Proof of Concept

Predictive Anomaly Detection & Incident prevention for SRE Production Monitoring Using Machine Learning

Vardharaj Konar

Summary:

This POC demonstrates how AI/ML can be integrated into an SRE monitoring workflow to **proactively predict anomalies** in production application logs. By leveraging log features such as event volume, error rates, and message characteristics, the ML system forecasts high-risk time windows empowering SREs to take preventive action before incidents occur.

1. Business Problem / SRE Challenge

Background:
Traditional SRE monitoring detects incidents *after* they occur, triggering alerts based on static thresholds or rule-based log scanning. However, this is reactive and can lead to slower incident response.

SRE Pain Points:

- High-severity events often follow subtle patterns missed by static rules.
- Over-alerting on benign spikes, under-alerting on novel failure modes.
- Need for "smart" early warning for incident prevention.

2. Problem It Solves:

Modern SRE teams rely on traditional monitoring and static alert rules, which often fail to catch complex or emerging incidents before they impact production. These methods can generate too many false alerts or miss early warning signs, resulting in slower response and longer downtime.

This POC solves the problem by:

- Using AI/ML to analyze log data and detect abnormal patterns **before** a critical incident happens.
- Providing SREs with **proactive alerts** so they can investigate and resolve issues earlier, reducing outages and improving system reliability.

3. Objective of the POC

- **Proactively predict periods of high risk ("anomalies") using ML models trained on log data.**
- Provide actionable alerts to SREs before a full-blown incident occurs.
- Integrate with existing observability pipelines and show the value of AI-driven monitoring.

4. Technical Approach

Data Pipeline

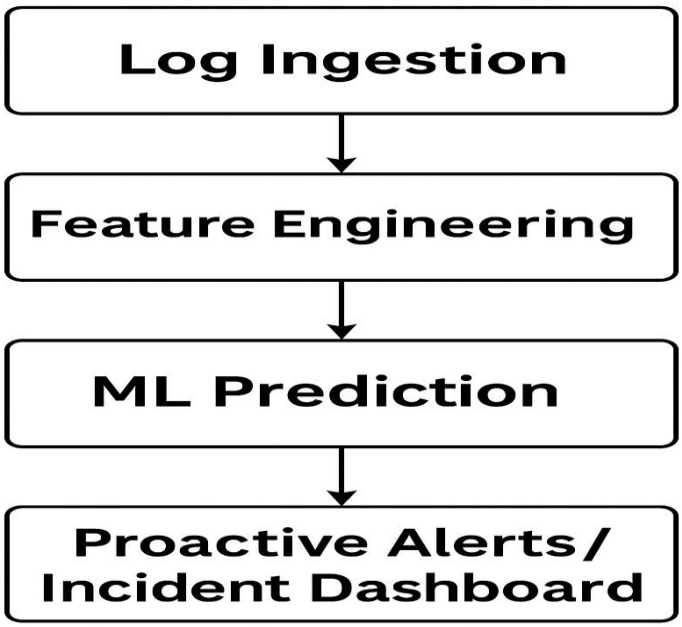
- **Input:** Application/system logs with timestamp, event ID, log level (INFO, WARNING, ERROR), and message content.
- **Preprocessing:**
    - Resample logs per minute.
    - Engineer features:
        - Event counts (per minute, rolling windows)

        - Error/warning rates
        - Average message length
- **Labeling:**
    - Define “anomaly” as time windows with high event rates

Modeling & Evaluation

- **ML Models Used:** Logistic Regression, Decision Tree, Random Forest.
- **Evaluation:** Accuracy, ROC-AUC, and confusion matrices on held-out test data.
- **Alerts:** Print actionable system prediction alerts for the last 10 minutes (simulating proactive intervention).

5. Architecture Diagram



6. Key Results

- ML models successfully learned to identify “risky” periods in log data.
- **Best model:** (fill in based on your results)
- **Alerts** :POC prints predicted system failure.

7. SRE/Production Benefits

- Early warning: **catch problems before they escalate**.
- Reduces incident MTTR (Mean Time To Recovery) by surfacing anomalies with context.
- Model can be retrained on real outages or extended with more features (CPU, memory, latency, etc.).

8. Limitations & Next Steps

- Current POC uses a synthetic anomaly definition (high log volume + errors).
  Real production deployment should use known incident timestamps for supervised training.
- Integration with live incident response tools (PagerDuty, Slack, ServiceNow) is a next step.
- Possible future work: anomaly explanation, root cause analysis, auto-remediation triggers.

