the fastest validation is:

- Are anomalies aligned with visible spikes/drops?
- Are we flagging too many routine fluctuations?
- Does the pattern look consistent with operational incidents?

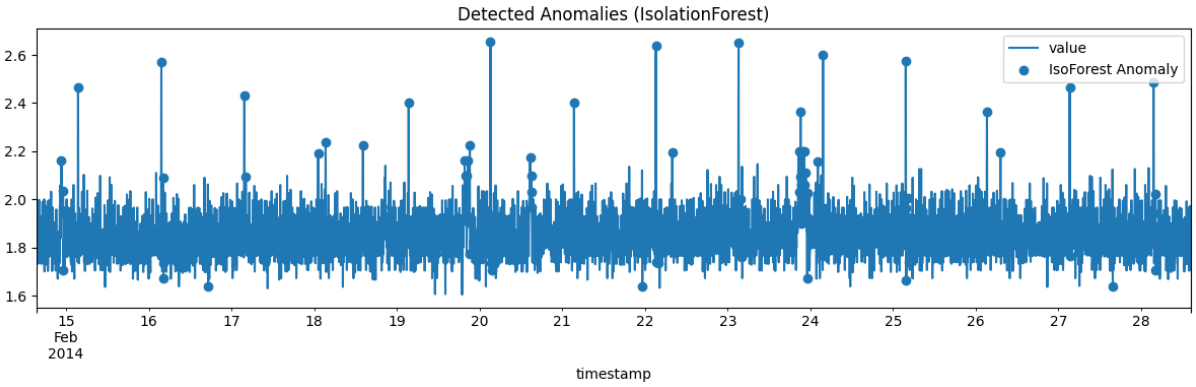
In a real SRE workflow, this step is usually combined with domain context (deployments, traffic changes, known outages).

In [9]:
```
# Plot detected anomalies
plot_df = X.copy()
plot_df["baseline_anom"] = baseline_anom
plot_df["iso_anom"] = iso_anom

fig, ax = plt.subplots(figsize=(12, 4))
plot_df["value"].plot(ax=ax)

anoms = plot_df[plot_df["iso_anom"] == 1]
ax.scatter(anoms.index, anoms["value"], label="IsoForest Anomaly")

ax.set_title("Detected Anomalies (IsolationForest)")
ax.legend()
plt.tight_layout()
plt.show()
```



9) Export the top anomalies (ranked by anomaly score)

To make results auditable and usable outside the notebook, we export the **top 50 anomalies** to:

- `artifacts/top_50_anomalies.csv`

This table becomes:

- evidence for the README
- a quick "incident list" for inspection
- a simple handoff format for downstream workflows (alerting dashboards, triage scripts)

Each row includes:

- timestamp
- metric value
- anomaly score
- binary anomaly flag

In [10]:
```
# Export top anomalies
top = pd.DataFrame({
    "timestamp": plot_df.index,
    "value": plot_df["value"].values,
    "anomaly_score": scores,
    "is_anomaly": iso_anom
}).sort_values("anomaly_score", ascending=False)

top_out = top.head(50)
top_path = ARTIFACTS_DIR / "top_50_anomalies.csv"
top_out.to_csv(top_path, index=False)

print("Saved:", top_path)
top_out.head(10)
```

```
Saved: /Users/shivesh/Desktop/PythonProject/SRE Metrics Intelligence /artifacts/top_50_anomalies.csv
```

Out[10]:

	timestamp	value	anomaly_score	is_anomaly
2445	2014-02-23 03:10:00+00:00	2.650	0.159293	1
2157	2014-02-22 03:10:00+00:00	2.638	0.156640	1
1581	2014-02-20 03:10:00+00:00	2.656	0.155227	1
2739	2014-02-24 03:40:00+00:00	2.600	0.154184	1
3027	2014-02-25 03:40:00+00:00	2.576	0.145970	1
435	2014-02-16 03:40:00+00:00	2.570	0.138395	1
2661	2014-02-23 21:10:00+00:00	2.364	0.137582	1
3890	2014-02-28 03:35:00+00:00	2.488	0.127801	1
145	2014-02-15 03:30:00+00:00	2.466	0.123660	1
1872	2014-02-21 03:25:00+00:00	2.400	0.118274	1

10) Save anomaly visualization for GitHub

We save the anomaly plot to:

- `images/anomaly_plot.png`

This is important for portfolio value:

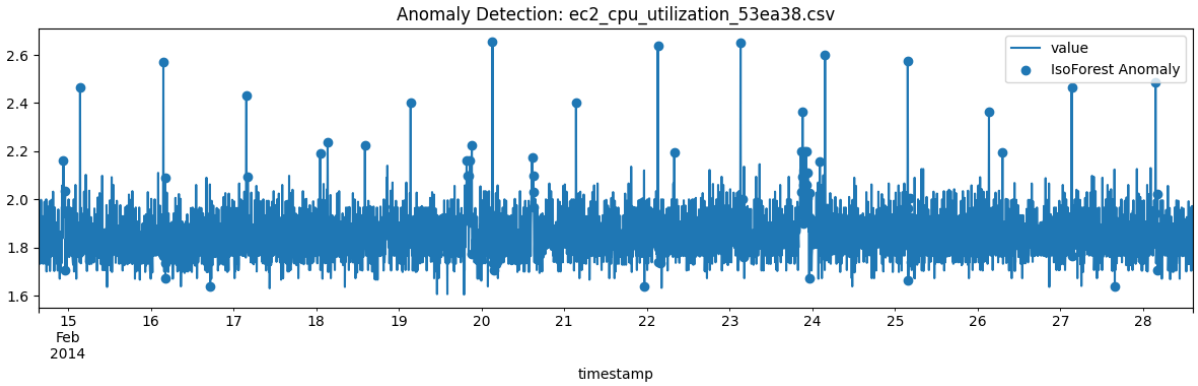
- GitHub visitors can understand the output instantly without running code.
- It communicates “monitoring intelligence” visually.

In [11]:

```
# Save anomaly plot
plot_path = IMAGES_DIR / "anomaly_plot.png"

fig, ax = plt.subplots(figsize=(12, 4))
plot_df["value"].plot(ax=ax)
anoms = plot_df[plot_df["iso_anom"] == 1]
ax.scatter(anoms.index, anoms["value"], label="IsoForest Anomaly")
ax.set_title(f"Anomaly Detection: {SERIES_PATH.name}")
ax.legend()
plt.tight_layout()
plt.savefig(plot_path, dpi=150)
plt.show()

print("Saved:", plot_path)
```




```
Saved: /Users/shivesh/Desktop/PythonProject/SRE Metrics Intelligence /images/anomaly_plot.png
```

Forecasting: Predicting near-future metric values

11) Lag-based forecasting setup

Operationally, forecasting helps with:

- capacity planning
- anticipating load spikes
- detecting deviations from expected behavior (forecast residuals)

We build a simple forecasting dataset using **lag features**:

- target: `y(t)` (current metric value)
- features: `y(t−1) ... y(t−24)`

With 5-minute sampling, 24 lags ≈ 2 hours of history.

We also use a **time-based split** (80% train, 20% test) without shuffling to avoid leakage.

In [12]:
```python
# Lag features
def make_lag_features(series: pd.Series, lags=24) -> pd.DataFrame:
    d = pd.DataFrame({"y": series})
    for i in range(1, lags + 1):
        d[f"lag_{i}"] = d["y"].shift(i)
    return d.dropna()

LAGS = 24
lag_df = make_lag_features(df["value"], lags=LAGS)

split = int(len(lag_df) * 0.8)
train = lag_df.iloc[:split]
```

```
test = lag_df.iloc[split:]

X_train, y_train = train.drop(columns=["y"]), train["y"]
X_test, y_test = test.drop(columns=["y"]), test["y"]

X_train.shape, X_test.shape
```

Out[12]: ((3206, 24), (802, 24))

12) Train a Ridge regression forecaster

We fit a `Ridge(alpha=1.0)` model on lag features.

Why Ridge?

- linear + fast
- stable with correlated lag features
- strong baseline for time-series forecasting

Metrics:

- MAE (mean absolute error): average absolute prediction error
- RMSE (root mean squared error): penalizes larger errors more heavily

Observed result (your run):

- Forecast MAE: **0.0469051630**
- Forecast RMSE: **0.0676286410**

Interpretation:

- The forecast tracks the signal closely and errors are relatively small compared to the metric range (~1.6 to ~2.6).

