



VMware Tanzu Reference Architecture Performance Validation

Performance study for VMware Tanzu
Kubernetes Grid on VMware vSphere

Table of contents

Introduction	3
Test environment	3
Hardware	3
Software	4
Node configuration	4
Performance validation with K-Bench	4
Execution of control plane test cases	4
Observations from control plane test case results	8
Execution of data plane test case	9
Observations from data plane test case results	10
Performance validation with Locust	10
Observations from Locust load test results	12
Performance validation with Vegeta	13
Observations from Vegeta load test results	14
Performance validation of more than 10,000 concurrent requests per second on Tanzu Kubernetes Grid	15
Observations from Tanzu Kubernetes Grid load test results	15
Conclusion	17

Introduction

This VMware Tanzu® reference architecture performance validation focuses on benchmarking the cluster operations and applications on VMware Tanzu Kubernetes Grid™ on VMware vSphere®. It presents results and observations from executing the following benchmark frameworks and tools on the cluster:

- K-Bench
- Locust
- Vegeta

These benchmarks can be broadly categorized as follows:

- **Nodes benchmark** – The control plane and the workload plane nodes of a workload cluster are benchmarked in this category. K-Bench is used for this purpose. Performance metrics of the data plane and the control plane, such as transaction throughput, pod creation throughput, and CPU usage, are collected after running the respective test suites of K-Bench.
- **Kube suites benchmark** – Kubernetes benchmark suites that are adopted by the industry are executed on the applications running on the workload cluster. Locust and Vegeta are used in this performance validation to load test the applications. Performance metrics for the applications, such as requests per second (RPS) and response time for concurrent users, are collected in this study.

Test environment

Hardware

- 4 Dell PowerEdge R640 servers
- CPU: 245.84GHz (total), with the following per server:
 - Intel Xeon Gold 5120 CPU at 2.20GHz
 - Cores: 28 CPUs x 2.2GHz
 - Sockets: 2
 - Cores per socket: 14
 - Logical processors: 56
 - Hyperthreading: Active
- Memory: 1.53TB (total), with 382.62GB per server
- Storage: VMware vSAN™ (6.99TB total), with the following per server:
 - Cache: 1 x 380GB SSD
 - Capacity: 2 x 900GB SSD
- Network: 2 Intel X710 per server for 10GbE SFP+ - 10Gb/s

Software

- VMware vSphere 7.0 U3:
 - VMware vCenter® 19480866
 - VMware ESX® 19482537
 - vSphere High Availability (HA) and vSphere Distributed Resource Scheduler™ (DRS) enabled
- Tanzu Kubernetes Grid 2.1
- Tanzu Kubernetes releases 1.24.10
- VMware NSX® Advanced Load Balancer™ 22.1.3
- Avi Kubernetes Operator 1.9.3

Node configuration

Table 1: Workload cluster configuration – production plan

Plane	Number of nodes	CPU	Memory	Disk
Control	3	4	16GB	100GB
Worker	3	8	32GB	20GB

Table 2: Management cluster configuration – production plan

Plane	Number of nodes	CPU	Memory	Disk
Control	3	4	16GB	40GB
Worker	3	4	16GB	40GB

Performance validation with K-Bench

As part of this validation, the K-Bench test suite framework was executed on the clusters and collected the corresponding control plane and data plane metrics. Also, the metrics are integrated with Prometheus and Grafana for comparing the performance of nodes under different loads. The test suites include workflow of operations (such as Create, Update, List, Delete, Run and Copy) on different types of resources, including pod, deployment, service, replication controller, and so on.

Consider the following while executing the K-Bench test suites:

- Corresponding data and control plane metrics have been collected and mapped in the graphical representation.
- The observability dashboard containing CPU usage, memory usage, and networking usage is prepared and data is collected during the test case execution.

Execution of control plane test cases

Control plane test cases perform concurrent requests of Create, List, Get, Update and Delete on resource types, including pods, deployments, namespaces and services.

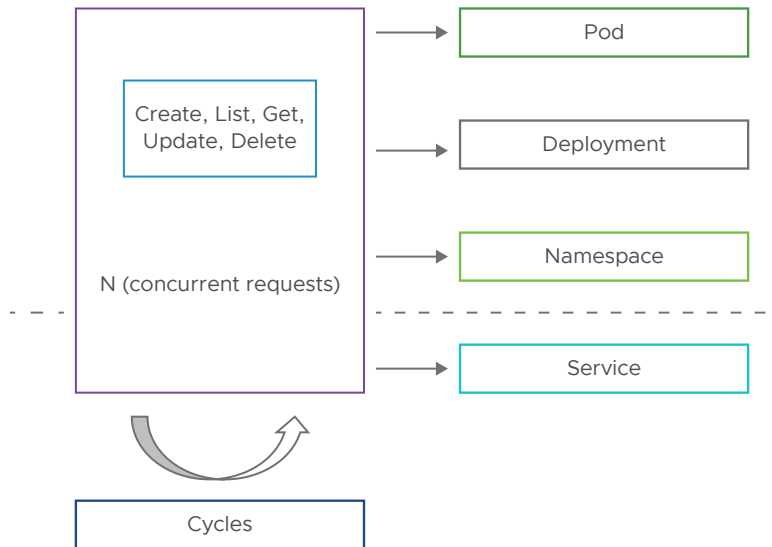


Figure 1: Test case execution.

- **cp_heavy_12client** – This test case executes 12 concurrent operations of Create, List, Get, Update and Delete on each resource type, including pod, deployment, namespace and service. This means in the first cycle, 12 pods, 12 deployments, 12 namespaces and 12 services creation, list, get, update and delete requests are sent. The process is performed twice, which is sequential during the complete test case execution, resulting in the execution of each operation (Create, List, Get, Update and Delete) for 24 resource types (pod, deployment, namespace and service).
- **cp_heavy_8client** – This test case executes eight concurrent operations of Create, List, Get, Update and Delete on each resource type, including pod, deployment, namespace and service. This means in the first cycle, eight pods, eight deployments, eight namespaces and eight services creation, list, get, update and delete requests are sent. The process is performed twice, which is sequential during the complete test case execution, resulting in the execution of each operation (Create, List, Get, Update and Delete) for 16 resource types (pod, deployment, namespace and service).
- **cp_light_1client** – This test case executes one concurrent operation of Create, List, Get, Update and Delete on each resource type, including pod, deployment, namespace and service. This means in the first cycle, one pod, one deployment, one namespace and one service creation, list, get, update and delete requests are sent. The process is performed twice, which is sequential during the complete test case execution, resulting in the execution of each operation (Create, List, Get, Update and Delete) for two resource types (pod, deployment, namespace and service).
- **cp_light_4client** – This test case executes four concurrent operations of Create, List, Get, Update and Delete on each resource type, including pod, deployment, namespace and service. This means in the first cycle, four pods, four deployments, four namespaces and four services creation, list, get, update and delete requests are sent. The process is performed twice, which is sequential during the complete test case execution, resulting in the execution of each operation (Create, List, Get, Update and Delete) for eight resource types (pod, deployment, namespace and service).
- **custom_suite** – This custom test case executes a different number of concurrent requests for resource types. It executes the Scale operation along with Create, List, Get, Update and Delete, and also executes a request to create three replicas for deployment.

Table 3: Test case results

Test case	Pod creation requests	Created pods	Throughput (pods/min)	Pod creation success ratio	Pod creation latency (ms)
cp_heavy_12client	24	24	148.77	100%	2.5
cp_heavy_8client	16	16	140.47	100%	1.79
custom_suite	4	4	53.99	100%	2.11
cp_light_1client	2	2	31.9	100%	1.88
cp_light_4client	8	8	78.64	100%	1.47

The followings tables show the benchmark results of test case cp_heavy_12client.

Table 4: Namespace API call latency for cp_heavy_12client

Operation	Median	Min	Max
List	4.839	2.747	1,446.2
Get	6,578.57	4.95	13,199.6
Update	8.97	4.96	35,342.18
Delete	11.52	6.79	17.09
Create	30.234	12.33	6,481.49

Table 5: Deployment API call latency for cp_heavy_12client

Operation	Median	Min	Max
List	27,522.98	19.12	27,528.15
Get	852.47	5.9	1,552.71
Update	14.54	9.7	20.93
Delete	9.44	4.75	25.72
Create	27,522.98	19.12	27,528.15

Table 6: Pod API call latency for cp_heavy_12client

Operation	Median	Min	Max
List	9.16	5.88	12.57
Get	13,326.96	7,332.69	13,852.02
Update	14.96	11.28	22.84
Delete	10.52	6.36	1,569.03
Create	27.6	19.64	34.29

Table 7: Service API call latency for cp_heavy_12client

Operation	Median	Min	Max
List	4.85	3.98	11.72
Get	3,055.81	4.3	3,227.36
Update	10.95	4.56	18.58
Delete	78.95	37.22	119.9
Create	3,238.41	16.79	3,417.67

Figure 2 shows the observability dashboard result of CPU, memory and networking usage per node during the time of execution of all control plane test cases.

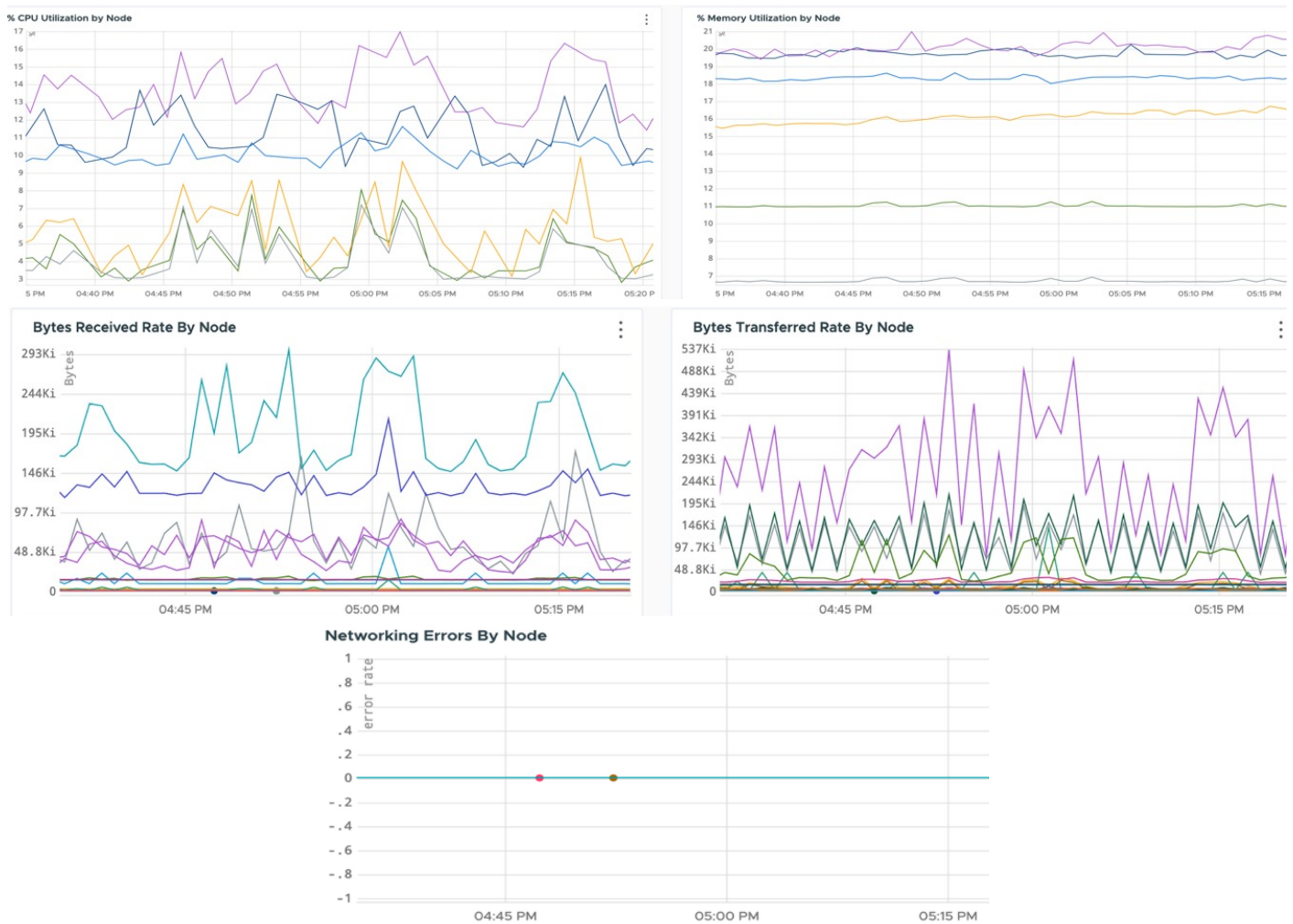


Figure 2: CPU, memory and networking usage results from test case execution.

Observations from control plane test case results

- When test cases are being executed, there is a 100 percent success rate of pod creation requests with no latency lag.
- CPU and memory loads are evenly distributed between the nodes.

Execution of data plane test case

The `dp_netperf_internode` test case captures the network performance measurement between client and server pods. It uses an NGINX image to create pods with the following resource parameters:

- `Memory_limit`: 4,200Mi
- `Cpu_limit`: 1,600m
- `Memory_requests`: 4,200Mi
- `Cpu_requests`: 100m
- `Ephemeral-storage_requests`: 5Gi

425,984 bytes of socket size is sent over an elapsed time of approximately 600 seconds. This resulted in no loss of the received socket size. The average throughput during the time is $31(10^6)$ bits/second).

The observability dashboard in Figure 3 shows the CPU utilization per node during the test case execution is throttled to 38 percent during the highest requests. The memory usage is also distributed over the nodes, and the maximum is 19 percent and no networking error received.



Figure 3: CPU, memory and networking usage per node during the test case execution.

Observations from data plane test case results

- From Figure 3 and the results of the `dp_netperf_internode` test case, it can be said that the netperf between the pods are completed successfully with 100 percent of packets received. All sockets sent are received successfully with no latency lag.
- During the maximum number of requests, CPU load is throttled to 38 percent and distributed among the data plane nodes. Not much deviation is observed in the memory load between the nodes, which is at a max of 19 percent. Tx/Rx bytes were at the maximum of 134Mbps with no errors or packet loss.

Performance validation with Locust

Locust is one of the tools that can be used for performing user behavior load tests. It relies on the popular programming language Python to define the load test scenarios. This means that it is possible to perform some conditional behavior or some calculations.

Locust also supports running distributed load tests over multiple data nodes. The tool comes with a web interface to configure and run the predefined tests with multiple configurations.

The following steps were done to load test the application in a distributed model:

- Define the test cases (Locust files).
- Deploy control and data Locust nodes.
- Allow communications between Locust control and data nodes.

For this validation, an NGINX server with a single pod as the web application was used to load test using Locust. The NGINX server is deployed on the workload cluster, exposed through a load balancer, and an external IP address is obtained. After that, concurrent Get requests of 15,000 clients are sent to the NGINX IP address for 120 seconds through Locust.

The purpose of this test is to check if the NGINX server with a single pod can handle 15,000 transactions per second and observe throughput, latencies and the resource consumption (CPU, memory and network) during the load test.

Figures 4 and 5 show the results of this test.



Figure 4: Load test results from Locust.

The benchmark result shows that Tanzu Kubernetes Grid can achieve a requests rate of 15,275 per second with no failure in response.

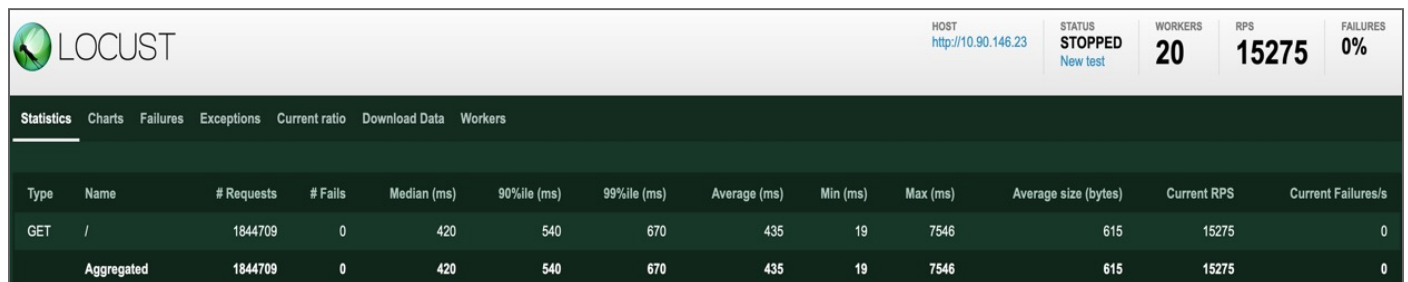


Figure 5: Requests and latencies for the test.

The total number of requests sent in 120 seconds is 1,844,709 with a median time per request of 420ms.



Figure 6: CPU, memory and network usage per node during the load test.

The maximum CPU usage is 34 percent, and the maximum memory usage is 13.5 percent during the execution. Tx/Rx bytes are in the range of 8Mbps and 5Mbps.

Observations from Locust load test results

- With a max CPU usage of 34 percent and a max memory usage of 13.5 percent, the NGINX server with a single pod on Tanzu Kubernetes Grid can handle 15,275 transactions per second with no failures. This shows that a large number of transactions can be handled easily with an increase in pods manually or with autoscaling because resource utilization is efficiently done between the nodes.
- From the observability graph in Figure 6, it can be seen that the CPU and memory loads are also distributed between the nodes during the load test execution and there is no loss in networking packets.

Performance validation with Vegeta

Vegeta is an open source application written in the Go programming language that is used to run load testing on an application.

For this validation, an NGINX server with a single pod as the web application was used to load test using Vegeta. The NGINX server is deployed on the workload cluster, exposed through a load balancer, and an external IP address is obtained. After that, concurrent Get requests of 5,000 clients are sent to the NGINX IP address for 120 seconds.

The purpose of this test is to check if the NGINX server with a single pod can handle 5,000 transactions per second and observe throughput, latencies and the resource consumption (CPU, memory and network) during the load test.

Table 8 shows the benchmark metrics and the corresponding results. From the table, it can be seen that a throughput of 4,999.96 is received with zero failure in response. Figure 7 shows the latency graph, where max latency is 22.13ms.

Table 8: Results of Vegeta performance validation

Metric	Result
Requests (total, rate, throughput) – The total number of requests sent during the test and the rate of the requests	60,00,00; 5,000.02; 4,999.96
Duration (total, attack, wait) – The total duration of the test, the attack period simulating load on the application, and the wait time	2m0s, 2m0s, 1.13ms
Latencies (min, mean, 50, 90, 95, 99, max) – The mean latency, 50th, 95th and 99th percentiles, respectively, of the latencies of all requests in an attack as well as the maximum latency recognized	339.25µs, 1.163ms, 995.56µs, 1.482ms, 2.09ms, 5.322ms, 22.13ms
Success (ratio) – The percentile of successful requests sent to the application	100.00 percent
Status codes (code:count) – A counter of the HTTP response codes received and their occurrence; a status code of 0 means that a request failed to be sent	200:60,00,00

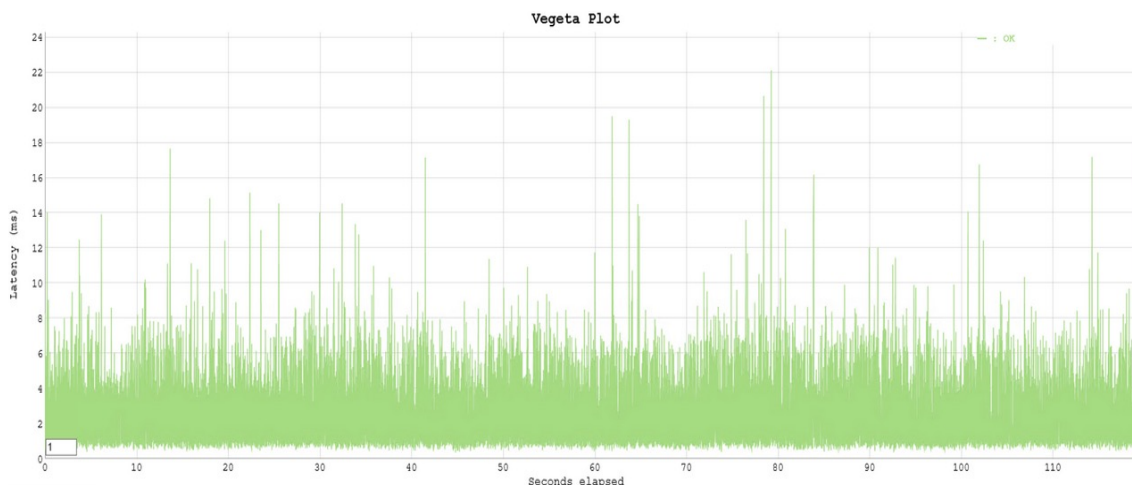


Figure 7: Latencies results during the load test.

The observability dashboard in Figure 8 shows the distribution of CPU, memory and network loads during the Vegeta run.

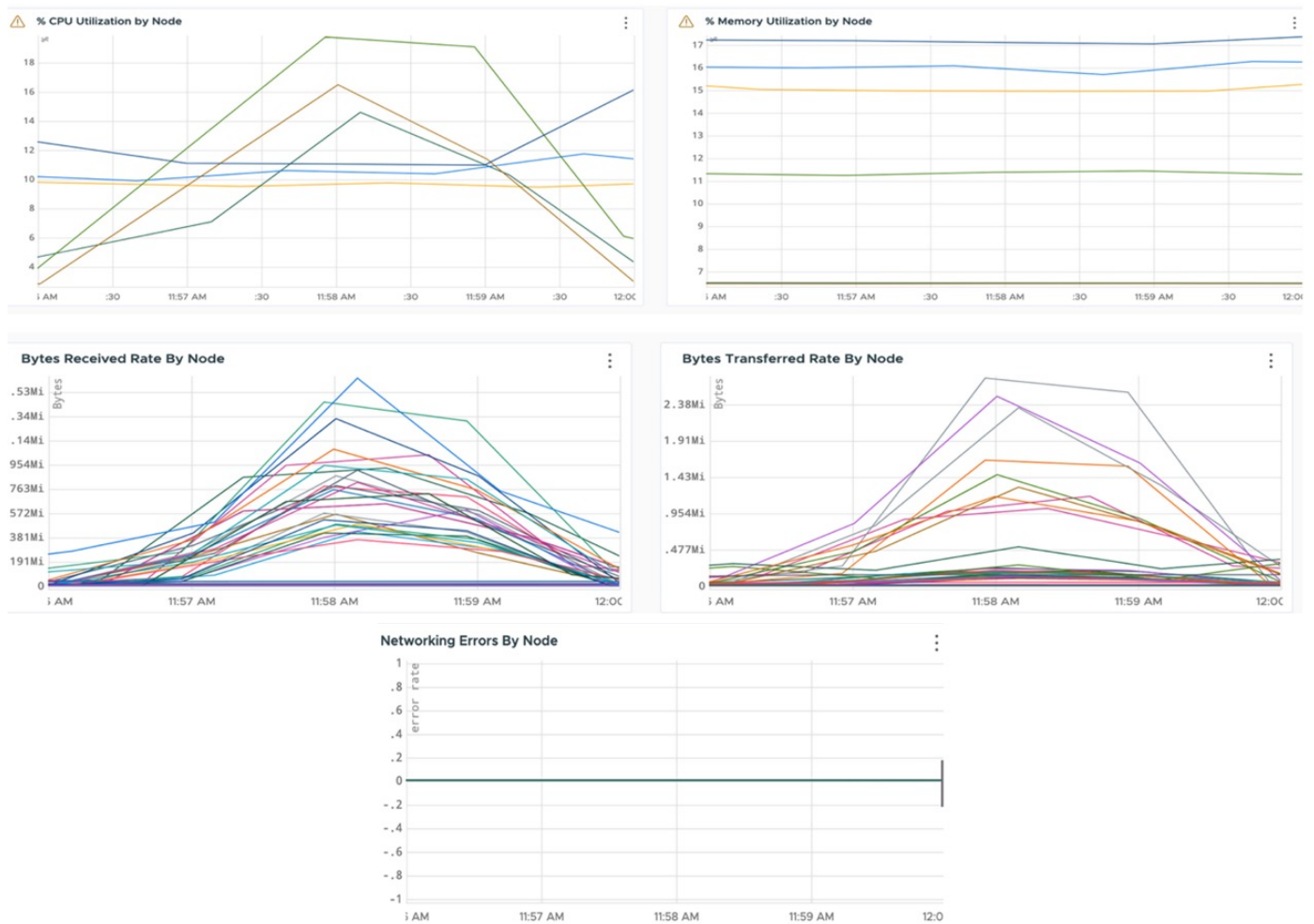


Figure 8: CPU, memory and network usage per node during load test.

Observations from Vegeta load test results

The NGINX web application with a single pod efficiently handles 5,000 transactions per second with a max CPU utilization of 19 percent and a max memory utilization of 17 percent. Tx/Rx bytes were observed in ranges of 2.38Mbps and 1.53Mbps, respectively, with no errors or packet loss.

Performance validation of more than 10,000 concurrent requests per second on Tanzu Kubernetes Grid

For this validation, a multitier shopping web application was tested. The application comprises six microservices and four datastores. For the test, more than 10,000 concurrent transactions/second were spawned to the application's endpoint on a Tanzu Kubernetes Grid workload cluster for a duration of 120 seconds and observed CPU, memory and networking loads.

Observations from Tanzu Kubernetes Grid load test results

Application transaction handling

A total of 134,8033 requests were sent during 120 seconds with an average latency of 367ms, as shown in Figure 9. The benchmark results show that Tanzu Kubernetes Grid can achieve a requests rate of 11,568 per second with no failure in response.

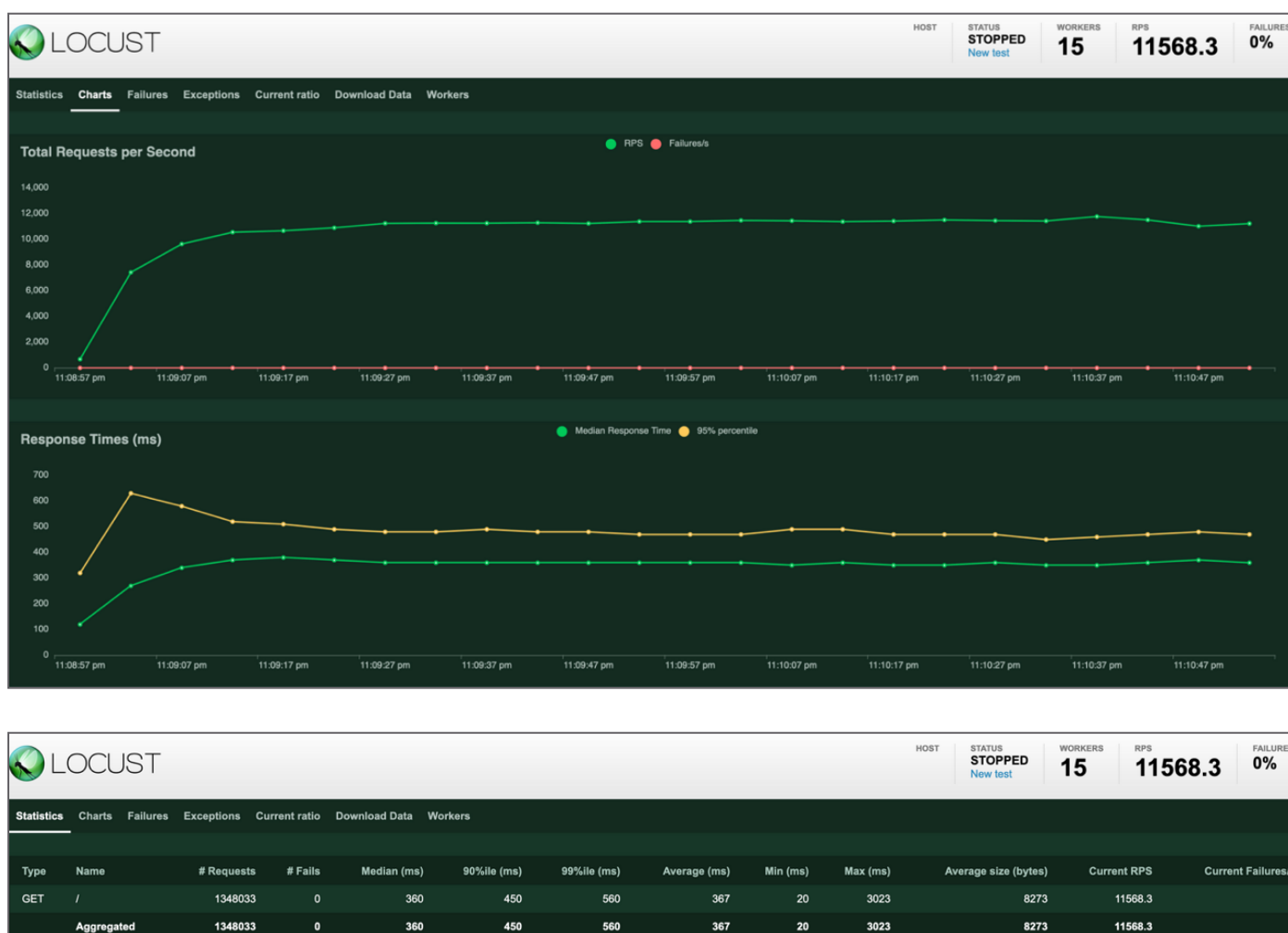


Figure 9: Benchmark results during more than 10,000 transactions.

Node and networking monitoring

- A maximum CPU usage of 36 percent and a maximum memory usage of 17.5 percent were observed during the execution of more than 10,000 transactions.
- CPU and memory resources were released, and no resource monopolizing was observed after the transaction benchmark.
- Tx/Rx bytes were observed in ranges of 48Mbps and 17Mbps, respectively, with no errors or packet loss.

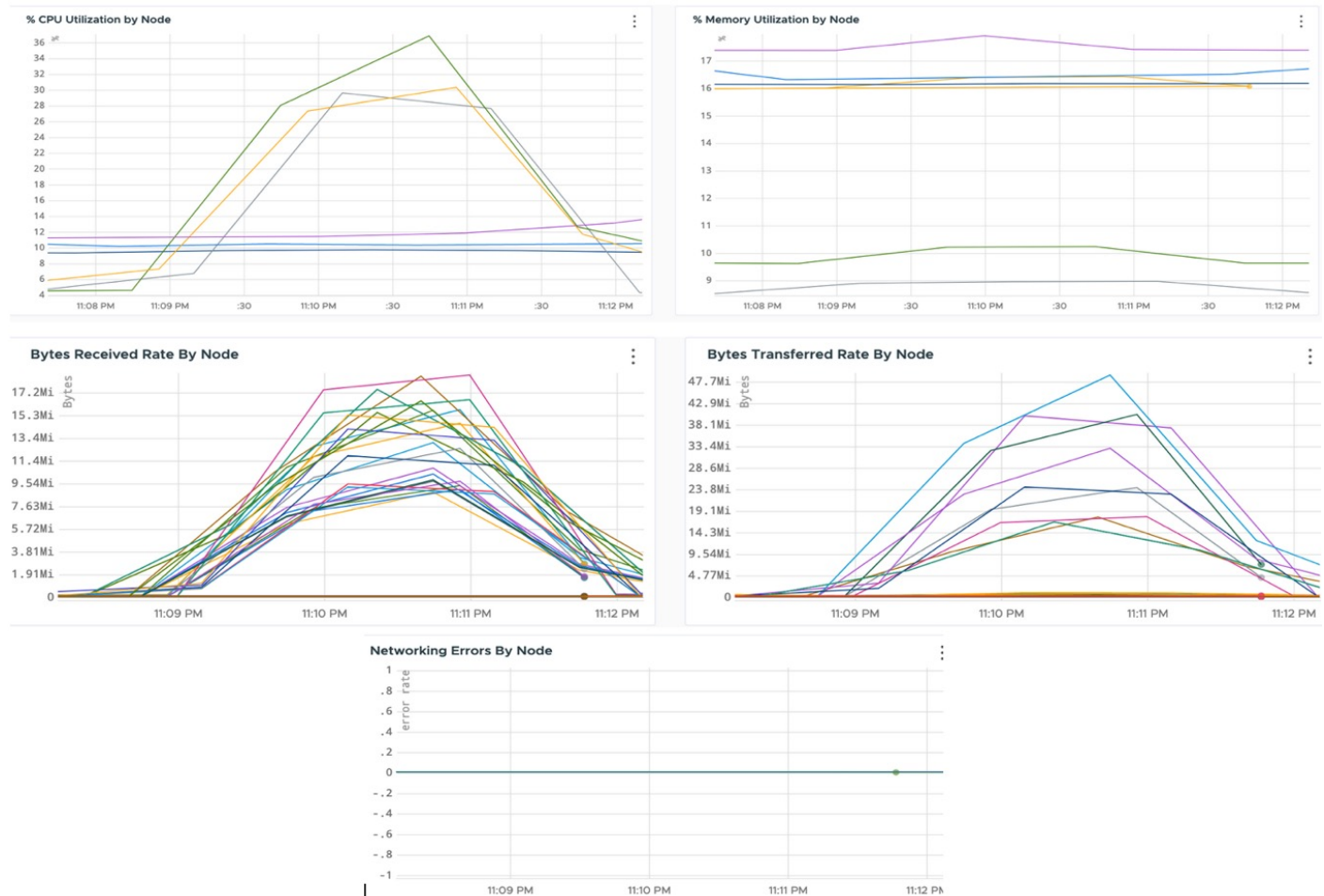


Figure 10: Observability graphs capturing CPU, memory and networking usage during more than 10,000 transactions.

Conclusion

- The results in this study show that the execution of multiple test suites of K-Bench where the control plane and data plane nodes of a workload cluster were benchmarked resulted in a 100 percent success rate of workflow of operations. Nodes can handle the concurrent requests with efficient distribution of CPU and memory loads between the nodes.
- When load tested with a different number of requests, a level of concurrency, a time duration to send the requests, and so on using Locust and Vegeta, the benchmark metrics of the application on the workload cluster showed good results and didn't bring down the application. The CPU and memory loads were distributed evenly during the execution time.
- Load testing the application's endpoint on a Tanzu Kubernetes Grid workload cluster with more than 10,000 transactions per second did not result in any response failure. Also, the CPU and memory loads were distributed evenly between the nodes during the test execution. CPU load gradually decreases as the execution completes, and no networking errors or packet drops were observed.