9. Sample Output

"System Prediction Alerts for Latest 10 Minutes" output.
This shows that the system predicts risk, notifies the SRE, and recommends action.

```
        else:
            print(f"[INFO] System stable at {ts}")
```

```
[System Prediction Alerts for Latest 10 Minutes]
[INFO] System stable at 2006-01-03 07:04:00 (probability of failure: 0.00)
[INFO] System stable at 2006-01-03 07:05:00 (probability of failure: 0.00)
[INFO] System stable at 2006-01-03 07:06:00 (probability of failure: 0.00)
[INFO] System stable at 2006-01-03 07:07:00 (probability of failure: 0.00)
[INFO] System stable at 2006-01-03 07:08:00 (probability of failure: 0.00)
[INFO] System stable at 2006-01-03 07:09:00 (probability of failure: 0.00)
[INFO] System stable at 2006-01-03 07:10:00 (probability of failure: 0.00)
[INFO] System stable at 2006-01-03 07:11:00 (probability of failure: 0.00)
[INFO] System stable at 2006-01-03 07:12:00 (probability of failure: 0.00)
[ALERT] System failure predicted at 2006-01-03 07:13:00 (probability: 1.00)
[ACTION] restart_application_server()
```

10. Appendix: Code snippet

File Edit View Insert Runtime Tools Help

nands + Code + Text ▷ Run all

[28]
```python
# --- SYNTHETIC LABEL ---
df_feat['label'] = (df_feat['count_1min'] > df_feat['count_1min'].quantile(0.7)).astype(int)

# --- DEFINE FEATURES & TARGET ---
feature_cols = [
    'count_1min', 'count_3min', 'count_5min',
    'error_warn_per_min', 'avg_msg_length'
]
# Ensure only existing columns are selected
feature_cols = [col for col in feature_cols if col in df_feat.columns]

X = df_feat[feature_cols]
y = df_feat['label']

# --- DATA CHECK ---
print("df_feat shape:", df_feat.shape)
print("X shape:", X.shape)
print("Label counts:\n", y.value_counts())

# --- TRAIN/TEST SPLIT ---
if len(X) > 1 and y.nunique() > 1:
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )
    print("Training set shape:", X_train.shape)
    print("Test set shape:", X_test.shape)
    print("Features used:", feature_cols)
else:
    print("\nNot enough data for a train/test split. Please check your feature engineering or add more log data.")
```

Features Engineering

1. count_1min

- **What it is:**
  The **number of log events (rows) recorded within each 1-minute window**.
- **How it's calculated:**
  After resampling the logs by minute, you simply count how many log entries occurred in that minute.
- **Why it matters:**
  A sudden spike in event count can indicate abnormal activity or an issue (such as rapid error generation, retries, or flooding).

2. count_3min

- **What it is:**
  The **rolling average of event counts over the past 3 minutes** for each 1-minute window.
- **How it's calculated:**
  For each minute, take the average number of events in that minute plus the previous two minutes.
- **Why it matters:**
  This smooths out short-term fluctuations and highlights short-term trends in log volume. A rising average can signal that something is building up toward a failure.

3. count_5min

- **What it is:**
  The **rolling average of event counts over the past 5 minutes** for each 1-minute window.
- **How it's calculated:**
  Similar to count_3min, but averaged over the current and previous four minutes.
- **Why it matters:**
  This captures more medium-term trends and helps distinguish between temporary bursts and sustained increases in event rates.

4. error_warn_per_min

- **What it is:**
  The **number of log entries with level 'ERROR' or 'WARNING' per minute**.
- **How it's calculated:**
  For each 1-minute window, count the logs where the Level column is either 'ERROR' or 'WARNING'.
- **Why it matters:**
  High numbers here directly reflect problems and issues being logged. If this number rises, it's a red flag for possible or impending failure.

5. avg_msg_length

- **What it is:**
  The **average length (number of characters) of log messages within each minute**.
- **How it's calculated:**
  For each 1-minute window, calculate the average string length of all entries in the Content (or message) column.
- **Why it matters:**
  Sometimes, error logs or stack traces are much longer than regular logs. An increase in average message length may indicate the system is generating more verbose (detailed) error reports, which often precede or follow failures.

Data set source:
https://github.com/logpai/loghub/blob/master/BGL/BGL_2k.log_structured.csv

Source Code:

https://colab.research.google.com/drive/16ki83LyBd6NDIiubTwWFoLh2NzZoJlXt?usp=sharing