In [14]:
```
# Train Ridge forecaster
model = Ridge(alpha=1.0)
model.fit(X_train, y_train)

pred = model.predict(X_test)

mae = mean_absolute_error(y_test, pred)

# Universal RMSE (works regardless of sklearn version)
mse = mean_squared_error(y_test, pred)
rmse = float(np.sqrt(mse))

print("Forecast MAE:", float(mae))
print("Forecast RMSE:", rmse)
```

```
Forecast MAE: 0.04490516302559592
Forecast RMSE: 0.06762864104937613
```

13) Forecast vs Actual plot

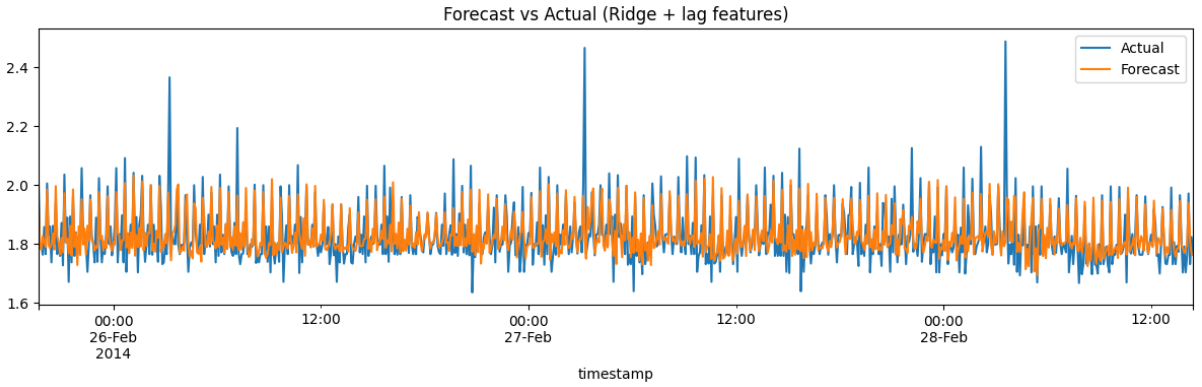
We plot the predicted series against the actual series on the test window.

What we look for:

- Does the forecast capture the periodic pattern?
- Do spikes get smoothed out (common for linear models)?
- Are there visible systematic errors?

This plot also becomes a strong portfolio artifact because it clearly demonstrates modeling output.

In [15]:
```
# Plot forecast vs actual
fig, ax = plt.subplots(figsize=(12, 4))
pd.Series(y_test.values, index=y_test.index).plot(ax=ax, label="Actual
pd.Series(pred, index=y_test.index).plot(ax=ax, label="Forecast")
ax.set_title("Forecast vs Actual (Ridge + lag features)")
ax.legend()
plt.tight_layout()
plt.show()
```



14) Save forecasting artifacts (metrics + plot)

We export:

- `artifacts/forecast_metrics.json` (MAE, RMSE, model config, lags, resample rule)
- `images/forecast_plot.png`

This makes results reproducible and easy to compare across runs if we later:

- change lags

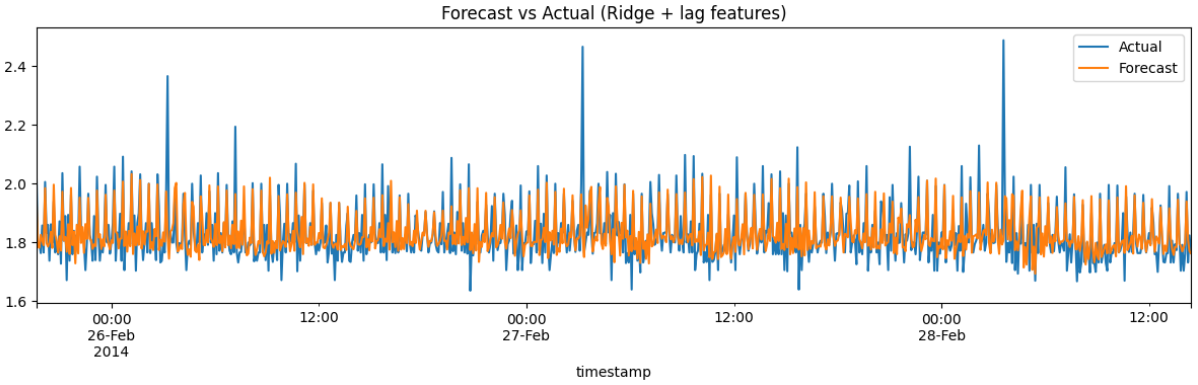
- try other models (XGBoost, LSTM, Prophet)
- tune resampling or feature engineering

