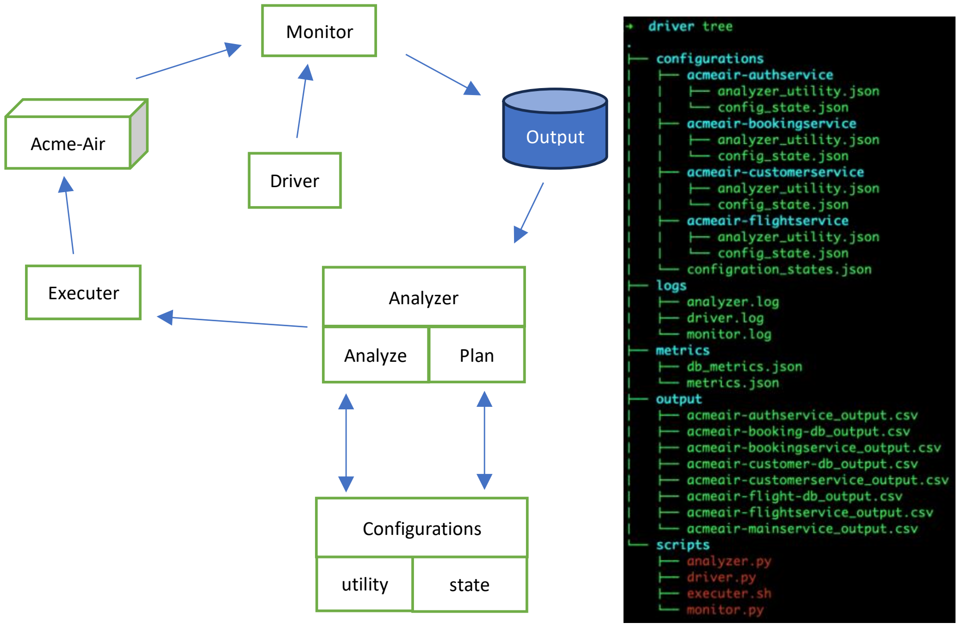
This implementation to make the Acme-Air application self-adaptive follows industry standards and best practices, exemplified by its well-structured codebase with separate directories for configuration, logs, metrics, output, and scripts, enhancing manageability. The monitor script collects real-time metrics data based on criteria specified in JSON files in the metrics folder, while the analyzer script calculates utility values for each configuration for optimal configuration selection. The executer script efficiently executes adaptation plans within the MAPE-K adaptation loop. I have added a mechanism to handle the scenario when the pods are scaling, minimizing disruptions during deployment updates and incorporating a waiting period for Kubernetes cluster stabilization. The organization of your codebase is meticulous and follows industry standards, mirroring the structure often seen in professional software development projects. Each directory serves a distinct purpose, making it easier to manage and navigate different aspects of our project.



(a) Architecture: MAPE Closed Loop

```
→  driver tree
.
├── configurations
│   ├── acmeair-authservice
│   │   ├── analyzer_utility.json
│   │   └── config_state.json
│   ├── acmeair-bookingservice
│   │   ├── analyzer_utility.json
│   │   └── config_state.json
│   ├── acmeair-customerservice
│   │   ├── analyzer_utility.json
│   │   └── config_state.json
│   ├── acmeair-flightservice
│   │   ├── analyzer_utility.json
│   │   └── config_state.json
│   └── configration_states.json
├── logs
│   ├── analyzer.log
│   ├── driver.log
│   └── monitor.log
├── metrics
│   ├── db_metrics.json
│   └── metrics.json
├── output
│   ├── acmeair-authservice_output.csv
│   ├── acmeair-booking-db_output.csv
│   ├── acmeair-bookingservice_output.csv
│   ├── acmeair-customer-db_output.csv
│   ├── acmeair-customerservice_output.csv
│   ├── acmeair-flight-db_output.csv
│   ├── acmeair-flightservice_output.csv
│   └── acmeair-mainservice_output.csv
└── scripts
    ├── analyzer.py
    ├── driver.py
    ├── executer.sh
    └── monitor.py
```

(b) Code Structure

Figure 1: Implementation Design

- In Figure 1(a) we see the architecture diagram of our self-adaptive system design for Acme-Air. The arrows show the control and information flow between modules. Components in Figure 1(b) are as follows:

- The "configurations" directory stored configuration files for various services within the system, each having "analyzer_utility.json" and "config_state.json" files. These are used by the analyzer module.

- The "logs" directory contains log files for different parts of the system, such as "analyzer.log," "driver.log," and "monitor.log."

- The "metrics" directory holds metric data in JSON format. These are used by the monitoring module.

- The "output" directory stores CSV files with output data of monitoring stage from different services.

- Lastly, the "scripts" directory contains Python scripts and a shell script for monitoring, analyzing, and executing the system.

Driver: The heart of your adaptation system is the driver program, which makes use of the Python **schedule** library. This library provides scheduling capabilities that ensure the adaptation code runs at regular intervals, precisely **every 5 minutes**. This scheduling is crucial because it enforces consistent monitoring and adaptation of your system. This regularity allows your system to stay up-to-date and responsive to changes in your managed application.

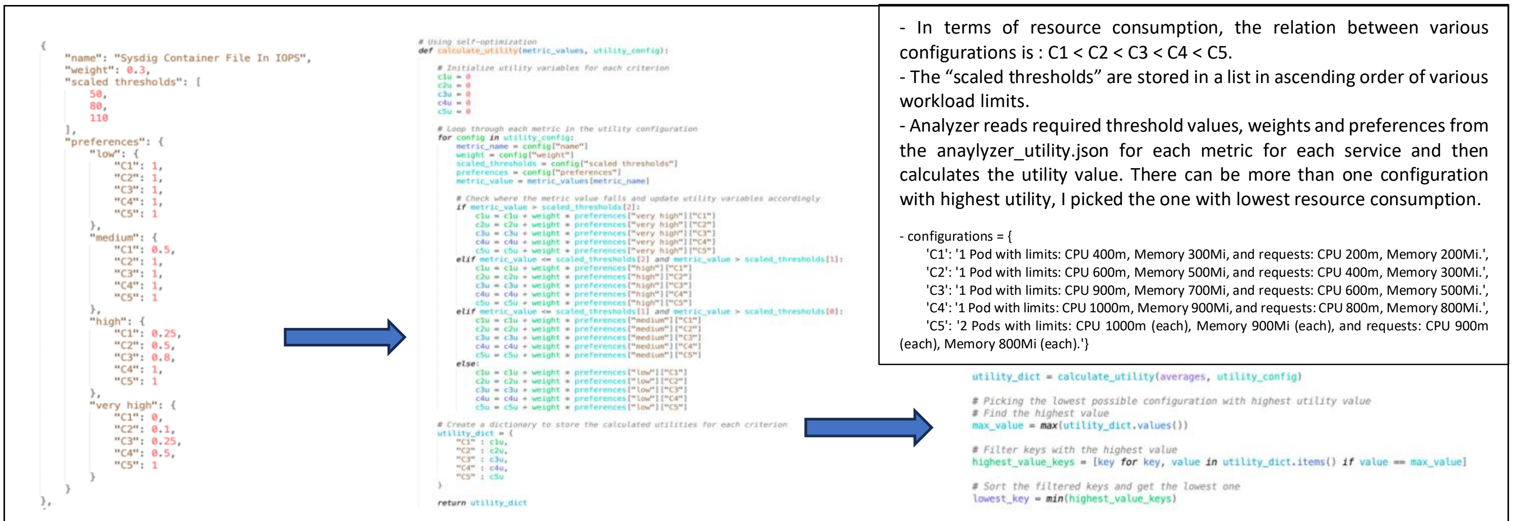
Monitor Script: The monitor script serves as the initial data collection point in your MAPE-K loop. It gathers relevant metrics data for the past 5 minutes and appends this information to output CSV files dedicated to each service or deployment. The choice of metrics to collect is determined by the criteria specified in JSON files located in the metrics folder. This component ensures that real-time data is consistently collected and stored for subsequent analysis, which is vital for making informed adaptation decisions.

Analyzer Script: The analyzer script is the core component of the MAPE-K loop, encompassing the Analysis and Planning phases. Its tasks are multifaceted:

- It reads the output data for services like authentication, booking, customer, and flight. These are services that require adaptation based on changing workloads.
- It calculates utility function values for each configuration. These calculations consider a variety of factors, including metric weights and preference settings. These factors are retrieved for each service from the analyzer utility JSON files situated in the configurations folder.
- The primary goal is to identify the configuration with the lowest resource requirement that best adapts to the current system conditions. This in-depth analysis and calculation are pivotal for making precise adaptation decisions.

I chose "Sysdig Container File In IOPS", "Sysdig Container Net HTTP Request Count", and "JMX JVM Heap Used" as key metrics for system analysis at this stage as a comprehensive approach to estimate the workload on our application. These metrics encompass different critical aspects of the system's behavior. "Sysdig Container File In IOPS" provides insights into the efficiency of file operations, reflecting the system's storage performance. "Sysdig Container Net HTTP Request Count" is crucial for understanding network activity and communication with external systems, helping gauge user demand and external dependencies. "JMX JVM Heap Used" monitors JVM heap memory usage, offering valuable information about memory management and potential issues, particularly in Java-based applications. By analyzing these metrics under various load conditions, it becomes possible to make well-informed configuration decisions, allowing for the optimization of resource allocation and performance by achieving our adaptation goals. This approach helps in identifying potential bottlenecks and making the necessary adjustments for an optimized system.

In the next page we will see various steps used by the analyzer to implement Analysis and Planning phase of MAPE-K adaptation loops. The idea was to implement an architecture where configuration changes happen in JSON files and the code need not be changed when we want to change weight or preferences for different metrics.



```
{
    "name": "Sysdig Container File In IOPS",
    "weight": 0.3,
    "scaled thresholds": [
        50,
        80,
        110
    ],
    "preferences": {
        "low": {
            "C1": 1,
            "C2": 1,
            "C3": 1,
            "C4": 1,
            "C5": 1
        },
        "medium": {
            "C1": 0.5,
            "C2": 1,
            "C3": 1,
            "C4": 1,
            "C5": 1
        },
        "high": {
            "C1": 0.25,
            "C2": 0.5,
            "C3": 0.8,
            "C4": 1,
            "C5": 1
        },
        "very high": {
            "C1": 0,
            "C2": 0.1,
            "C3": 0.25,
            "C4": 0.5,
            "C5": 1
        }
    }
},
```

(a) Sample metric in analyzer_utility.json

```
# Using self-optimization
def calculate_utility(metric_values, utility_config):

    # Initialize utility variables for each criterion
    c1u = 0
    c2u = 0
    c3u = 0
    c4u = 0
    c5u = 0

    # Loop through each metric in the utility configuration
    for config in utility_config:
        metric_name = config["name"]
        weight = config["weight"]
        scaled_thresholds = config["scaled thresholds"]
        preferences = config["preferences"]
        metric_value = metric_values[metric_name]

        # Check where the metric value falls and update utility variables accordingly
        if metric_value > scaled_thresholds[2]:
            c1u = c1u + weight * preferences["very high"]["C1"]
            c2u = c2u + weight * preferences["very high"]["C2"]
            c3u = c3u + weight * preferences["very high"]["C3"]
            c4u = c4u + weight * preferences["very high"]["C4"]
            c5u = c5u + weight * preferences["very high"]["C5"]
        elif metric_value <= scaled_thresholds[2] and metric_value > scaled_thresholds[1]:
            c1u = c1u + weight * preferences["high"]["C1"]
            c2u = c2u + weight * preferences["high"]["C2"]
            c3u = c3u + weight * preferences["high"]["C3"]
            c4u = c4u + weight * preferences["high"]["C4"]
            c5u = c5u + weight * preferences["high"]["C5"]
        elif metric_value <= scaled_thresholds[1] and metric_value > scaled_thresholds[0]:
            c1u = c1u + weight * preferences["medium"]["C1"]
            c2u = c2u + weight * preferences["medium"]["C2"]
            c3u = c3u + weight * preferences["medium"]["C3"]
            c4u = c4u + weight * preferences["medium"]["C4"]
            c5u = c5u + weight * preferences["medium"]["C5"]
        else:
            c1u = c1u + weight * preferences["low"]["C1"]
            c2u = c2u + weight * preferences["low"]["C2"]
            c3u = c3u + weight * preferences["low"]["C3"]
            c4u = c4u + weight * preferences["low"]["C4"]
            c5u = c5u + weight * preferences["low"]["C5"]

    # Create a dictionary to store the calculated utilities for each criterion
    utility_dict = {
        "C1" : c1u,
        "C2" : c2u,
        "C3" : c3u,
        "C4" : c4u,
        "C5" : c5u
    }

    return utility_dict
```

(b) Analyzer code calculating utility values

- In terms of resource consumption, the relation between various configurations is : C1 < C2 < C3 < C4 < C5.

- The "scaled thresholds" are stored in a list in ascending order of various workload limits.

- Analyzer reads required threshold values, weights and preferences from the anaylyzer_utility.json for each metric for each service and then calculates the utility value. There can be more than one configuration with highest utility, I picked the one with lowest resource consumption.

- configurations = {
  'C1': '1 Pod with limits: CPU 400m, Memory 300Mi, and requests: CPU 200m, Memory 200Mi.',
  'C2': '1 Pod with limits: CPU 600m, Memory 500Mi, and requests: CPU 400m, Memory 300Mi.',
  'C3': '1 Pod with limits: CPU 900m, Memory 700Mi, and requests: CPU 600m, Memory 500Mi.',
  'C4': '1 Pod with limits: CPU 1000m, Memory 900Mi, and requests: CPU 800m, Memory 800Mi.',
  'C5': '2 Pods with limits: CPU 1000m (each), Memory 900Mi (each), and requests: CPU 900m (each), Memory 800Mi (each).'}

```
utility_dict = calculate_utility(averages, utility_config)

# Picking the lowest possible configuration with highest utility value
# Find the highest value
max_value = max(utility_dict.values())

# Filter keys with the highest value
highest_value_keys = [key for key, value in utility_dict.items() if value == max_value]

# Sort the filtered keys and get the lowest one
lowest_key = min(highest_value_keys)
```

(c) Getting the optimal utility value with lowest resource consumption

Figure 2: Reference for Information flow in the Analyzer Code

Handling State Change: The **'handle_state_change'** function in analyzer.py is designed to efficiently manage deployment updates, minimizing unnecessary disruptions during system adaptations. It stores the current and previous states in the 'configuration_states.json' file within the 'configurations' folder. This function evaluates whether a configuration state update is needed based on specific conditions. Initially, it checks if the current configuration matches the latest configuration and returns **'False'** when they are the same, signifying no immediate update requirement. In cases where the current configuration differs from the previous, it updates the previous configuration, saves the changes to the JSON file, and also returns **'False'**. This strategy allows for a waiting period, enabling the cluster to stabilize after a configuration change. When the current and previous configurations match, it updates the current configuration with the latest one, retains the previous configuration, saves the changes to the JSON file, and returns **'True'**. This return value is used to decide whether an update should occur, providing the cluster with an opportunity to adapt before applying updates.

Executer Script: It is the deployment script implements the Execution part of MAPE self-adaptive system, implemented as a Bash script, is tasked with executing the actual adaptation plans. It takes care of updating resource requests and limits based on input arguments such as CPU and memory requirements and limits. Additionally, it manages the scaling of deployments to the desired number of pods within the Kubernetes cluster. The script is run by the analyzer script with arguments for CPU and Memory resources, number of pods and service name. The script interacts with the cluster using the **oc command-line** tool, ensuring that adaptation actions are executed efficiently and accurately.

I initially conducted a load test on the Acme-Air application without any adaptive features using JMeter, setting the thread count to 5, 25, and 100 to mimic low, medium, and high load settings, respectively, each for a duration of 15 minutes at C1 pod configuration (minimum CPU and Memory) for each microservice. Following this baseline assessment, I executed another round of testing under the same low, medium, and high load conditions, this time with self-adaptation mechanisms enabled through driver script, in order to assess whether the system could dynamically adjust to varying load intensities. This methodical approach allows for a comparative analysis between the baseline performance and the efficiency post-adaptation.

Figures 3 and 4 serve as valuable illustrations to help us grasp the positive impact of self-adaptation on resource utilization and system performance. These graphs are thoughtfully divided into two sections, represented by yellow (non-adaptive) and purple (adaptive) colours. Every 30 minutes (marked in the graph), the load settings changed to (non-adaptive - low, medium, high) and (adaptive - low, medium, high) roughly between 8:30 to 11:30. Interestingly, in both graphs, even though the load settings remained consistent for both the non-adaptive and adaptive scenarios, we observe that the self-adaptive system exhibits higher values for Heap Usage and HTTP Request Count. This observation aligns with our findings that pointed to increased server-side and network errors in the non-adaptive scenarios, as illustrated in Figures 5 and 7. This reduced responsiveness in the non-adaptive system contributes to lower resource utilization and the inability to handle incoming HTTP requests effectively. These metrics play a pivotal role in our analysis since their increased values act as crucial signals to the self-adaptive system, indicating the necessity for adaptive adjustments.

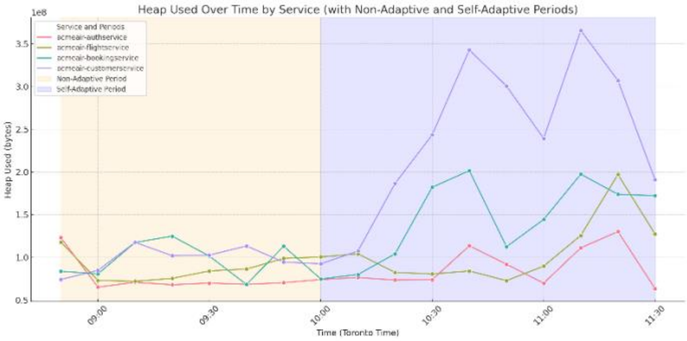


Figure 3: JVM Heap Used (before and after self-adaptive)

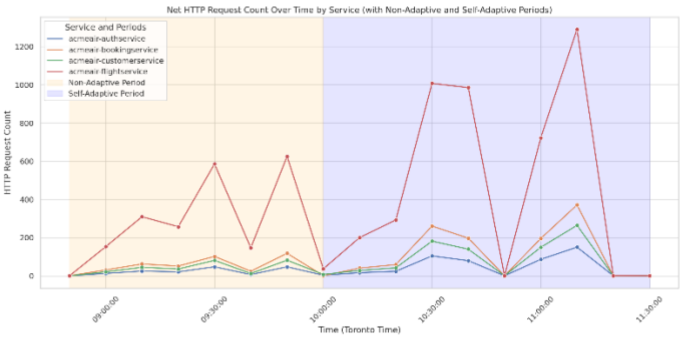


Figure 4: Net HTTP Request Count (before and after self-adaptive)

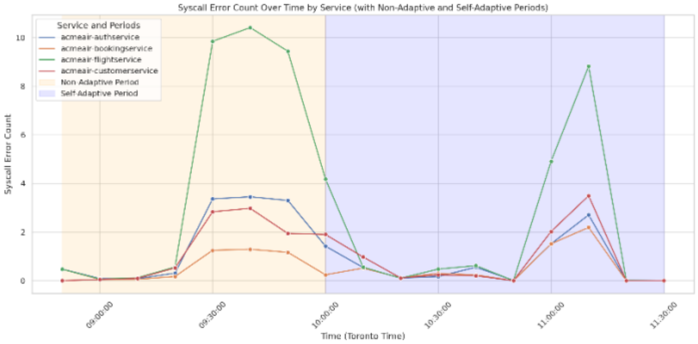


Figure 5: Syscall Error Count (before and after self-adaptive)

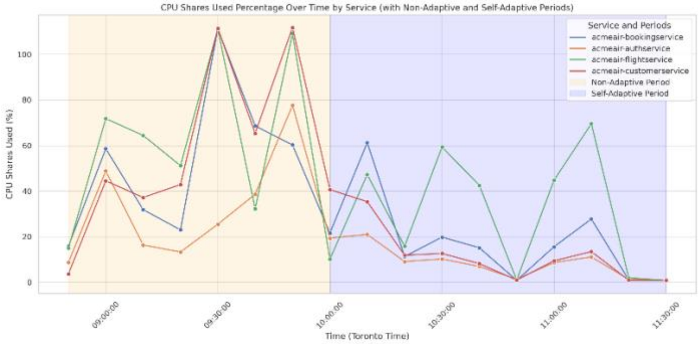
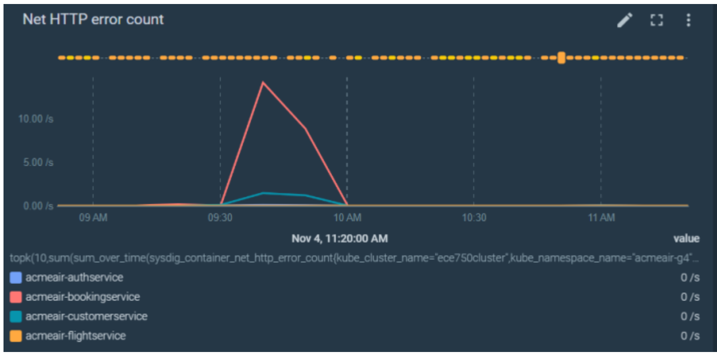


Figure 6: CPU shares used percentage (before and after self-adaptive)

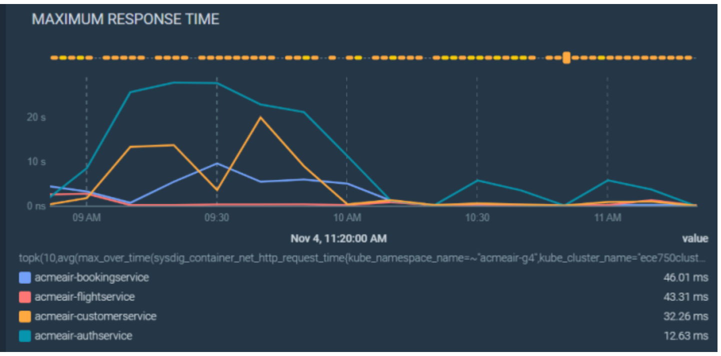


Nov 4, 11:20:00 AM value

topk(10,sum(sum_over_time(sysdig_container_net_http_error_count{kube_cluster_name="ece750cluster",kube_namespace_name="acmeair-g4"...

Nov 4, 11:20:00 AM	value
acmeair-authservice	0 /s
acmeair-bookingservice	0 /s
acmeair-customerservice	0 /s
acmeair-flightservice	0 /s

Figure 7: Net HTTP Error Count (before and after self-adaptive)



topk(10,avg(max_over_time(sysdig_container_net_http_request_time{kube_namespace_name=~"acmeair-g4",kube_cluster_name="ece750clust...

Nov 4, 11:20:00 AM	value
acmeair-bookingservice	46.01 ms
acmeair-flightservice	43.31 ms
acmeair-customerservice	32.26 ms
acmeair-authservice	12.63 ms

Figure 8: Maximum Response Time (before and after self-adaptive)

The pursuit of adaptation goals focused on reducing Syscall Error Counts and HTTP error Counts, while also decreasing maximum response times, represents a strategic approach to enhance the performance and reliability of the system. In each of the graphs we can observe the behaviour before 10 AM and after 10 AM, the self-adaptation code was ran at 10 AM. This effort reflects a well-informed strategy to optimize the system in several critical aspects:

1. Reducing Syscall Error Counts: Syscall errors are indicative of issues at the operating system level, such as resource constraints or interruptions in system calls. A decrease in Syscall Error Counts implies that the system's resource allocation and management have been improved, resulting in a smoother and more stable operation. It suggests that the system can efficiently utilize resources, reducing the likelihood of resource-related bottlenecks and system interruptions.
2. Reducing HTTP Error Counts: HTTP errors are detrimental to user experience and can signal problems in request handling or external service interactions. Lowering HTTP error Counts is crucial for delivering a seamless user experience. A reduction in HTTP errors indicates that the system can handle incoming requests and external dependencies more effectively, resulting in improved reliability and customer satisfaction.
3. Decreasing Maximum Response Times: Lowering maximum response times is essential for ensuring that the system is responsive and can serve users quickly. It directly impacts user experience, as shorter response times lead to higher user satisfaction. Achieving this goal suggests that the system's performance has been optimized to handle varying workloads efficiently, resulting in faster response times even during peak load conditions.

The analysis of the system's performance, particularly between 9 AM and 10 AM, provides valuable insights into the interplay between Syscall and HTTP Error Counts, Maximum Response Time, and CPU Shares Used percentage. During this period, the system experiences an evident surge in Syscall and HTTP Error Counts. These increasing error counts are closely linked to the behavior observed in the Maximum Response Time and CPU Shares Used percentage graphs. The system's resources become overburdened, and the clusters begin to exhibit signs of unhealthiness. This connection underscores the critical role that efficient resource management plays in maintaining system stability. However, a pivotal turning point occurs after 10 AM, where the system initiates its adaptation mechanisms. The decrease in error counts is a testament to the system's improved ability to handle requests and external dependencies effectively. Furthermore, the adaptation mechanisms bring Maximum Response Times below the critical threshold of 10 seconds, ensuring a responsive system even during peak load conditions. Notably, the CPU Shares Used percentage remains well within the 50% threshold post-adaptation, indicating efficient resource allocation and utilization.

The presence of unhealthy containers doesn't necessarily correlate with low or zero syscall or HTTP error counts. Instead, it often stems from issues related to failed Liveness and Readiness Probes. To address this, it's crucial to investigate and fine-tune probe conditions, such as timeout and failure thresholds, to ensure containers are correctly marked as healthy or unhealthy based on their actual operational status. The suggestion to wait for a cycle (5 minutes) after configuring the pods before considering the monitor script's readings in the analysis and adaptation decision proved to be highly valuable. This waiting period allows the cluster to stabilize after any updates, preventing potentially skewed data and enabling a more accurate and informed adaptation process. This adaptation strategy is a testament to the system's ability to respond effectively to changing conditions, ultimately leading to improved performance, resource management, and system stability.

In summary, the observations and analysis conducted support the affirmation that our system effectively attains its adaptation goals, following the MAPE-K adaptation loop approach for self-adaptive software systems.