Impact Analysis Of
Cloud Based Microservices Application Using Chaos Engineering

Sonu Kumar

School Of Computing

Newcastle University

Supervisor

Prof. Rajiv Ranjan (Newcastle University)

In *partial* fulfillment of the requirements for the degree of

MSc in Cloud Computing

CSC8199

August 23, 2022

DECLARATION I, Sonu Kumar, hereby declare that this thesis entitled Impact Analysis Of Cloud Based Microservice Application Using Chaos Engineering is a bonafide record of research work done by me for the award of MSc in Cloud Computing from Newcastle University, UK. It was never submitted to any university or entity before, in part or in full, for any degree, diploma, or other qualification.

Signature [illegible]

Abstract

Microservices-based applications can be written and deployed in a cloud context considerably more quickly, thanks to developing container technologies like Docker and container orchestration technology like Kubernetes. For application providers, the reliability of these microservices becomes a top priority. Abnormal actions must be detected, and robust behaviour must be assessed in advance in case unexpected failures do occur. Fault injection is a specialized method to test certain situations, while chaos engineering is an experimental way of learning new information.

In this study, we attempt to define a method to find which layer of infrastructure is impacted by these attacks, based on different types of fault injection using chaos engineering practices. At first, test infrastructure has been designed along with an example microservices application. Two services are running in this application. Later, the steady state of the deployment is defined by running a small set of threads and observing the application performance. The threshold user load in corresponds to the deployment is set before doing the actual fault injection. Consequently, different types of experiments were performed on the designed test infrastructure. Collected the experimental metrics and analyzed them by using *K-means* and outliers detection activity performed to evaluate the most impacted areas. This would help to debug and fix the problem quickly as compared to inspecting all the infrastructure. The impact of this will be in day-to-day activities that the turnaround time is reduced if any failure is reported.

Keywords: Chaos Engineering · Microservices · Performance monitoring · Machine learning. Fault Injection.

1 CONTENTS

2 List Of Figures

3 List Of Tables

CHAPTER 1

INTRODUCTION

The foundation of trust for the users is software quality. All the organizational engineering teams strive to guarantee that web applications are available and effectively satisfy the demands of end users. Maintaining a high level of availability is critical for guaranteeing client satisfaction and confidence. Any website failure, receives persistent criticism from consumers, leading to a decline in usage[1].

To develop the high availability software, a new development methodology got popularity. It is known as *Microservices*. The emergence of microservice architecture has had a considerable impact on web application development, deployment, and ongoing maintenance. The microservice approach decomposes the application into several independently executable software components or units that coherently interoperate to deliver specific application functionality, as opposed to the normal monolithic application architecture, which builds the entire application as a single unified system[2]. However, the microservice architecture has its complexity to deploy on cloud platforms and that makes it vulnerable to faults. For the team of engineers, it is a challenging thing to make sure the availability and ability to handle a high load of requests once deployed in a production environment. So, there must be some mechanism to test the overall application resiliency before deploying to a production environment. And here Chaos Engineering came into the picture which provides a simulation to check the resiliency of the developed application against the potential simulated loads and failures.

According to Gremlin, “Chaos Engineering is a disciplined approach to identifying failures before they become outages.” The introduction of microservice-based design, as well as the necessity to handle many requests without fail, is critical to any organization. Chaos Engineering allows us to intentionally damage our system to make it more robust and less susceptible to failure. Chaos Engineering is concerned with designing systematic and well-defined experiments to depict the behavior of our systems in turbulent periods. It begins with the analysis of the overall system and the sub-components that can be focused on [1].

Operating at high performance has challenges, just like with any other complex application. In this context, system maintenance and debugging of the cloud-based microservice in case of some failure is mostly connected to the operational challenges faced frequently. These issues might include high CPU utilization, pod failure, network outrage, etc, and sometimes even extraordinarily high operating loads. These maintenance intervals are often labor and time intensive. Furthermore, there is a potential that frequent infrastructure outages may occur during maintenance. Since microservices run individually and cover several layers, debugging and getting a category of error is a daunting task.

