MACHINE LEARNING ANOMALY DETECTION SYSTEM

THREADING LABS

MARCO ZETINA

TABLE OF CONTENTS

EXECUTIVE SUMMARY

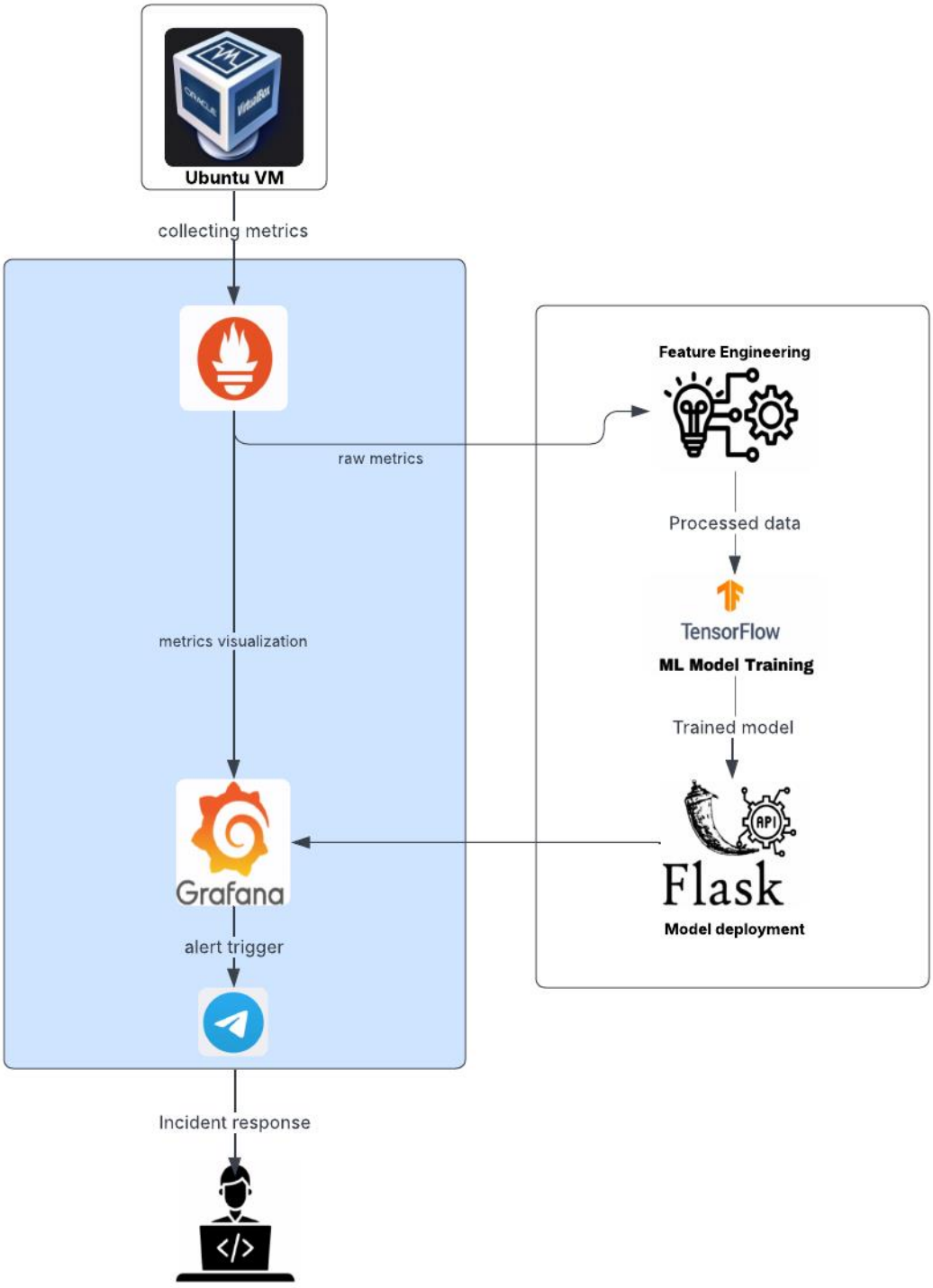
This solution monitors system metrics in real time, detects anomalies using an autoencoder trained on the normal behavior of the machine, and triggers alerts via Prometheus and Grafana. This proactive approach minimizes downtime and supports business continuity by enhancing the system reliability.

The anomaly detection system goes beyond the traditional DevOps approach of setting fixed metric thresholds. Instead of waiting for a single metric, for example CPU, to exceed a known limit, the pipeline continuously monitors subtle multivariate correlations between the different metrics, such as CPU and memory usage, disk I/O and many more. Whenever the reconstruction error surpasses the pipeline threshold, an anomaly is flagged, and it could reveal problems before any single metric reach their own individual threshold.

This proactive detection of hidden correlations and unusual resource usage patterns aims to reduce false negatives and helps DevOps teams by improving the reliability of the system, and leading to a faster detection.

Moreover, the solution is designed for scalability and integration with modern DevOps with the option of adding automatic remediation actions such as restarting failing pods or scaling resources when anomalies are detected.

ARCHITECTURE



THE SYSTEM IS COMPOSED OF THE FOLLOWING MAIN COMPONENTS:

1. **Ubuntu VM**

- Hosts the target environment where system metrics originate.

2. **Prometheus and Grafana**

- **Prometheus**: Scrapes system metrics from the Ubuntu VM at regular intervals and stores them in a time series database.
- **Grafana**: Visualizes the collected metrics in real time, providing dashboards and alerting capabilities.

3. **Feature Engineering and Model Training**

- A Python script "process_prometheus_data.py" processes raw metrics into a comprehensive dataset, applying:
  - **Data cleaning:** All the possible NaN values are handled: either removed or filled.
  - **Min-Max Scaling** to normalize values.
  - **Rolling averages**, **change rates**, **and ratios** to provide meaningful features.
- A TensorFlow autoencoder model is trained on this processed dataset to learn the system's normal behavior.

4. **Flask Model Deployment**

- The trained autoencoder is converted to TensorFlow Lite for inference in Ubuntu.
- A **Flask API** (flask_inference.py) hosts the TFLite model and exposes endpoints to:
  - Compute the reconstruction error on the latest metrics for the anomaly detection.
  - Serve the current anomaly score via the /metrics endpoint.

5. **Prometheus Scraping (Anomaly Score)**

- **Prometheus** is configured to scrape the Flask API's /metrics endpoint every 15s.
- This collects the **anomaly_score** (reconstruction error) and stores it alongside system metrics.

6. **Alerting**

- **Grafana** reads both system metrics and the anomaly score from Prometheus.
- A **threshold** flags anomalies when exceeded and notifies the DevOps team via Telegram.

JUSTIFICATION FOR DESIGN CHOICES

WHY AUTOENCODER FOR ANOMALY DETECTION?

The metrics collected for this project are unlabeled. Therefore, as an unsupervised deep learning model, the autoencoder, which is widely used for anomaly detection, is an ideal choice for this project. The autoencoder learns the normal behavior of the system purely by the input data. It does it by compressing the data into a lower dimensional representation (the bottleneck layer) and then reconstructing it back to its original form, outputting a reconstruction error. The autoencoder adapts to different system conditions to set a threshold, and identifies subtle anomalies that the human eye, as well as traditional arbitrary thresholds (e.g., CPU > 80%) could not foresee.

WHY FEATURE ENGINEERING? (ADDING ROLLING AVERAGES, RATIOS, AND CHANGE RATES)

Raw system metrics, such as CPU, memory or disk usage may not be enough to detect complex anomalies. Feature engineering enhances the model's ability to detect more subtle issues by creating other meaningful insights, like: Rolling averages, change rates and ratios. By creating this engineered features the model gains a much richer representation of the system's behavior, leading to more accurate anomaly predictions.

WHY FLASK FOR MODEL DEPLOYMENT?

Flask is a lightweight and efficient framework to quickly deploy the trained model. Since anomaly detection needs to be real time and scalable, Flask provides seamlessly integration with between the trained model, the monitoring system and alerting mechanisms. It gives and easy way to deploy the model via an API, making anomaly detection to be accessible by HTTP endpoints.

WHY PROMETHEUS AND GRAFANA FOR MONITORING?

Prometheus and Grafana where chosen for this project due to their robust monitoring capabilities. Prometheus is designed for collecting and storing time-series data, which makes it ideal for system performance monitoring, and it efficiently handles high frequency scrapping. It also provides PromQL, which allows powerful queries for real time analysis.

Grafana, on the other hand, seamlessly integrates with Prometheus, and also allows for PromQL, it allows data pull from several sources which makes it ready for future expansions, as well as real time notifications through several sources.

Note: The autoencoder was trained with over two thousand samples that can be found in the processed_prometheus_data_scaled.py capturing a variety of system behaviors and workload conditions, however, the model is designed to continuously evolve as new data is collected. Future iterations can incorporate additional training data to improve accuracy and adapt to changing system patterns to ensure long term reliability.

INSTALL AND CONFIGURE PROMETHEUS

First, update the system packages:

sudo apt update

sudo apt upgrade

Then, download the latest version of Prometheus from the Github repository.

wget *https://github.com/prometheus/prometheus/releases/download/v3.2.1/prometheus-3.2.1.linux-amd64.tar.gz*

Then extract the downloaded file:

tar xvfz prometheus-*.tar.gz

Move the binary files

sudo mv prometheus /usr/local/bin

sudo mv promtool /usr/local/bin

and reload systemd:

sudo systemctl daemon-reload

sudo systemctl enable prometheus

sudo systemctl start Prometheus

After that, install similarly the Node Exporter[1]:

wgethttps://github.com/prometheus/node_exporter/releases/download/v1.9.0/node_exporter-linux-amd64.tar.gz

tar xvf node_exporter-linux-amd64.tar.gz

sudo mv node_exporter-linux-amd64/node_exporter /usr/local/bin/

And start it as service:

sudo systemctl daemon-reload

sudo systemctl enable node_exporter

sudo systemctl start node_exporter

then, modify the Prometheus.yml file to collect system metrics, (and later to scrap Flask), so run

sudo nano /etc/prometheus/prometheus.yml

and add the following job to get system metrics:

```
- job_name: 'node-exporter'
  scrape_interval: 15s
  static_configs:
    - targets: ['localhost:9100']
```

Finally, restart it

sudo systemctl restart Prometheus

[1] https://prometheus.io/docs/guides/node-exporter/

SET CRONJOBS

In order to continuously store system metrics, we need to set up cronjobs that run every minute.

crontab -e

and add the following content:

** * * * * /usr/bin/curl -G 'http://localhost:9090/api/v1/query' --data-urlencode 'query=100 - (avg by (instance) (rate(node_cpu_seconds_total{mode="idle"}[5m])) * 100)' > /home/marco/>*

** * * * * /usr/bin/curl -G 'http://localhost:9090/api/v1/query' --data-urlencode 'query=node_memory_Active_bytes / node_memory_MemTotal_bytes * 100' > /home/marco/memory_usage.json*

** * * * * /usr/bin/curl -G 'http://localhost:9090/api/v1/query' --data-urlencode 'query=pg_stat_database_numbackends' > /home/marco/postgres_connections.json*

** * * * * /usr/bin/python3 /home/marco/process_prometheus_data.py*

** * * * * /usr/bin/curl -G 'http://localhost:9090/api/v1/query' --data-urlencode 'query=node_disk_read_bytes_total' > /home/marco/disk_read.json*

** * * * * /usr/bin/curl -G 'http://localhost:9090/api/v1/query' --data-urlencode 'query=node_disk_written_bytes_total' > /home/marco/disk_write.json*

** * * * * /usr/bin/curl -G 'http://localhost:9090/api/v1/query' --data-urlencode 'query=node_network_receive_bytes_total' > /home/marco/net_receive.json*

** * * * * /usr/bin/curl -G 'http://localhost:9090/api/v1/query' --data-urlencode 'query=node_network_transmit_bytes_total' > /home/marco/net_transmit.json*

Note: Update paths accordingly.

There is a cronjob for every initial system feature, as well as for running the next file, process_prometheus_data.py

This queries will be running every minute and storing system metrics into JSON files, for later data cleaning and preprocessing. The more time that the cronjobs run, the more data that will be collected and used for model training.

DATA CLEANING AND PREPROCESSING

Clone the repository:

git clone https://github.com/Marrrco7/ML-Anomaly-Detection.git

cd ML-Anomaly-Detection

If we only use Ubuntu, when we will import Tensorflow in Python, we will have an AVX problem. Even though our machines support AVX instructions the issue may still persist due to the VM configuration. For an easier way to train the model we will use the host system to train the Autoencoder and then use the shared folder of the virtual machine to share all the necessary files, and then convert it to Tensorflow Lite.

Once the paths are adjusted, run the ***process_prometheus_data.py***, to clean all the extracted data from the JSON files used in the cronjobs, append it together and scaling it into values between 0 and 1, since the different metrics have different scale measures like percentages for CPU usage and Mb/s for disk write, we need to ensure that all the data is in the same scale.

The data was cleaned either by removing the NaN values, or replacing them with the average value for that particular feature to avoid outliers in the data.

Moreover, are two main ways of scaling our data: Normalization and standardization. For our collected data we have used normalization, since all the collected metrics don't follow a normal distribution. Otherwise we would have used the z-score formula for standardization.

After running ***process_prometheus_data.py*** the data has to be split into train and test sets. So the next step will be to run the ***prepare_data.py*** file.

However, according to the date ranges of the collected data, before running the file it will be necessary to modify the following lines, 10 and 12 of code in ***prepare_data.py***:

```
train_df = df[(df['timestamp'] >= '2025-02-21') & (df['timestamp'] < '2025-02-24')]
test_df = df[df['timestamp'] < '2025-02-21']
```

Is recommended to allocate more days for the training data. The tests data will be the unseen data for the model in order to test its reconstruction error.

MODEL TRAINING

After this it will be necessary to train the autoencoder with the test and train data.

Now, let's take a look at the autoencoder architecture in train_autoencoder.py.

The autoencoder structure is composed of three parts:

- Encoder: Compresses the input into a lower dimensional representation
- Bottleneck: The layer with the most compressed representation of all the data
- Decoder: Tries to reconstruct the input from the previous compressed representation.

For the number of neurons, we go from:

Input → 14 → 8 → 4 (bottleneck) → 8 → 14 → Output

At first, the number of neurons decreases because the decoder learns a more compact representation of the data. The smaller number of neurons in the bottleneck forces the neural network to retain the most essential system patterns. After that the decoder expands the compact bottleneck representation back to the original shape.

In contrast, for the activation functions we used:

- Relu: In hidden layers, to help to learn complex patterns and deal with the vanishing gradient problem.
- Sigmoid: to Normalize back the output values into 0 and 1, so that it matches the system metrics scale.

Now we have a trained model on the collected system metrics.

