Real-Time Performance Analysis and Optimization of System Metrics in Response to Traffic Fluctuations

Yibo Zhang, Jiaxin Ge
University of Waterloo
Email: y549zhan@uwaterloo.ca, j3ge@uwaterloo.ca

***Index Terms*—Real-time monitoring, performance analysis, resource allocation, system optimization, traffic fluctuations, CPU usage, memory management, feedback loop, latency control, quality of service (QoS).**

I. Introduction

WE, in this coursework, developed a driver program used in the keeping and analyzing of a system's performance in real-time. This is done by focusing on the pre-defined quality requirements in handling traffic fluctuations. For effective handling, we have built an automated process for tracking critical system metrics: CPU usage, memory usage, latency, throughput, and garbage collection time. By using these metrics, we implemented a feedback loop to investigate when adaptations were necessary. Our system was designed for self-optimization through the distribution of more resources and/or instantiating new pods, since doing so would enable it to handle growing traffic demands with less latency. The results reflected how well the system met performance criteria and how effectively adaptation mechanisms handled uncertainties.

II. Methodology

Our monitoring and analysis components are designed to address key uncertainties, such as fluctuating traffic that can result in congestion, delays, or even system failures. To mitigate these risks, we employ a feedback loop that continuously monitors the system and adjusts resource allocation as needed. The main source of uncertainty stems from traffic fluctuations, such as spikes in the number of requests or active concurrent users. These surges lead to higher CPU and memory consumption, causing system instability. We identified CPU usage, especially during peak traffic, as a key issue. The above scenario requires a standard workload, which, after successive iterations, should overload the system. The essential configuration parameters are the number of threads, users, duration, and ramp-up time. After obtaining the ideal workload and a comprehensive analysis, we established the following quality requirements for our system: The CPU should be less than 80% of any 5-minute period Memory usage should remain under 80% within the same time. - The response latency shall not exceed 1×10^7. These thresholds had been set such that Error Rate and QoS for a system facing varying workloads were kept within the marks.

We aimed at ensuring the performance of this system via dynamic resource allocation. The feedback loop was thus set such that the triggering of changes would be performed in case any of the set thresholds have been violated. The system could do the following:

- Increase CPU Resources
- Increase Memory Resources
- Increase the Number of Pods
- Optimize Garbage Collection

Accordingly, for locating system bottlenecks, we monitored the containers' performance by five parameters "cpu.used.percent," "memory.used.percent," "jvm.gc.global.time," "net.http.request.time," and "net.request.count.in" with a period of 5 minutes. Data collecting was done by means of JMeter scripts, which run constantly and generate requests at different loads. This output from the monitoring component was passed on to the analysis component for further processing.

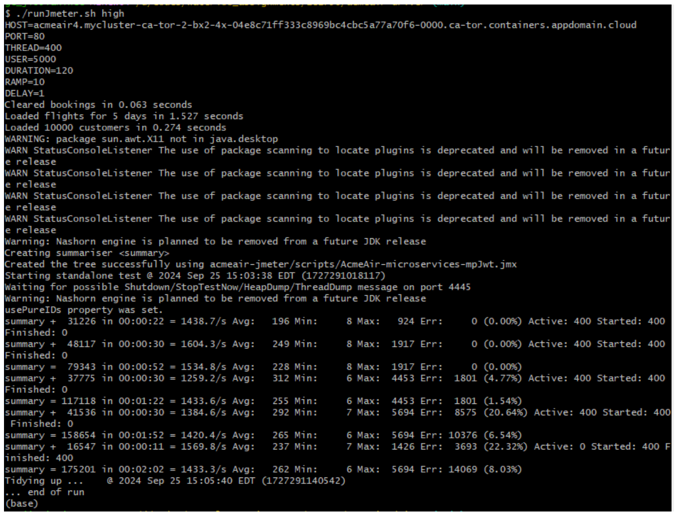
"cpu.used.percent" and "memory.used.percent" reflect resource usage, helping to prevent overload, while "jvm.gc.global.time" measures garbage collection efficiency to avoid memory leaks. Additionally, "net.http.request.time" represents latency, and "net.request.count.in" indicates traffic load, aiding in scaling decisions.

Monitoring too frequently burdens the system, while a low frequency may miss those critical spikes. This 5-minute interval captures the trend well, such as CPU and memory usage and application response times, hence the ability to proactively address the bottlenecks without affecting performance.

The monitoring component uses the "SdMonitorClient" of IBM Cloud for pulling data, which is persisted into JSON files for analysis. The analyzer reads those files and calculates utility scores for CPU, memory, latency, TPS, and garbage collection time to decide whether adaptation of the system is required. Data is aggregated at a per-service granularity, and the Analyzer class computes averages and other metrics that assist in optimizing resource usage based on thresholds.

III. Results

We first create low, medium, and high workloads that stress the system by using a JMeter script. Setting up a threshold workload is required to define the scope of these workloads. Figure 1 shows one possible high workload. Low, medium and high workload definitions are given in Table 1.



```
$ ./runJmeter.sh high
HOST=acmeair4.mycluster-ca-tor-2-bx2-4x-04e8c71ff333c8969bc4cbc5a77a70f6-0000.ca-tor.containers.appdomain.cloud
PORT=80
THREAD=400
USER=5000
DURATION=120
RAMP=10
DELAY=1
Cleared bookings in 0.063 seconds
Loaded flights for 5 days in 1.527 seconds
Loaded 10000 customers in 0.274 seconds
WARNING: package sun.awt.X11 not in java.desktop
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a futur
e release
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a futur
e release
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a futur
e release
WARN StatusConsoleListener The use of package scanning to locate plugins is deprecated and will be removed in a futur
e release
Warning: Nashorn engine is planned to be removed from a future JDK release
Creating summariser <summary>
Created the tree successfully using acmeair-jmeter/scripts/AcmeAir-microservices-mpJwt.jmx
Starting standalone test @ 2024 Sep 25 15:03:38 EDT (1727291018117)
Waiting for possible Shutdown/StopTestNow/HeapDump/ThreadDump message on port 4445
Warning: Nashorn engine is planned to be removed from a future JDK release
usePureIDs property was set.
summary +  31226 in 00:00:22 = 1438.7/s Avg:   196 Min:     8 Max:   924 Err:     0 (0.00%) Active: 400 Started: 400
Finished: 0
summary +  48117 in 00:00:30 = 1604.3/s Avg:   249 Min:     8 Max:  1917 Err:     0 (0.00%) Active: 400 Started: 400
Finished: 0
summary =  79343 in 00:00:52 = 1534.8/s Avg:   228 Min:     8 Max:  1917 Err:     0 (0.00%)
summary +  37775 in 00:00:30 = 1259.2/s Avg:   312 Min:     6 Max:  4453 Err:  1801 (4.77%) Active: 400 Started: 400
Finished: 0
summary = 117118 in 00:01:22 = 1433.6/s Avg:   255 Min:     6 Max:  4453 Err:  1801 (1.54%)
summary +  41536 in 00:00:30 = 1384.6/s Avg:   292 Min:     7 Max:  5694 Err:  8575 (20.64%) Active: 400 Started: 400
 Finished: 0
summary = 158654 in 00:01:52 = 1420.4/s Avg:   265 Min:     6 Max:  5694 Err: 10376 (6.54%)
summary +  16547 in 00:00:11 = 1569.8/s Avg:   237 Min:     7 Max:  1426 Err:  3693 (22.32%) Active: 0 Started: 400 F
inished: 400
summary = 175201 in 00:02:02 = 1433.3/s Avg:   262 Min:     6 Max:  5694 Err: 14069 (8.03%)
Tidying up ...    @ 2024 Sep 25 15:05:40 EDT (1727291140542)
... end of run
(base)
```

Fig. 1: Configuration that may lead to system failure

TABLE I: Configuration Parameters for Different Load Levels

Load Level	Threads	Users	Duration (s)	Ramp (s)	Delay (s)
Low	10	100	300	60	30
Medium	50	500	300	30	10
High	200	2000	300	10	1

Secondly, the analyzer read the metrics, computed averages, and compared these against the quality requirements. Thresholds breaching triggered the system to perform an adaptation using the utility function. For all the services, we conducted a number of experiments as highlighted in Fig. 2, 3 and 4 below, where we measured low, medium and high volumes of workload against their effects on the five metrics we choose.

The above pictures illustrate the effectiveness of our 5 metrics.

The strategy we implemented is designed to evaluate and optimize the performance of a system (such as a service or application) based on several key performance metrics: CPU usage, memory usage, latency, TPS (transactions per second), and GC (garbage collection) time[2]. The goal is to automatically suggest the best optimization strategy from several possible strategies by calculating an overall utility score for each strategy.

A. Strategy 1: Increase CPU Resources

Increasing CPU resources leads to a reduction in CPU usage and latency, while also slightly improving transactions per second (TPS). The garbage collection (GC) time remains unchanged. The utility score was calculated as follows:

$$\text{Utility}_1 = \text{calculate\_utility}\Bigg(\frac{\text{cpu}}{1.5}, \text{memory}, \frac{\text{latency}}{1.2}, \tag{1}$$

$$\text{tps} \times 1.1, \text{gc\_time}, "\mathit{cpu}"\Bigg) \tag{2}$$

B. Strategy 2: Increase Memory Resources

Increasing memory resources effectively reduces memory usage and GC time, leading to a slight improvement in TPS

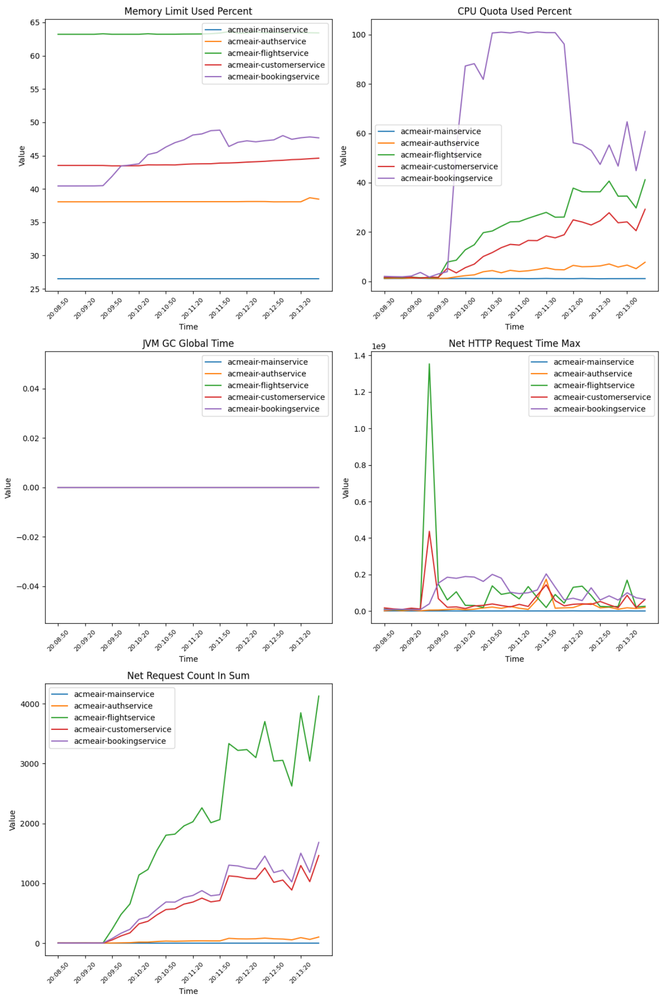


Fig. 2: Metrics with low workload within 5 minutes capture.

and a reduction in latency. The utility score for this strategy is calculated as:

$$\text{Utility}_2 = \text{calculate\_utility}\Bigg(\text{cpu}, \frac{\text{memory}}{2.0}, \frac{\text{latency}}{1.1}, \tag{3}$$

$$\text{tps} \times 1.05, \frac{\text{gc\_time}}{1.5}, "\mathit{memory}"\Bigg) \tag{4}$$

C. Strategy 3: Increase the Number of Pods

Adding more Pods significantly improves TPS and reduces latency, with only minor impacts on CPU and memory usage. The utility score for this strategy is given by:

$$\text{Utility}_3 = \text{calculate\_utility}\Bigg(\frac{\text{cpu}}{1.2}, \frac{\text{memory}}{1.2}, \frac{\text{latency}}{1.5}, \tag{5}$$

$$\text{tps} \times 1.5, \frac{\text{gc\_time}}{1.1}, "\mathit{pod}"\Bigg) \tag{6}$$

D. Strategy 4: Optimize Garbage Collection

Optimizing garbage collection time enhances overall performance by either optimizing the Java Virtual Machine (JVM)

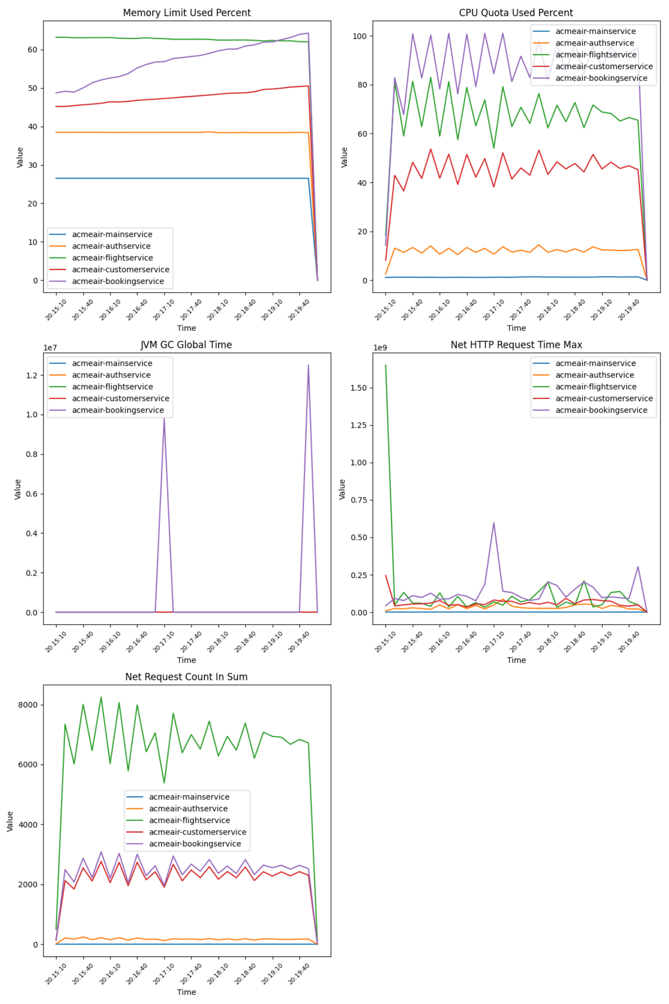


Fig. 3: Metrics with medium workload within 5 minutes capture.

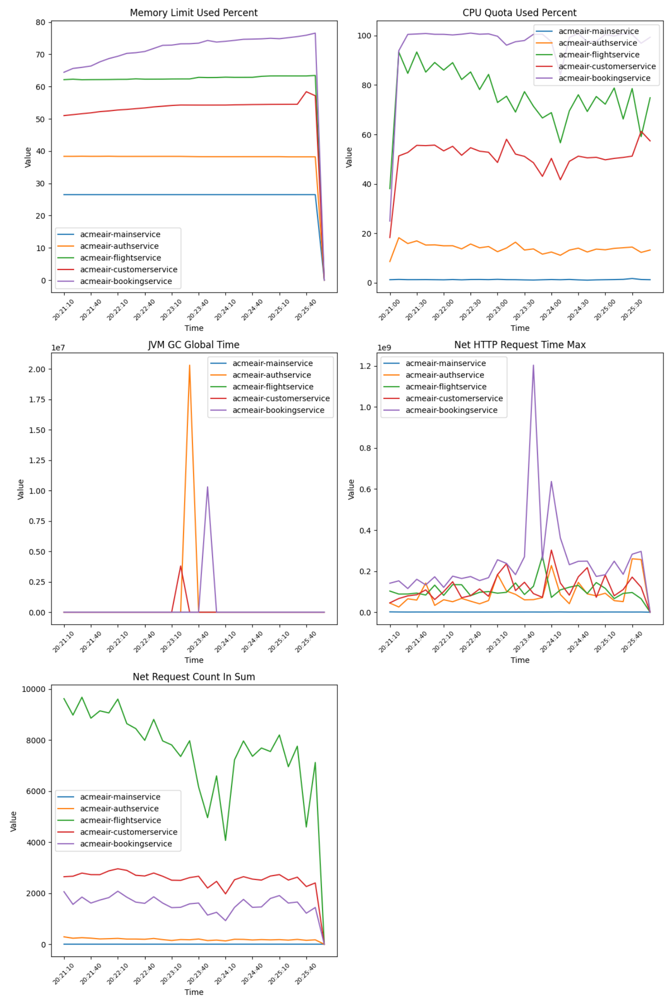


Fig. 4: Metrics with high workload within 5 minutes capture.

settings or allocating more memory. The utility score for optimizing GC time is calculated as:

$$\text{Utility}_4 = \text{calculate\_utility}\Bigg(\text{cpu}, \text{memory}, \frac{\text{latency}}{1.05}, \tag{7}$$

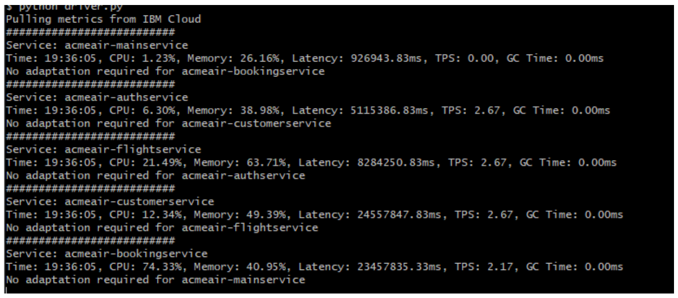
$$\text{tps}, \frac{\text{gc\_time}}{2.0}, \text{"}gc\text{"}\Bigg) \tag{8}$$

These utility scores reflect how well each metric is performing, with higher scores representing better performance. Example: If CPU usage is high (close to 100%), the utility score will be low (closer to 0). If CPU usage is low (e.g., 20%), the utility score will be high (closer to 1). Each metric is assigned a weight based on its importance to the overall system performance (e.g., CPU might have a higher weight than GC time). The total utility score for each strategy is calculated by summing the weighted utility scores for all metrics.

After calculating the utility score for each strategy, the strategy with the highest total utility score is selected as the best option. This decision is made based on which strategy provides the highest overall improvement in performance metrics.

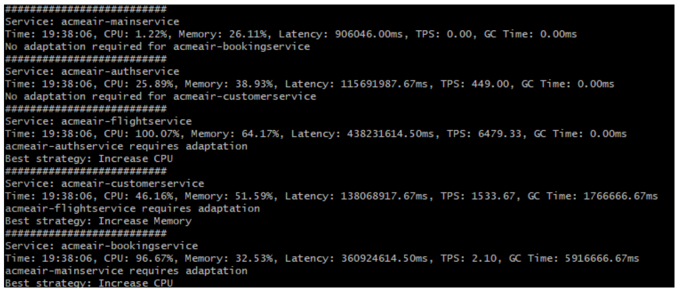
Fig. 5 and Fig. 6 demonstrate that prior to using our driver and without the use of JMeter, there was no need for container self-adaptation. However, when simulating a high workload, various metrics changed. By applying the strategies mentioned above, we evaluated each micro-service individually and provided tailored self-optimization for them.

In summary, these utility functions can be used as a logical basis upon which system optimization methods could be evaluated; hence, one could target effective means that would improve overall performance.



```
$ python driver.py
Pulling metrics from IBM Cloud
##########################
Service: acmeair-mainservice
Time: 19:36:05, CPU: 1.23%, Memory: 26.16%, Latency: 926943.83ms, TPS: 0.00, GC Time: 0.00ms
No adaptation required for acmeair-bookingservice
##########################
Service: acmeair-authservice
Time: 19:36:05, CPU: 6.30%, Memory: 38.98%, Latency: 5115386.83ms, TPS: 2.67, GC Time: 0.00ms
No adaptation required for acmeair-customerservice
##########################
Service: acmeair-flightservice
Time: 19:36:05, CPU: 21.49%, Memory: 63.71%, Latency: 8284250.83ms, TPS: 2.67, GC Time: 0.00ms
No adaptation required for acmeair-authservice
##########################
Service: acmeair-customerservice
Time: 19:36:05, CPU: 12.34%, Memory: 49.39%, Latency: 24557847.83ms, TPS: 2.67, GC Time: 0.00ms
No adaptation required for acmeair-flightservice
##########################
Service: acmeair-bookingservice
Time: 19:36:05, CPU: 74.33%, Memory: 40.95%, Latency: 23457835.33ms, TPS: 2.17, GC Time: 0.00ms
No adaptation required for acmeair-mainservice
```

Fig. 5: Results of the driver before adaptation.



```
##########################
Service: acmeair-mainservice
Time: 19:38:06, CPU: 1.22%, Memory: 26.11%, Latency: 906046.00ms, TPS: 0.00, GC Time: 0.00ms
No adaptation required for acmeair-bookingservice
##########################
Service: acmeair-authservice
Time: 19:38:06, CPU: 25.89%, Memory: 38.93%, Latency: 115691987.67ms, TPS: 449.00, GC Time: 0.00ms
No adaptation required for acmeair-customerservice
##########################
Service: acmeair-flightservice
Time: 19:38:06, CPU: 100.07%, Memory: 64.17%, Latency: 438231614.50ms, TPS: 6479.33, GC Time: 0.00ms
acmeair-authservice requires adaptation
Best strategy: Increase CPU
##########################
Service: acmeair-customerservice
Time: 19:38:06, CPU: 46.16%, Memory: 51.59%, Latency: 138068917.67ms, TPS: 1533.67, GC Time: 1766666.67ms
acmeair-flightservice requires adaptation
Best strategy: Increase Memory
##########################
Service: acmeair-bookingservice
Time: 19:38:06, CPU: 96.67%, Memory: 32.53%, Latency: 360924614.50ms, TPS: 2.10, GC Time: 5916666.67ms
acmeair-mainservice requires adaptation
Best strategy: Increase CPU
```

Fig. 6: Results of the driver after adaptation.

IV. Conclusion

Our analysis results show how effective our feedback loop is in keeping the system inline with the quality requirements, especially those concerning latency. Further, by handling fluctuating traffic and maintaining stable performance, we adapted resources in concert with real-time monitoring. It ensures the system is always kept responsive, given that it can perform self-optimizing of its configuration in respect to different loads-a high-traffic environment. The same methodology can then be extended further in the second half of the MAPE-K loop (Execution and Knowledge phases) in order to learn continuously and hence continuously adapt in such a way that in the longer run would result in better performance and reliability[1].

References

[1] M. Lee, "Optimizing Quality of Service in Cloud Environments," *Journal of Computer Networks and Communications*, vol. 2021, Article ID 7391503, 2021.

[2] R. Erdei and L. Toka, "Minimizing Resource Allocation for Cloud-Native Microservices," *Journal of Network and Systems Management*, vol. 31, no. 1, pp. 35, 2023. Available: https://doi.org/10.1007/s10922-023-09726-3