The same is planned for an experiment, and a hypothesis as to what might go wrong is stated. Every test has an effect and knowing when those impacts are limited is crucial. This is underlined by defining the "blast radius"[5]. For the initial round of trials, the blast radius is the lowest and increases significantly with each subsequent round. The procedure is repeated until a problem is identified or the greatest radius is achieved. Based on these fault injections, collected metrics can be analyzed to find the most impacted layer. This will help to correct the shortcomings of that layer’s configuration. These actions will lead to an increased application's resiliency and make it less prone to error when deployed in a production environment. The experiments aid with comprehension of the product's potential problems before taking action to address them and raise their quality.

MOTIVATION

As earlier mentioned, cloud-based microservices function in a multitude of encapsulated layers, making it a time-consuming and tedious job to troubleshoot faults or improve performance. Cloud-based Microservices are implemented in distributed clusters with varying degrees of virtualization to manage resources including operating systems, hardware, applications, and communications, among many others.

All of the above factors contribute to the acceptance and implementation of cloud-based microservice architecture, but when something goes wrong due to poor design or if a layer doesn't operate as expected, the situation could become a nightmare for administrators and developers. To prevent run-time errors and their recurrences, a mechanism that can highlight the potential segment of fault during the test is desired.

There are numerous monitoring tools available, but they're often expensive or resource intensive. Also, many Anomaly detection systems are present however most of them monitor and analyze the metrics collected at the Virtual machine or Container level. Sometimes due to poor configuration of infrastructure, an application may behave differently. For example, in a real-life production scenario, generic errors are reported that do not pinpoint the actual cause of the error and this will lead to checking every aspect of the infrastructure which relates to the generic error. The existing monitoring and fault detection system may also point out that the issue is related to *CPU, memory, bandwidth, latency,* etc. However, these parameters exist at all the layers in cloud-based microservices. Many Nodes constitute each other under one master and form a cluster, it would have its way to handle, and said parameter will vary. The application deploys as containers and runs within a pod and these entities communicate with each other with their network, using a direct pod *IP address* or using *overlay* network architecture, it depends on the type of network plugin used to create a cluster. Similarly, being a scalable system, if loads go up, the cluster scales up or scales out to cope with the load, this results in either adding more nodes to the cluster or adding more capacity VMs. These nodes have their way to share resources. Hence existing monitoring or fault detection systems do not highlight which layer has the issue, it leads to a review of all the layers while troubleshooting the error or fault reported in production. Since the production environment hosts the application, which is used by actual users, longer downtime to resolve it may lead to big user dissatisfaction and ultimately a loss to the organization.

This study tries to define the method through which problematic layers can be highlighted so that the troubleshooting can be done directly to that layer and the turnaround time would be reduced.

The goal of this study is to define a method that can determine the potential origin of a fault while introducing chaos engineering faults under the steady state of the application load. Chaos engineering is a process to test the resiliency of the whole system against disruptions. Using this, real-world faults would be injected at the run time of the entire infrastructure, and based-on metrics collected, manual analysis is performed to find which layer is most impacted. The study is based on metrics collected at different layers of cloud-based microservices, metrics such as CPU usage, Memory Usage, and HTTP response time of APIs at different layers. The said Key Performance Indicator (KPIs) will be collected for the cluster, pod, and container level, and later manual analysis done by using a machine learning algorithm to find the outliers in different datasets. This would be helpful to identify the fault quickly. It will also help to know the pointed area of impact so that it is not required to debug the entire infrastructure.

RESEARCH AIM AND SCIENTIFIC OBJECTIVES

Since cloud-based microservices are distributed in nature and hosted on different nodes within a cluster and it has different layers to make them run with proper communication with each other. The structure is having different layers like clusters, nodes, pods, containers, etc. When an API is hit by a user on the client side, the request passes through the load balancer, and reaches a routed node, on that node, a certain pod is configured to serve the request and finally reaches the destined container to get the response. Consequently, the container, initiate the response based on the request and the same traverse through all these different layers to reach the client side. So, the exchange of requests and responses in the microservice application needs to pass through different infrastructures and different types of validation and security check done at these layers. It can be imagined, that if any one of these layers finds the request invalid and terminate all these steps there, it would be difficult to find the reason immediately however the monitoring tool will only tell something went wrong in the infrastructure, and then the classical way of investigation would be initiated. To debug these individual layers are difficult and need a higher turnaround time to identify the faulty layer. Hence this experimental study explores the methodology to find the impacted layer quickly.