In [16]:
```python
# Save forecast artifacts
forecast_metrics = {
    "series": SERIES_PATH.name,
    "resample_rule": rule,
    "lags": LAGS,
    "model": "Ridge(alpha=1.0)",
    "mae": float(mae),
    "rmse": float(rmse),
}

forecast_metrics_path = ARTIFACTS_DIR / "forecast_metrics.json"
with open(forecast_metrics_path, "w") as f:
    json.dump(forecast_metrics, f, indent=2)

forecast_plot_path = IMAGES_DIR / "forecast_plot.png"
fig, ax = plt.subplots(figsize=(12, 4))
pd.Series(y_test.values, index=y_test.index).plot(ax=ax, label="Actual
pd.Series(pred, index=y_test.index).plot(ax=ax, label="Forecast")
ax.set_title("Forecast vs Actual (Ridge + lag features)")
ax.legend()
plt.tight_layout()
plt.savefig(forecast_plot_path, dpi=150)
plt.show()

print("Saved:", forecast_metrics_path)
print("Saved:", forecast_plot_path)
```




```
Saved: /Users/shivesh/Desktop/PythonProject/SRE Metrics Intelligence /a
rtifacts/forecast_metrics.json
Saved: /Users/shivesh/Desktop/PythonProject/SRE Metrics Intelligence /i
mages/forecast_plot.png
```

15) Serialize models for reuse

We save trained models to disk:

- `artifacts/isolation_forest.joblib`
- `artifacts/ridge_forecaster.joblib`

This simulates a production pipeline step:

- train offline
- ship the model artifact
- load it in another script/service for batch scoring or real-time inference

This is exactly the kind of “systems + ML” behavior large tech companies care about.

In [17]:
```
# Save Models
iso_path = ARTIFACTS_DIR / "isolation_forest.joblib"
ridge_path = ARTIFACTS_DIR / "ridge_forecaster.joblib"

joblib.dump(iso, iso_path)
joblib.dump(model, ridge_path)

print("Saved:", iso_path)
print("Saved:", ridge_path)
```

```
Saved: /Users/shivesh/Desktop/PythonProject/SRE Metrics Intelligence /a
rtifacts/isolation_forest.joblib
Saved: /Users/shivesh/Desktop/PythonProject/SRE Metrics Intelligence /a
rtifacts/ridge_forecaster.joblib
```

Final Conclusions

What we built

This project implemented a compact “SRE metrics intelligence” pipeline on real AWS CloudWatch data:

- standardized loading + resampling
- explainable baseline anomaly detection (z-score residual)
- ML anomaly detection (Isolation Forest) with anomaly scoring
- short-term forecasting (Ridge regression + lag features)
- saved artifacts for reproducibility and GitHub presentation

Results summary

- Resample frequency: **5 minutes**
- Baseline anomalies flagged: **5**

- IsolationForest anomalies flagged: **121**
- Forecast MAE: **0.0469**
- Forecast RMSE: **0.0676**
- Artifacts exported: plots, CSV anomaly ranking, JSON metrics, joblib models

Interpretation

- The baseline method is conservative (few flags).
- Isolation Forest is more sensitive and provides a ranked anomaly list for investigation.
- The Ridge forecaster captures the short-term dynamics well, producing low MAE/RMSE and a close visual fit.

Limitations

- No ground-truth anomaly labels in this local dataset layout, so anomaly detection is validated via:
  - visual inspection
  - ranked anomaly scores
  - plausibility of flagged spikes
- The forecasting model is linear; extreme spikes may be under-predicted.

Next steps

1. Add an evaluation dataset with labels (precision/recall/F1) OR create synthetic incident windows.
2. Turn this into a small CLI tool:
   - `python score.py ––file ec2_cpu_utilization_53ea38.csv`
3. Add “forecast residual anomaly” detection:
   - anomalies = large forecast error, not just raw spikes
4. Add a simple alert rule:
   - “top N anomalies per day” + thresholds

This notebook demonstrates practical ML applied to monitoring metrics — relevant for backend, systems, SRE-adjacent ML, and platform engineering roles.

In [ ]: