BANGALORE INSTITUTE OF TECHNOLOGY
K.R. Road, V.V. Pura, Bengaluru-560 004

Hackathon Phase 1
AI/ML Model for Predicting Kubernetes Issues

Submitted by

Team Name
“BIT MAVERICKS”

Team Members

Abhiraam S
Ranjan P
Darshan Sunil Kuranagi
Abhishek R
Dornadula Sai Gagan

GUIDEWIRE
DEVTrails
University Hackathon

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Background

Guidewire Software, a leader in P&C insurance technology, is renowned for empowering innovators in the industry. In 2025, the company is hosting the DEVTrails University Hackathon—a 45-day virtual event exclusively designed for university students across India. This hackathon is a high-energy, fast-paced competition that challenges participants to tackle real-world issues in areas like AI, sustainability, and tech accessibility. By bringing together emerging talent and seasoned industry experts, the event aims to ignite creativity, foster collaboration, and drive technological advancements that can transform the future of insurance and beyond.

1.2 Objectives

This 45-day hackathon challenge invites participants to develop a model that can predict potential issues in Kubernetes clusters (Phase 1) and recommend or automatically implement solutions to address those issues (Phase 2). The aim is to optimize the efficiency of Guidewire solutions on Cloud Infrastructure, using only publicly available resources and open-source solutions.

1.3 Phase 1: AI/ML Model for Predicting Kubernetes Issues

Problem Statement
Kubernetes clusters can encounter failures such as pod crashes, resource bottlenecks, and network issues. The challenge in Phase 1 is to build an AI/ML model capable of predicting these issues before they occur by analyzing historical and real-time cluster metrics.

Key Objectives for Phase 1:

- **Data Collection:**
    - Utilize publicly available datasets or simulate key metrics from Kubernetes clusters.
    - Collect metrics such as CPU usage, memory usage, pod status, network IO, and any other relevant information.

- **Model Design:**
    - Build a model capable of predicting the following issues (minimal viable scope, with potential for additional functionalities):
        - Node or pod failures.
        - Resource exhaustion (CPU, memory, disk).
        - Network or connectivity issues.
        - Service disruptions based on logs and events.

- **Prediction Accuracy:**
    - Focus on developing models that accurately forecast potential failures.
    - Implement techniques such as anomaly detection, time-series analysis, and other applicable approaches.

2. SCOPE

This report provides a detailed account of our journey throughout the hackathon and is organized into the following key sections:

2.1. Problem Statement:

Kubernetes clusters can encounter failures such as pod crashes, resource bottlenecks, and network issues. The challenge in Phase 1 is to build an AI/ML model capable of predicting these issues before they occur by analysing historical and real-time cluster metrics.

2.2. Methodology:

Data Collection & Preprocessing:

- Retrieve and aggregate datasets from multiple sources.
- Clean and normalize data, handling missing values and formatting log data into a structured form suitable for model ingestion.

Model Design:

- **Pod Failure Prediction:**
  - **LSTM Model:** Utilized for analyzing sequential log data to predict pod crashes based on historical behavior.
- **Network Failure Prediction:**
  - **ANN Model:** Designed to detect anomalies in network traffic and connectivity issues.
- **Resource Exhaustion Prediction:**
  - **CNN Model:** Applied to recognize spatial patterns in resource usage metrics (CPU, memory, disk) to forecast resource exhaustion.

Training & Evaluation:

- Train each model on the preprocessed data, employing cross-validation and hyperparameter tuning to optimize performance.
- Evaluate the models using metrics such as accuracy, precision, recall, and F1-score to ensure reliable predictions.

Integration & Deployment:

- Package the trained models and all dependencies within a Kubernetes container.
- Deploy the solution in a test environment to monitor real-time performance and validate the predictive capabilities.

3. LITERATURE REVIEW AND RELATED WORK

3.1 Autoscaling in Kubernetes

Kubernetes has revolutionized the management of containerized applications by providing built-in autoscaling mechanisms such as the Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA). These autoscalers adjust the number of running pods or their resource allocations based on real-time metrics like CPU and memory usage. Although these reactive approaches help maintain service performance under varying workloads, they tend to lag in response to rapid workload changes. This lag often results in either under-provisioning or over-provisioning of resources, which can compromise application performance and increase operational costs. The traditional autoscaling systems in Kubernetes primarily rely on preset thresholds, making them less adaptive to the complexities of dynamic cloud environments.

3.2 Predictive Hybrid Autoscaling

To overcome the limitations of reactive autoscaling, recent research has focused on predictive hybrid autoscaling techniques that integrate both horizontal and vertical scaling strategies into a unified framework. Predictive models such as SARIMA are employed to forecast future workload demands based on historical data, enabling the system to adjust resource allocation preemptively. By analyzing historical metrics and identifying workload patterns, these models anticipate workload spikes and resource requirements in advance. Additionally, burst identification algorithms are incorporated to detect sudden surges in workload, ensuring that scaling decisions are both timely and efficient. This hybrid approach addresses the dual objectives of maintaining Quality of Service (QoS) and optimizing resource utilization in cloud-native environments.

3.3 Existing Systems and Limitations

Existing systems for anomaly prediction and autoscaling in Kubernetes encompass a range of techniques. Traditional monitoring tools, including Prometheus and Grafana, are widely used to collect real-time metrics and trigger alerts based on predefined rules. However, these tools often fall short in accurately predicting anomalies because they rely on reactive, threshold-based methods. On the machine learning front, several studies have explored time-series forecasting models, such as ARIMA and SARIMA, to predict resource utilization and potential anomalies. Unsupervised learning techniques like Isolation Forest and One-Class SVM have also been applied to identify outlier behaviors in cluster performance data. Furthermore, cloud-native solutions provided by major providers—such as AWS CloudWatch, Azure Monitor, and Google Cloud's Operations Suite—incorporate built-in anomaly detection features that leverage machine learning for enhanced predictive capabilities. Despite these advances, many current systems are

limited by their reactive nature, inability to handle class imbalance effectively, or their reliance on fixed resource configurations. These shortcomings often result in suboptimal performance under highly dynamic workloads. In contrast, our proposed solution distinguishes itself by integrating predictive workload forecasting, oversampling techniques like SMOTE, and a hybrid scaling algorithm that simultaneously determines pod count and resource allocation, thereby directly addressing both QoS and resource optimization challenges in Kubernetes clusters.

4. METHODOLOGY

4.1 Data Collection and Preprocessing

We leverage multiple publicly available datasets:

- Mendeley Dataset: https://data.mendeley.com/datasets/ks9vbv5pb2/1
- Kaggle Dataset: https://www.kaggle.com/datasets/nickkinyae/kubernetes-resource-and-performancemetricsallocation?resource=download
- 4TU Dataset: https://data.4tu.nl/articles/dataset/AssureMOSS_Kubernetes_Run-time_Monitoring_Dataset/20463687

The data preprocessing notebooks (*data_preprocessing1.ipynb* and *data_preprocessing1_copy.ipynb*) document our exploratory trials and experiments to understand feature relevance and model accuracy.

4.2 Feature Engineering and Oversampling

Key derived features include:

- Resource Utilization Ratios: Calculated by dividing resource usage by limits (e.g., CPU utilization).
- Differential Metrics: Differences between actual usage and requested resources.
- Time-based Rolling Metrics: Rolling averages and standard deviations over a defined window to capture temporal behavior.

Our feature engineering process begins with the derivation of resource utilization ratios. These ratios are calculated by dividing the observed resource usage (e.g., CPU or memory) by their respective limits, providing a normalized measure of how intensively resources are used. This step is crucial because it allows for the comparison of resource consumption across different nodes or applications, regardless of their absolute capacity.

In addition to utilization ratios, we derive differential metrics that represent the differences between the actual resource usage and the requested resources. These metrics are particularly useful in identifying deviations from expected behavior, which can be early indicators of potential anomalies. Capturing these differences helps the model to distinguish between normal fluctuations and significant outliers that may require intervention.

To capture temporal behavior, we incorporate time-based rolling metrics. By calculating rolling averages and standard deviations over a defined window, we can observe trends and fluctuations in resource usage over time. This approach not only smooths out short-term noise but also highlights sudden changes in usage patterns, which are critical for effective anomaly detection in dynamic Kubernetes environments.

Handling missing data is also an important aspect of our feature engineering pipeline. We use a SimpleImputer to address any gaps in the dataset by replacing missing values with statistical measures, such as the mean or median. This step ensures that our dataset remains complete and reliable, preventing potential issues during model training.

Furthermore, to reduce the dimensionality of our feature set and eliminate redundant information, we apply Principal Component Analysis (PCA). PCA transforms the original features into a smaller set of uncorrelated components that capture most of the variance in the data. This not only simplifies the model but also improves computational efficiency while retaining the essential patterns necessary for accurate predictions.

Finally, to tackle the challenge of class imbalance—especially given the rarity of anomaly events—we employ SMOTE (Synthetic Minority Over-sampling Technique). SMOTE generates synthetic samples for the minority class by interpolating between existing examples rather than simply duplicating them. This technique ensures that rare events, such as resource usage anomalies, are adequately represented during model training, thereby enhancing the model's ability to detect and predict critical issues.

4.3 Model Design and Workload Forecasting

For Phase 1, our predictive model consists of:

- Anomaly Detection Models:
    - Pod Failure Prediction: Implemented using an LSTM-based architecture.
    - Network and Resource Exhaustion Prediction: Our ANN model is enhanced with SMOTE to handle class imbalance and trained on features extracted from resource usage metrics.
- Time-Series Forecasting:
  We explore forecasting methods including SARIMA and Bi-LSTM, which predict future workload demand based on historical patterns. These methods help in proactive scaling decisions.

5. SYSTEM ARCHITECTURE

5.1 Overall Architecture

Our framework is a comprehensive solution designed to predict and remediate anomalies in Kubernetes clusters through a hierarchical hybrid model. This architecture organizes the system into multiple tiers based on usage intensity and resource criticality, allowing for more granular and responsive scaling decisions. At the lowest level, the Data Preprocessing Module cleanses, normalizes, and aggregates raw cluster metrics to remove noise and inconsistencies. Our experiments documented in the notebooks (data_preprocessing1.ipynb and data_preprocessing1_copy.ipynb) guided us in developing robust preprocessing pipelines.

Following preprocessing, the Feature Extraction Module derives key indicators, such as CPU and memory utilization ratios, differential metrics, and rolling statistics that capture temporal trends. These extracted features are vital for our subsequent modeling efforts. In the next tier, the Model Training Module employs a variety of machine learning algorithms to detect anomalies and forecast workload patterns. Our primary approach is to train an Artificial Neural Network (ANN) enhanced with SMOTE to mitigate class imbalance. Alongside, we explore alternative models including LSTM networks for capturing long-term temporal dependencies, Isolation Forest for unsupervised anomaly detection, and Random Forest for robust classification. These models operate in a hierarchical fashion where higher tiers refine predictions made at the lower levels based on usage hierarchy and criticality.

The Evaluation Module, implemented in evaluation.py and testing.py, rigorously assesses model performance using metrics such as accuracy, precision, recall, F1 score, and visualizations like ROC and precision-recall curves, ensuring that our predictions are both accurate and actionable. Complementing these is the Hybrid Scaling Controller, which integrates predictions from our forecasting and performance models to make real-time decisions on both horizontal scaling (adjusting pod counts) and vertical scaling (modifying resource allocations). Finally, the Monitoring and Deployment Module integrates with Kubernetes APIs and external tools such as Prometheus and Kafka, and leverages insights from the k8sgpt website, ensuring that scaling actions are executed seamlessly without service disruption.

5.2 Module Overview and Integration

The modules in our framework are tightly integrated into a hierarchical pipeline that reflects usage-based priority levels. The Preprocessing and Feature Extraction modules ensure that raw metrics are transformed into a structured, usable format. This data is then used by the Model Training Module, where our ANNTrainer (enhanced with SMOTE) and other models such as LSTM, Isolation Forest, and Random Forest are trained to predict anomalies in a tiered approach. The Evaluation Module then quantifies performance and provides visual insights into prediction quality. Concurrently, the Retrieval Module continuously collects real-time metrics from Prometheus, Kafka, and Kubernetes APIs, ensuring that our Hybrid Scaling Controller receives dynamic inputs to adjust scaling decisions across different hierarchy levels.

5.3 Unique Aspects of Our Approach

Our solution is unique due to its hierarchical hybrid model, which organizes autoscaling decisions based on usage priority and resource criticality. Unlike conventional methods that apply a uniform scaling policy, our approach dynamically combines horizontal and vertical scaling strategies across different tiers. The predictive workload forecasting component, particularly leveraging LSTM, anticipates workload fluctuations to trigger proactive scaling. The integration of multiple anomaly detection models, including an ANN enhanced with SMOTE, Isolation Forest, and Random Forest, allows our system to capture diverse patterns of abnormal behavior. Finally, our custom monitoring integration, which aggregates real-time metrics from multiple sources, ensures that scaling decisions are contextually prioritized based on the hierarchy of usage, optimizing resource utilization and maintaining service quality in dynamic Kubernetes environments.

6. IMPLEMENTATION

Below is an explanation of the key code snippets in each file that contribute to network issues detection for your report:

Below are key code snippets extracted from each file that directly relate to network issues detection. Each snippet is accompanied by a brief explanation.

The hybrid.ipynb, hybrid1.ipynb and hybrid1_backup.ipynb notebook are still experimental which uses the unique approach of Graph Neural Networks(GNN) and Long-Short Term Memory(LSTM) and Reinforcement learning approach for solving the network anomaly issue of kubernetes cluster.

Network or Connectivity Issues Detection:

1. data_preprocessing.py

Snippet: Converting IP Addresses to Numeric Format
This snippet converts both source and destination IP addresses into a numeric representation. This conversion enables downstream models to use IP-based features for detecting network anomalies.

```
def ip_to_numeric(self, ip_str):
    """
    Converts an IP address string to an integer.
    Works for both IPv4 and IPv6 addresses.
    If conversion fails, returns NaN.
    """
    try:
        return int(ipaddress.ip_address(ip_str))
    except ValueError:
        return np.nan


def convert_ips(self, df):
    """
    Converts both source and destination IP addresses to numeric
format.
    """
```

```
    df['source_ip_numeric'] =
df['_source_source_ip'].apply(self.ip_to_numeric)
    df['destination_ip_numeric'] =
df['_source_destination_ip'].apply(self.ip_to_numeric)
    return df
```

2. evaluation.py

Snippet: Plotting the ROC Curve
This snippet generates a ROC curve to evaluate the performance of the network anomaly detection model, illustrating the trade-off between the true positive and false positive rates.

```
def plot_roc_curve(self):
    """
    Plots the ROC curve and prints the AUC.
    """
    if self.y_pred_proba is None:
        print("No probability scores provided; cannot plot ROC
curve.")
        return
    fpr, tpr, thresholds = roc_curve(self.y_true,
self.y_pred_proba)
    roc_auc = auc(fpr, tpr)
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, label=f'ROC Curve (AUC = {roc_auc:.2f})',
color='blue')
    plt.plot([0, 1], [0, 1], 'r--', label='Random Chance')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend(loc='lower right')
    plt.tight_layout()
    plt.show()
    print("ROC AUC:", roc_auc)
```

3. feature_extraction_collection.py

Snippet: Mapping Protocols to Numeric Values
This snippet maps textual protocol identifiers (such as "tcp" and "udp") to numeric values. This conversion is crucial to include protocol information as a feature when analyzing network traffic.

```
def add_protocol_numeric(self, df):
    """
    Maps protocol strings in '_source_network_transport' to
numeric values and creates a new column 'protocol_numeric'.
    Returns the updated DataFrame.
    """
    protocol_mapping = {'tcp': 0, 'udp': 1}
    try:
        df['protocol_numeric'] =
df['_source_network_transport'].map(protocol_mapping).fillna(-1).
astype(int)
    except KeyError as e:
        raise KeyError(f"Expected column
'_source_network_transport' not found: {e}")
    return df
```

Snippet: Port Classification
This snippet categorizes port numbers to detect any unusual port usage which may indicate network misconfigurations or attacks.

```
def classify_port_range(self, port):
    """
    Classifies the port number into one of three categories:
      - 'Well-Known' for ports 0 to 1023,
      - 'Registered' for ports 1024 to 49151,
      - 'Ephemeral' for ports 49152 to 65535.
    If the port is missing or invalid, returns 'Unknown'.
    """
    try:
        p = int(port)
```

```
    except (ValueError, TypeError):
        return 'Unknown'
    if 0 <= p <= 1023:
        return 'Well-Known'
    elif 1024 <= p <= 49151:
        return 'Registered'
    elif 49152 <= p <= 65535:
        return 'Ephemeral'
    else:
        return 'Out of Range'
```

4. retrieval.py

Snippet: Setting Up Prometheus and Kafka Consumer for Streaming Network Data
This snippet demonstrates how to retrieve real-time network metrics from Prometheus and process live network flow data from Kafka. This data is fundamental for detecting network issues such as spikes in network traffic or unexpected changes.

```
def setup_prometheus():
    # Connect to Prometheus
    prometheus_url = "http://prometheus:9090"  # Replace with
your Prometheus URL
    return PrometheusConnect(url=prometheus_url,
disable_ssl=True)

def setup_kafka_consumer(topic):
    # Set up Kafka consumer for network flow data
    consumer = KafkaConsumer(
        topic,
        bootstrap_servers=['kafka:9092'],  # Replace with your
Kafka bootstrap servers
        auto_offset_reset='latest',
        enable_auto_commit=True,
        group_id='k8s-monitoring',
        value_deserializer=lambda x:
json.loads(x.decode('utf-8'))
    )
    return consumer
```

```
def process_streaming_data(api_client, prometheus, consumer):
    # Process Kubernetes API metrics
    v1 = kubernetes.client.CoreV1Api(api_client)
    pod_metrics = v1.list_pod_for_all_namespaces().items
    # Process Prometheus metrics
    network_metrics =
prometheus.custom_query(query='container_network_receive_bytes_to
tal')

    # Process Kafka messages for network flows
    for message in consumer:
        flow_data = message.value
        # Parse and aggregate data (example)
        df = pd.DataFrame([flow_data])
        # Here, you would feed df into your AI/ML model for
anomaly detection
        print(f"Processed flow at {datetime.now()}: {flow_data}")
```

5. testing.py

Snippet: Model Prediction for Network Anomalies

This snippet is part of the **ModelTester** class that predicts network anomalies. It processes the pre-scaled test data and applies a threshold to the model's probability outputs to generate binary anomaly flags.

```
def predict(self, X_new_scaled, threshold=0.5):
    """
    Uses the loaded model to predict anomalies on the new scaled
data.

    For:
     - ANN: predictions are probabilities; threshold is applied.
     - RF: predictions are binary labels and probabilities via
predict_proba.
     - IF: predictions are +1 (normal) and -1 (anomaly); convert
-1 to 1 (anomaly) and +1 to 0.

    Parameters:
```

```
        X_new_scaled (array): Scaled feature array.
        threshold (float): Classification threshold for ANN
(default 0.5).

    Returns:
        tuple: (binary_predictions, prediction_scores)
    """
    try:
        if self.model_type == 'ann':
            pred_probs = self.model.predict(X_new_scaled).ravel()
            predictions = (pred_probs >
threshold).astype("int32")
            return predictions, pred_probs
        elif self.model_type == 'rf':
            predictions = self.model.predict(X_new_scaled)
            try:
                pred_probs =
self.model.predict_proba(X_new_scaled)[:, 1]
            except Exception:
                pred_probs = None
            return predictions, pred_probs
        elif self.model_type == 'if':
            pred = self.model.predict(X_new_scaled)
            # IsolationForest: -1 indicates anomaly, +1 indicates
normal
            predictions = np.where(pred == -1, 1, 0)
            try:
                # decision_function returns anomaly scores; more
negative means more anomalous.
                pred_scores =
self.model.decision_function(X_new_scaled)
            except Exception:
                pred_scores = None
            return predictions, pred_scores
    except Exception as e:
        print(f"Error during prediction: {e}")
        raise
```

6. training.py

Snippet: Building the ANN Model for Network Anomaly Detection

In this snippet from the `ANNTrainer` class, a deep neural network is defined with dropout and L2 regularization. This architecture is tuned to learn complex patterns in network traffic data that might signal network issues.

```
def build_model(self, input_dim):
    """
    Builds the ANN model architecture.
    """
    model = Sequential()
    # Input layer + first hidden layer with L2 regularization and
dropout
    model.add(Dense(32, input_dim=input_dim, activation='relu',
kernel_regularizer=l2(0.01)))
    model.add(Dropout(0.2))
    # Second hidden layer
    model.add(Dense(16, activation='relu',
kernel_regularizer=l2(0.01)))
    model.add(Dropout(0.2))
    # Third hidden layer
    model.add(Dense(8, activation='relu',
kernel_regularizer=l2(0.1)))
    model.add(Dropout(0.2))
    # Output layer
    model.add(Dense(1, activation='sigmoid'))

    # Compile the model using Nadam optimizer
    model.compile(loss='binary_crossentropy',
optimizer=Nadam(learning_rate=0.001), metrics=['accuracy'])
    self.model = model
    return model
```

Data preprocessing and feature extraction ensure that raw network logs and metrics are transformed into a format suitable for anomaly detection.

The training and testing pipelines build, evaluate, and deploy models (like the ANN) to predict network failures such as pod crashes, resource exhaustion, or connectivity issues.

Real-time retrieval of network metrics from Kubernetes, Prometheus, and Kafka bridges the gap between static analysis and dynamic monitoring, enabling prompt detection and response.

This modular design not only streamlines the development process but also allows targeted improvements to specific components of the network anomaly detection pipeline.

Resource Exhaustion(CPU, Memory) Anomaly Detection:

1. Import Necessary Libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (classification_report, f1_score,
                             precision_recall_curve, roc_curve, auc,
                         confusion_matrix, PrecisionRecallDisplay)
from sklearn.ensemble import StackingClassifier
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import SelectFromModel
from sklearn.base import clone
from sklearn.calibration import calibration_curve
```

Explanation

This code imports all necessary libraries required for data preprocessing, feature engineering, model training, evaluation, and visualization.

---

2. Load and Preprocess Data

```
def load_and_preprocess(filepath):
    df = pd.read_csv(filepath)

    # Ensure numeric columns are properly typed
      numeric_cols = ['cpu_usage', 'cpu_limit', 'memory_usage',
'memory_limit', 'restart_count']
    for col in numeric_cols:
        df[col] = pd.to_numeric(df[col], errors='coerce')

    # Derived features
    df['cpu_utilization'] = df['cpu_usage'] / df['cpu_limit']
            df['memory_utilization']   =   df['memory_usage']   /
df['memory_limit']
    df['cpu_request_diff'] = df['cpu_usage'] - df['cpu_request']
           df['memory_request_diff']   =   df['memory_usage']   -
df['memory_request']
    df['cpu_request_ratio'] = df['cpu_request'] / df['cpu_limit']
             df['cpu_mem_interaction']   =   df['cpu_usage']   *
df['memory_usage']

    # Rolling Statistics (No Data Leakage)
    window_size = 7
    for col in ['cpu_usage', 'memory_usage']:
         df[f'{col}_var'] = df[col].shift(1).rolling(window_size,
min_periods=1).var()
        df[f'{col}_ema'] = df[col].shift(1).ewm(span=5).mean()
```

```
        df[f'{col}_min'] = df[col].shift(1).rolling(window_size,
min_periods=1).min()
        df[f'{col}_max'] = df[col].shift(1).rolling(window_size,
min_periods=1).max()

    # Lag Features
    df['cpu_usage_lag1'] = df['cpu_usage'].shift(1)
    df['memory_usage_lag1'] = df['memory_usage'].shift(1)

    return df
```

Explanation

- Reads the dataset from a CSV file.
- Converts selected columns to numeric type.
- Creates derived features for CPU and memory utilization.
- Computes rolling statistics (variance, exponential moving average, min, max) to capture past trends.
- Adds lag features for CPU and memory usage to model sequential behavior.

---

3. Data Preprocessing for Anomaly Detection

```
def preprocess_data(df_split):
    window_size = 7
    processed = df_split.copy()
    for metric in ['cpu', 'memory']:
                                    processed[f'{metric}_ma']    =
processed[f'{metric}_usage'].shift(1).rolling(window_size,
min_periods=1).mean()
                                   processed[f'{metric}_std']    =
processed[f'{metric}_usage'].shift(1).rolling(window_size,
min_periods=1).std()

    # Anomaly detection labels
```

```
        processed['cpu_anomaly'] = (processed['cpu_usage'] >
(processed['cpu_ma'] + 1.5 * processed['cpu_std'])).astype(int)
    processed['memory_anomaly'] = (processed['memory_usage'] >
(processed['memory_ma'] + 1.5 *
processed['memory_std'])).astype(int)

    return processed.dropna()
```

Explanation

- Computes rolling mean and standard deviation for CPU and memory usage.
- Defines anomalies as values exceeding the mean + 1.5 times the standard deviation.

---

4. Splitting Data into Training and Testing Sets

```
df=
load_and_preprocess("kubernetes_resource_allocation_dataset.csv")
train_df = preprocess_data(df.iloc[:int(0.8*len(df))])
test_df = preprocess_data(df.iloc[int(0.8*len(df)):])
```

Explanation

- Loads the dataset and applies preprocessing.
- Splits the data into 80% training and 20% testing sets.

---

5. Feature Scaling

```
def scale_features(X, scaler=None, fit=False):
    if fit:
        scaler = StandardScaler().fit(X)
    scaled = scaler.transform(X)
    return pd.DataFrame(scaled, columns=X.columns), scaler
```

Explanation

- Uses StandardScaler to normalize the feature values for better model performance.

---

6. Training and Evaluating the Model

```
def train_and_evaluate_cpu_optimized(target='cpu_anomaly'):
    X_train = train_df[features]
    X_test = test_df[features]
    y_train = train_df[target]
    y_test = test_df[target]

    X_train_scaled, scaler = scale_features(X_train, fit=True)
    X_test_scaled, _ = scale_features(X_test, scaler)

            skf   =   StratifiedKFold(n_splits=5,   shuffle=True,
random_state=42)
    best_model, best_threshold = None, 0.5

    for train_idx, val_idx in skf.split(X_train_scaled, y_train):
                X_tr,  y_tr  =  X_train_scaled.iloc[train_idx],
y_train.iloc[train_idx]
                X_val,  y_val  =  X_train_scaled.iloc[val_idx],
y_train.iloc[val_idx]

        smote = SMOTE(sampling_strategy=0.6, random_state=42)
        X_res, y_res = smote.fit_resample(X_tr, y_tr)

        model = StackingClassifier([
                        ('xgb',  XGBClassifier(n_estimators=150,
learning_rate=0.05, max_depth=3)),
                      ('lgbm',  LGBMClassifier(n_estimators=150,
learning_rate=0.05, max_depth=3))
        ], final_estimator=LogisticRegression(max_iter=2000))
```

```
        model.fit(X_res, y_res)
        val_probs = model.predict_proba(X_val)[:, 1]
        y_pred = (val_probs >= best_threshold).astype(int)

    test_probs = model.predict_proba(X_test_scaled)[:, 1]
    y_test_pred = (test_probs >= best_threshold).astype(int)

    print(classification_report(y_test, y_test_pred))
```

Explanation

- Uses StackingClassifier with XGBoost and LightGBM.
- Uses Stratified K-Fold for cross-validation.
- Applies SMOTE for handling class imbalance.
- Evaluates the model using precision, recall, and AUC.

---

7. Running the Model

train_and_evaluate_cpu_optimized('cpu_anomaly')
train_and_evaluate_cpu_optimized('memory_anomaly')

Explanation

- Trains and evaluates the model for both CPU and memory anomalies.

This document provides a structured, step-by-step guide to implementing Kubernetes resource anomaly detection using AI/ML.

.

7. RESULTS

These were the results for network anomaly detection when trained using ANN. Other models used were Random Forest Classifier and Isolation Forest which gave less accuracy when trained on test dataset but ANN(Artificial Neural Network) gave an accuracy of 74% on one dataset and 91% on the second testing dataset. The problem with the traditional models i.e, Random Forest Classifier and Isolation Forest Classifier is that they get easily overfitted with the dataset dropping its accuracy for testing data which is not good at real time.

When we used the SMOTE technique the accuracy reduced to 92% on the training data.The precision and recall will get reduced. The below results is without oversampling which creates synthetic examples by interpolating between existing minority class samples.

The below results is for benign dataset without dropout and regularization. We did this because there was no malicious data in a dataset. In another malicious dataset it consists of both malicious and benign data. For malicious dataset we get an accuracy of 97%.

Network or Connectivity Issues Detection:

Epoch 1/15

39083/39083 [==============================] - 39s 991us/step - loss: 0.0396 - accuracy: 0.9870 - val_loss: 0.0278 - val_accuracy: 0.9902

Epoch 2/15

39083/39083 [==============================] - 64s 2ms/step - loss: 0.0248 - accuracy: 0.9916 - val_loss: 0.0251 - val_accuracy: 0.9919

Epoch 3/15

39083/39083 [==============================] - 59s 2ms/step - loss: 0.0221 - accuracy: 0.9925 - val_loss: 0.0209 - val_accuracy: 0.9929

Epoch 4/15

39083/39083 [==============================] - 57s 1ms/step - loss: 0.0196 - accuracy: 0.9932 - val_loss: 0.0180 - val_accuracy: 0.9934

Epoch 5/15

39083/39083 [==============================] - 51s 1ms/step - loss: 0.0180 - accuracy: 0.9936 - val_loss: 0.0185 - val_accuracy: 0.9938

Epoch 6/15

39083/39083 [==============================] - 49s 1ms/step - loss: 0.0172 - accuracy: 0.9938 - val_loss: 0.0167 - val_accuracy: 0.9937

Epoch 7/15

```
39083/39083 [==============================] - 49s 1ms/step - loss: 0.0166 - accuracy: 0.9940 - val_loss: 0.0155 - val_accuracy: 0.9944

Epoch 8/15

39083/39083 [==============================] - 49s 1ms/step - loss: 0.0160 - accuracy: 0.9942 - val_loss: 0.0163 - val_accuracy: 0.9944

Epoch 9/15

39083/39083 [==============================] - 49s 1ms/step - loss: 0.0159 - accuracy: 0.9942 - val_loss: 0.0155 - val_accuracy: 0.9945

Epoch 10/15

39083/39083 [==============================] - 53s 1ms/step - loss: 0.0156 - accuracy: 0.9943 - val_loss: 0.0177 - val_accuracy: 0.9938

Epoch 11/15

39083/39083 [==============================] - 52s 1ms/step - loss: 0.0155 - accuracy: 0.9944 - val_loss: 0.0142 - val_accuracy: 0.9947

Epoch 12/15

39083/39083 [==============================] - 50s 1ms/step - loss: 0.0152 - accuracy: 0.9944 - val_loss: 0.0146 - val_accuracy: 0.9946

Epoch 13/15

39083/39083 [==============================] - 50s 1ms/step - loss: 0.0148 - accuracy: 0.9945 - val_loss: 0.0142 - val_accuracy: 0.9949

Epoch 14/15

39083/39083 [==============================] - 50s 1ms/step - loss: 0.0147 - accuracy: 0.9945 - val_loss: 0.0136 - val_accuracy: 0.9948

Epoch 15/15

39083/39083 [==============================] - 50s 1ms/step - loss: 0.0145 - accuracy: 0.9946 - val_loss: 0.0136 - val_accuracy: 0.9950

12214/12214 [==============================] - 9s 755us/step

Accuracy: 0.9948518894742633
Precision: 0.9727939678011005
Recall: 0.9281547734785145
F1-Score: 0.9499502487562189
[[369717    534]
 [  1478  19094]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    370251
           1       0.97      0.93      0.95     20572

    accuracy                           0.99    390823
   macro avg       0.98      0.96      0.97    390823
weighted avg       0.99      0.99      0.99    390823
Model: "sequential_3"
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_12 (Dense)            (None, 32)                320

 dropout_3 (Dropout)         (None, 32)                0

 dense_13 (Dense)            (None, 16)                528

 dropout_4 (Dropout)         (None, 16)                0

 dense_14 (Dense)            (None, 8)                 136

 dropout_5 (Dropout)         (None, 8)                 0

 dense_15 (Dense)            (None, 1)                 9

=================================================================
Total params: 993 (3.88 KB)
Trainable params: 993 (3.88 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

The files data preprocessing1.ipynb and data preprocessing_copy.ipynb document are exploratory analysis and experimentation. In these notebooks, our team conducted extensive trials with various models to identify which one best leveraged the dataset's features and delivered the highest accuracy across multiple performance metrics.When deploying the model for detecting real-time data we ensure that the model is trained on more number of parameters to improve the performance of the model in various aspects.

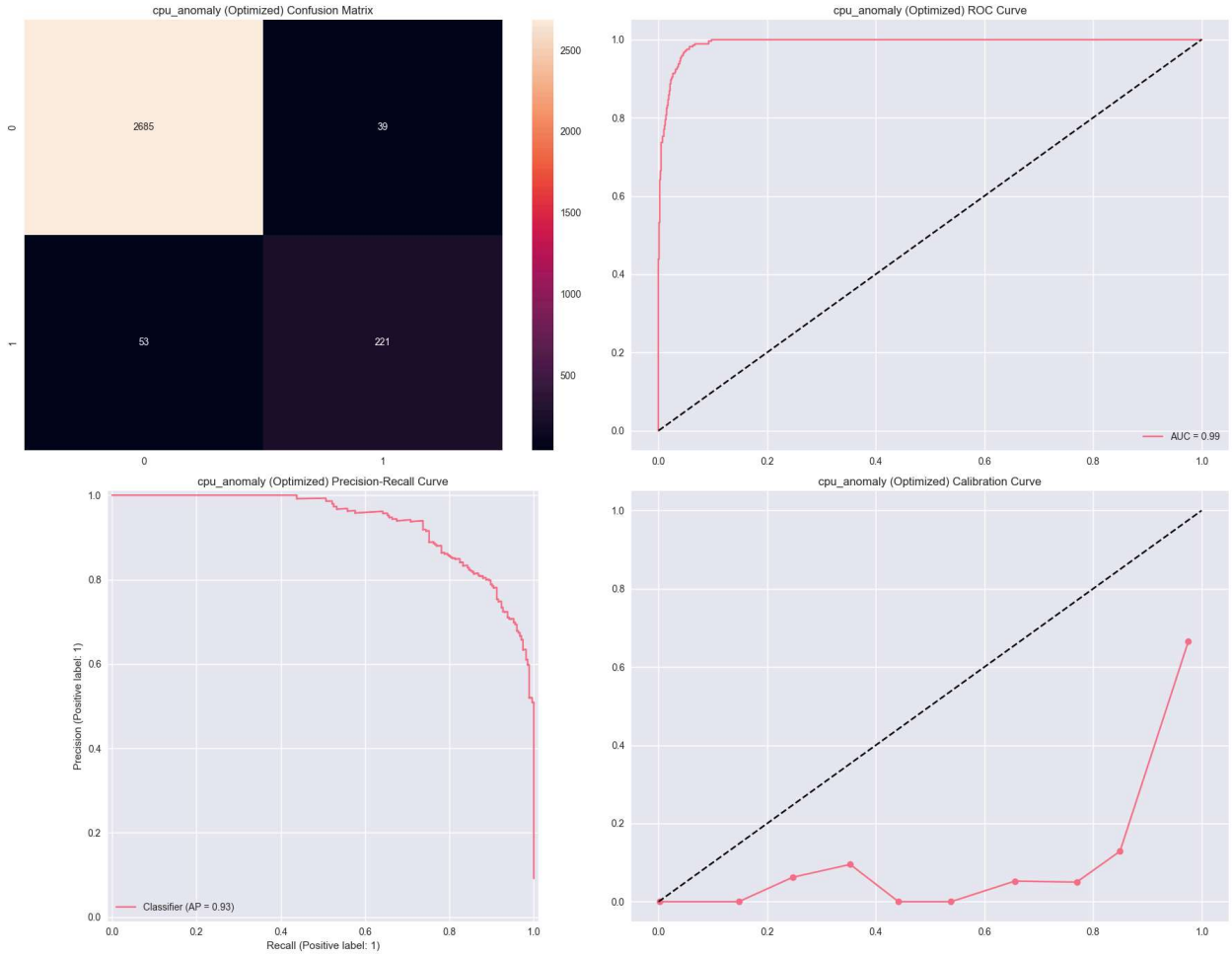
During the remediation phase, we will try our best to improve the accuracy by training the model on real world data

Resource Exhaustion(CPU, Memory) Anomaly Detection:

```
=== Optimized CPU Anomaly Detection ===
              precision    recall  f1-score   support

           0       0.98      0.99      0.98      2724
           1       0.85      0.81      0.83       274

    accuracy                           0.97      2998
   macro avg       0.92      0.90      0.91      2998
weighted avg       0.97      0.97      0.97      2998
```

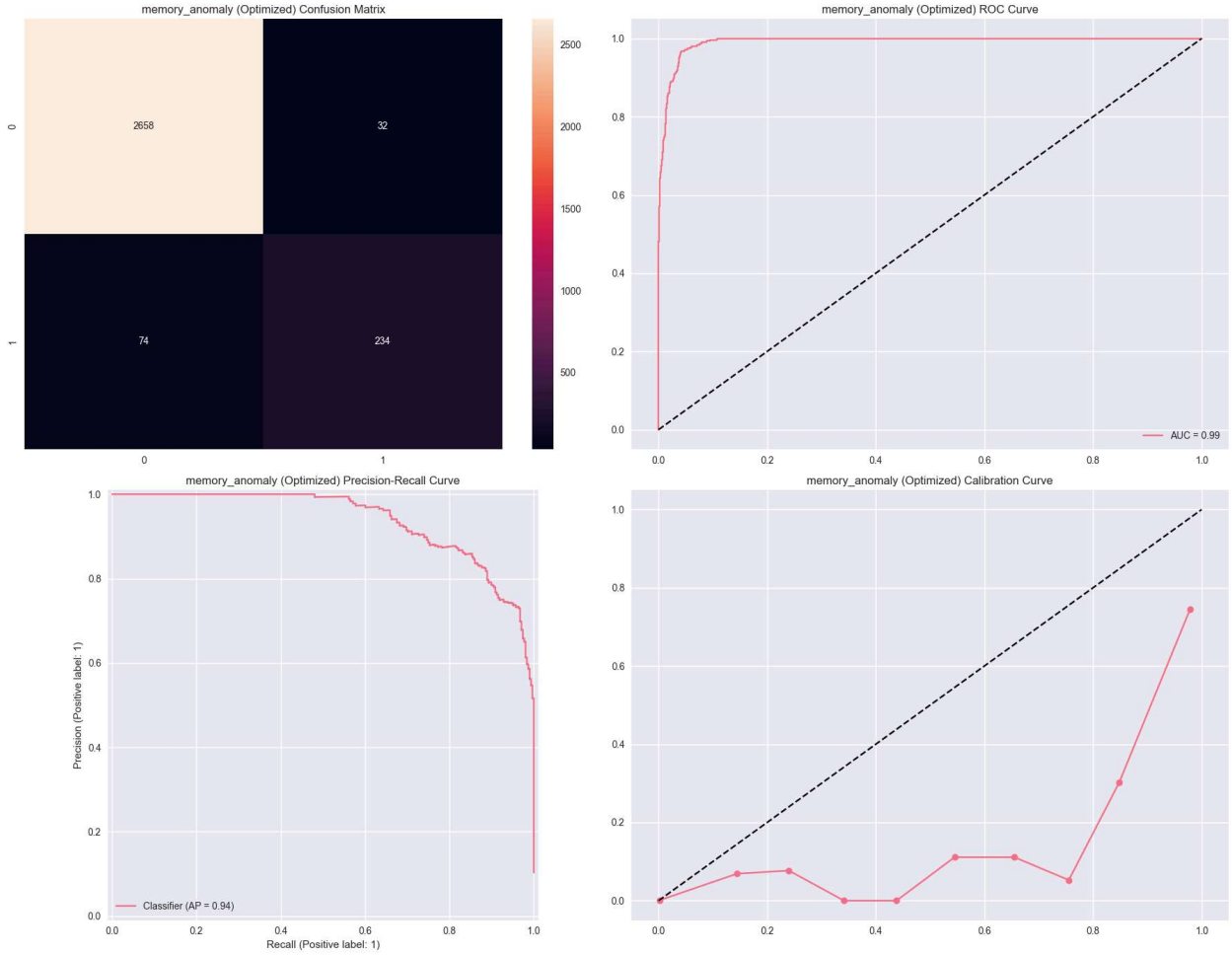


.

```
=== Optimized Memory Anomaly Detection ===
              precision    recall  f1-score   support

           0       0.97      0.99      0.98      2690
           1       0.88      0.76      0.82       308

    accuracy                           0.96      2998
   macro avg       0.93      0.87      0.90      2998
weighted avg       0.96      0.96      0.96      2998
```



8. CONCLUSION AND FUTURE WORK

Looking ahead, our future work aims to evolve this framework into a comprehensive, one-stop solution for managing all aspects of Kubernetes cluster health. We plan to incorporate AI agents such as n8n and LangGraph to create an intelligent orchestration infrastructure. In this envisioned system, an AI agent will continuously monitor and direct incoming metrics based on the identified issues to the appropriate predictive model. The specialized models will process these metrics and generate remediation actions, which will then be fed back into the system through the AI agent, creating a dynamic feedback loop for continuous optimization. Additionally, we aim to expand our predictive models to incorporate more granular application-level metrics, enhancing precision in anomaly detection. Enhancing real-time data ingestion and reducing the latency of scaling decisions are also critical areas for improvement. Finally, we plan to extend our framework to support multi-cloud and edge computing environments and explore reinforcement learning approaches to develop adaptive scaling policies. This comprehensive future direction will further automate and optimize Kubernetes cluster management, ensuring robust performance across a wide range of operational scenarios.

9. REFERENCES

1. Dinh-Dai Vu, Minh-Ngoc Tran, and Younghan Kim, "Predictive Hybrid Autoscaling for Containerized Applications," *IEEE Access*, vol. 10, 2022.
2. Akash Puliyadi Jegannathan, Rounak Saha, and Sourav Kanti Addya, "A Time Series Forecasting Approach to Minimize Cold Start Time in Cloud-Serverless Platform," *IEEE*, 2022.
3. J. Mary Ramya Poovizhi and Dr. R. Devi, "Performance Analysis of Cloud Hypervisor Using Network Package Workloads in Virtualization," *IEEE Conference SMART–2023*, 2023.
4. Mendeley Dataset, "Kubernetes Resource Metrics Dataset," Available: https://data.mendeley.com/datasets/ks9vbv5pb2/1
5. Kaggle Dataset, "Kubernetes Resource and Performance Metrics Allocation," Available: https://www.kaggle.com/datasets/nickkinyae/kubernetes-resource-and-performancemetricsallocation?resource=download
6. 4TU Dataset, "AssureMOSS Kubernetes Run-time Monitoring Dataset," Available: https://data.4tu.nl/articles/dataset/AssureMOSS_Kubernetes_Run-time_Monitoring_Dataset/20463687
7. k8sgpt Website, [online]. Available: https://www.k8sgpt.io
8. Additional literature on state machine approaches in autoscaling (State Machine Research Paper, details provided during hackathon discussions).