THRESHOLD

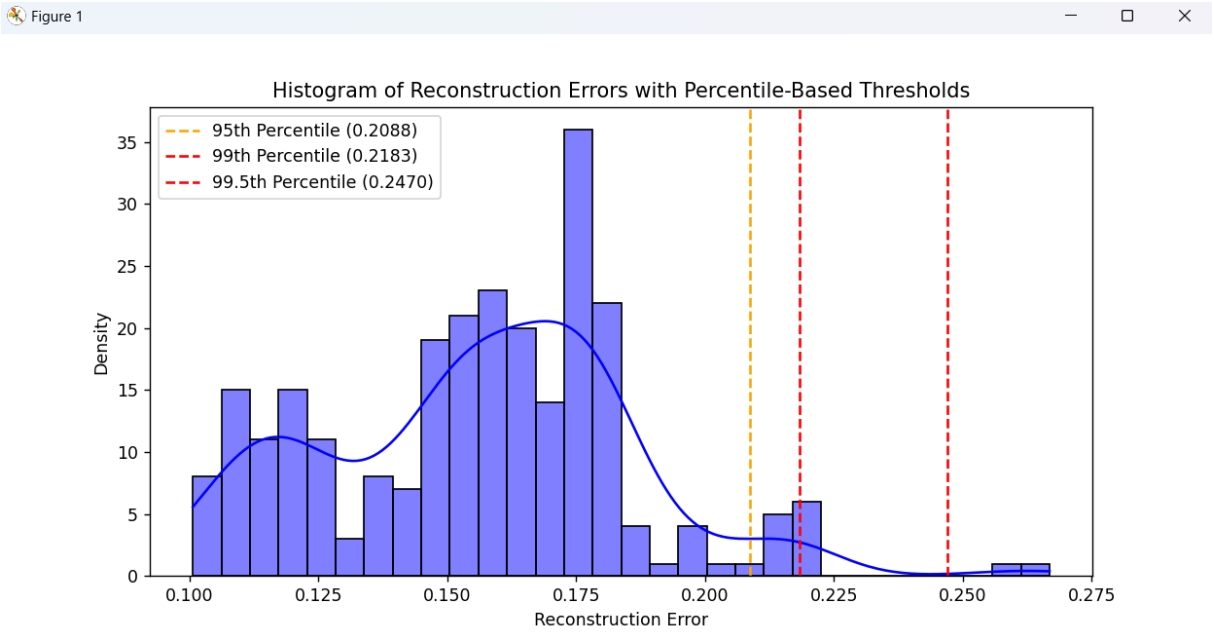
Now that we have the trained model and saved, it is necessary to find the threshold for the anomalies. Before that, we need to check what approach for finding a threshold suits the best, if either a percentile based threshold or computing the mean plus three standard deviations.

To decide that, you can first run the ***reconstruction_errors_py*** file, so that you can visualize the graph of the distribution of the reconstruction errors of the train model. Then if the graph is skewed more on either side, it means it does not follow a normal distribution. Apart from that, there are two tests in that file: the Shapiro Test and the Kolmogorov-Smirnov test. Both of them to determine if the distribution of the reconstruction errors is normal distributed.

Either way, if the data follows the normal distribution (meaning it shows a more even graph and both, the Shapiro and Kolmogorov tests for normalization are passed) then you will have to use the mean plus 3 standard deviations approach for more accuracy. Otherwise, as it was shown for our computed reconstruction errors, you will have to use the percentiles approach.

Now to answering the question: **what percentile should I choose?**

You can run the thresholds_percentiles.py file, where you will see a similar graph as this one:



There you can see the different percentiles to check the portions of data that would lie as anomalies, in order to minimize the false positive rate (the amount of times that there is an anomaly alert when actually there was no necessity) is recommended to choose a suitable percentile, such as the 99^{th}, or 95^{th} percentile. That would mean that either the top 1% or 5% of the analyzed system metrics minute by minute would be detected as an anomaly.

Note: The choice of percentile depends on how tolerant the team is to false alerts. it determines how sensitive the model will be to anomalies. The recommendation is a balanced approach, where the 95th percentile is a good starting point.

CONVERT TO TFLITE

Now, to solve the AVX problem we should convert the trained model into TFlite before moving to Ubuntu to run inference (applying the trained model to the new unseen data collected minute by minute).

For that just run the convert_to_tflite.py file, and then move that file to the home directory, so that it is visible in the shared folder of the VM.

MOVE BACK TO UBUNTU

Now that you are in the Ubuntu VM, you will have to install the following:

pip install flask prometheus-client apscheduler pandas numpy tflite-runtime

- flask: Provides the API framework.
- prometheus-client: Enables Prometheus metrics exposure.
- apscheduler: Allows the periodic execution of model inference.
- tflite-runtime: Runs the TensorFlow Lite model efficiently and solves the AVX issue.

We have built a **Flask API** (***flask_inference.py***) to serve the model and allow real time anomaly detection.

- The API **reads the latest system metrics** from the processed Prometheus data.
- The model **performs inference** on this new data.
- The **reconstruction error is calculated** to determine if an anomaly is present.

CONFIGURE PROMETHEUS

Also, before running flask_inference.py, it will be necessary to update the Prometheus.ym file and add the following job:

```
- job_name: 'flask_inference'
  scrape_interval: 15s
  static_configs:
    - targets: ['127.0.0.1:8001']
```

This to scrape the flask endpoint every 15 seconds to get the latest anomaly score.

sudo systemctl restart Prometheus

CONFIGURE NGINX AND GUNICORN

Because of the previous solution, Gunicorn and Nginx should be already installed, however, it will be necessary to add the following configuration for Nginx, run

nano /etc/nginx/sites-available/flask_inference

and paste the following content:

```
server {
    listen 80;
    server_name 127.0.0.1;

    location /anomaly/ {
        proxy_pass http://127.0.0.1:8001/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Then link this configuration file to the sites-enabled directory to activate it:

sudo ln -s /etc/nginx/sites-available/flask_inference /etc/nginx/sites-enabled/

Lastly, restart Nginx:
sudo systemctl restart nginx

For Gunicorn, add the following configuration:

nano /etc/systemd/system/flask_inference.service

```
[Unit]
Description=Flask Inference App Service
```

After=network.target

[Service]

User=marco

Group=www-data

WorkingDirectory=/home/marco

Environment="PATH=/home/marco/tflite_env/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin"

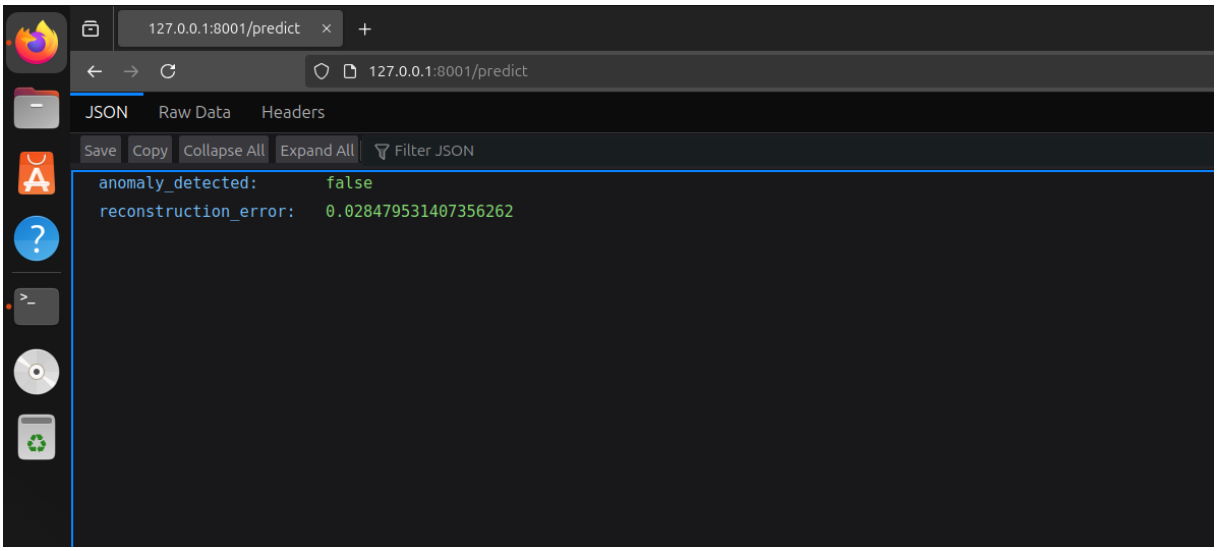
ExecStart=/home/marco/tflite_env/bin/gunicorn --workers 4 --bind 127.0.0.1:8001 flask_inference:app

Restart=always

[Install]

WantedBy=multi-user.target

Now check that the flask_inference.py is running at *http://127.0.0.1:8001/predict* this will be available to run on boot since you turn on the machine.



INSTALL AND CONFIGURE GRAFANA

First, install Grafana following the same instructions as in the official site[2]:

```
sudo apt-get install -y apt-transport-https software-properties-common wget

sudo mkdir -p /etc/apt/keyrings/

wget -q -O - https://apt.grafana.com/gpg.key | gpg --dearmor | sudo tee /etc/apt/keyrings/grafana.gpg > /dev/null

echo "deb [signed-by=/etc/apt/keyrings/grafana.gpg] https://apt.grafana.com stable main" | sudo tee -a /etc/apt/sources.list.d/grafana.list

# Updates the list of available packages

sudo apt-get update

# Installs the latest OSS release:

sudo apt-get install grafana
```

Now log in at *http://localhost:3000*, now go to Configuration → Data Sources. Click Add Data Source → Select Prometheus.

Go to Dashboards → Click New Dashboard. Click Add a New Panel. In the Query Editor, select Prometheus as the data source and enter the PromQL Query: ***anomaly_score***

And customize the graph with the desired time range, lastly save it.

[2] https://grafana.com/docs/grafana/latest/setup-grafana/installation/debian/

ROI AND KPI'S

RETURN ON INVESTMENT (ROI)

Reduced Downtime

Anomaly detection encourages early identification of potential system failures, preventing unplanned crashes and minimizing disruption to operations.

Proactive Issue Resolution

AI powered monitoring predicts and alerts on anomalies before they escalate into critical failures, allowing engineers to take preventive actions.

Improved Team Productivity

Automated detection reduces the time engineers spend troubleshooting, allowing them to focus on system improvements, innovation and a faster SDLC.

KEY PERFORMANCE INDICATORS (KPIS)

Detection Rate (Accuracy of Anomalies)

The percentage of actual anomalies correctly detected by the system.

Detection Rate=(True Positives/True Positives + False Negatives) * 100

False Positive Rate (False Alerts)

The percentage of alerts that are false positives, ensuring that only real issues are flagged.

False Positive Rate: (False Positives/False positives + True Negatives) * 100

Mean Time to Detect

The average time taken by the system to identify and alert an anomaly after it occurs.

Mean time =Total Number of Anomalies/Total Time Taken to Detect Anomalies

CONCLUSION

AI powered DevOps can be a game changer for modern practices. Implementing this technologies requires careful planning, but the benefits are worth the effort. By starting to adopt AI on DevOps, the teams can put a major focus on innovation rather than firefighting, allowing to expand the full potential of their infrastructure and applications. This pipeline successfully integrates data collection, machine learning, real time inference, and alerting. By combining Prometheus, Grafana, and an autoencoder served via Flask, the system automatically flags anomalous behavior and ensures proactive incident response

Note: The autoencoder was trained with over two thousand samples that can be found in the processed_prometheus_data_scaled.py capturing a variety of system behaviors and workload conditions, however, the model is designed to continuously evolve as new data is collected. Future iterations can incorporate additional training data to improve accuracy and adapt to changing system patterns to ensure long term reliability.

References