Below given are the aims to achieve in this study.

1. Methodology to find the most impacted layer in the cloud-based microservice application.
2. Define different types of chaos experiments and validate the outcome against the hypothesis formulated for the corresponding fault injection.

To achieve the previously mentioned aims, the scientific objectives are mentioned below.

1. Design a Public cloud-based cluster configured with several nodes. This will help to host microservices on this and later chaos experiments can be executed on it.

2. Define the microservice application manifest file to make it ready to define the application's steady state and later perform fault injection experiments.

3. Design the fault injection hypothesis and metrics collection system to critically analyze the impact of experiments and validate against the hypothesis designed. This objective will realize our second goal of study.

4. Finally analyze collected metrics with different parameters using the *k-means* clustering machine learning algorithm to accomplish our primary aim of the study. In addition, estimates and conclusions were drawn based on the outcomes of these studies to accomplish the overall purpose, aiding in the critical assessment of this experimental study.

Overview Of Dissertation

The scope of the research primarily focuses on the fault injection on the Azure Kubernetes cluster using the Chaos mesh tool and collecting the respective performance metrics for the resultant analysis to find the most impacted layer in infrastructure.

The primary emphasis of the research is on fault injection utilizing the Chaos mesh tool on an Azure Kubernetes cluster, integrated with information gathering for the consequent analysis. The background information, related work, research objective, and motivation are described in *Chapter 1*. *Chapter 2* gives a glimpse of the ongoing efforts to develop a comprehensive system and integrate it with different monitoring and user thread load simulating tools that aid in locating issue areas when the fault injection activities are performed and, on top of that, serves as the foundation for this study. Additionally, it details the process used to set up the Azure Kubernetes cluster, integrate Chaos-mesh for fault injection, and then combine Prometheus and Grafana to gather metrics. There is also a description of the procedures used for efficient configuration and integration as well as the parameters that were examined.

The methods used to conduct the experimental investigation are described in *Chapter 3*. In this chapter, the steady state of the application is evaluated before the fault injection experiment. The steady state is important to know the overall system capacity before fault injection experiments. Later, different types of fault injections were performed by following the blast radius strategy. The strategy significantly avoids the whole system to reach an unrecoverable state. Furthermore, the experiments, application deployment efforts, and associated metrics gathering steps performed will aid in further data analysis. It also offers an in-depth summary of the study's analytical techniques used.

Chapter 4 focuses on the results obtained from the experiments. It also describes the observations of the results. Data analysis using a machine learning clustering algorithm was undertaken as part of the research, along with the critical evaluation and limits of the research work. Consequently, it also gives an insight into the impact this research will have on real-world scenarios. Conclusively, *Chapter 5* provides the overall conclusion with a closing statement of our experiment along with the scope of future enhancements.

Background

The complexity of a typical monolithic application is moved into the infrastructure by using the cloud application design pattern known as microservice architecture. Microservices-based architecture, as opposed to monolithic systems, builds systems out of several smaller services, and each one is standalone, scalable, and failure-resistant. Language-neutral application programming interfaces are used by services to interact via networks (API). Microservice uses different kinds of virtualization to make space for services separately so that they can execute independently. Many types of virtualizations can separate different layers of the OSI model. Like, OS, Network, CPU, etc.

Containers are lightweight OS-level virtualizations that allow us to run an application and its dependencies in a resource-isolated process. Each component runs in an isolated environment and does not share memory, CPU, or the disk of the host operating system(OS)[10].

Since modern systems are built using a high number of microservices, therefore, to efficiently create, manage, and run such applications, quick provisioning, rapid deployment via decentralized continuous delivery, stringent DevOps methods, and holistic service monitoring are required. An entirely new discipline of infrastructure platform and tool development was opened up by the infrastructure demands imposed by microservices architectures for managing these sophisticated cloud-native apps[6].

Individual running services make microservice more resilient and highly available however these services still need to communicate with each other to make a complete application service for the end user. Developing lots of individual services makes the development process quick and easy to deploy however the maintainability, observability along with debugging in case of run-time failure is challenging.

Infrastructure is increasingly significantly more challenging as serverless models and microservices gain popularity. The inner workings of each module, serverless call, and cloud component are further hidden. Because of this, observability and tracing are troublesome since developers frequently cannot determine the internal state of a microservice system simply on outputs. Microservices are hard to comprehend user requests between asynchronous modules and recreate faults since they operate independently. Furthermore, it might be challenging to determine which services interact because each request may follow a different path. These characteristics cumulatively make it extremely difficult for developers and administrators to determine an error's causative factors[8].

To test distributed software's tolerance to arbitrary disturbances, a mechanism called *chaos engineering* was used, which intentionally produces failure and flawed situations. It is a methodical strategy of spotting errors before they lead to disruptions. A system's resilience to stress can be tested proactively to find problems before they become public and to rectify them. It begins by formulating an assumption on how a system ought to operate in the event of a problem[9]. Next, create the smallest experiment imaginable to test it on our system. At each stage, assess the impact of the failure and seek indications of success or failure. After the test, it will be clearer how the system behaves in actual use.

After the failure injection, we need to analyze the collected metrics to find out meaningful information so that the vulnerable aspect of the system would be rectified. A new paradigm to data analysis for fault injection experiments, which we call fault injection analytics. The approach combines distributed tracing to gather raw failure data, and unsupervised machine learning to discover the failure modes of the injected system[11].

RELATED WORK

Performance monitoring and performance evaluation have been a popular issue for container researchers due to the widespread usage of microservice architecture and container technology.

With performance data from the CPU and network, the authors assessed the performance of container-based microservices in two alternative models[12].

The authors of the paper described their technique, and how to do anomaly detection for cloud services [13]. They set up a cloud application that operated on many virtual machines and was made up of various services. Each VM's performance data was gathered and afterward analyzed to look for any abnormalities using machine learning methods.

To monitor and analyze real-time performance data of microservices and find and diagnose anomalies in them, an *anomaly detection system (ADS)*was developed[17]. An integrated fault injection module, a data processing module that employs machine learning models, and a monitoring module that collects container performance data make up the proposed ADS. The fault injection module is also used to assess the effectiveness of our ADS' diagnostic and anomaly detection capabilities.

In [27], the author proposed KubAnomly, a system that provides security monitoring capabilities for anomaly detection on the Kubernetes orchestration platform. It mainly focuses on overcoming the shortcoming of the containerization of the applications. Since a single server can host multiple services and share the same resources however it comes with challenges in terms of complete monitoring and security provisioning. The hackers can exploit the vulnerabilities of the container to gain control and it may lead to huge losses for any organization. Therefore, the author proposes a monitoring system that monitors especially at the container level. Later, neural network approaches were implemented to find abnormal patterns. Here the main focus at the container level is security monitoring and analysis for abnormal activities.

The authors of [28], proposed an anomaly detection system that predicts abnormal behavior by adapting the system's normal characteristics. The author used the performance log and monitoring KPIs to build a predictive model. In this paper, there would be a predictive alert for any abnormal. It uses an adaptive model by adjusting itself against the new threshold values. PCA algorithm is used to transform the performance metrics into the low-rank matrix. It focused mostly on performance data of the infrastructure however still no details of granular level KPIs and it alerts based on overall performance metrics.

Although all the linked research produced some sort of analytical framework to foresee the error or the aberrant behavior in real-time. Few monitor the parameters related to containers; some primarily monitor at the virtual machine level. However, the categorical failure of the complete infrastructure is not delivered by these strategies which could limit the debugging effort and may fail to pinpoint the potential area of the fault.

This study focused on finding the potential area of fault in the entire cloud cluster while injecting the fault into deployed microservices. The proposed methodology is divided into three units, Load & Fault Generator, Monitoring& Processing, and Analytical Unit. Load & Fault injection unit performs to generate user thread along with fault injection to the example microservices application to the entire cluster using key metrics as *CPU* and *Memory* usage along with *APIs response* time. In the end Analytical unit uses *the K-Means* clustering algorithm to analyze and find the outliers once the application runs in a steady state and different types of faults are injected. It categorizes the metrics collected into three categories namely, *Cluster, Node, and Pod*. Later based on the comparative study of change in metrics at a different layer, a conclusive outcome was drawn.

CHAPTER 2

TEST INFRASTRUCTURE SET UP

In this chapter, detailed architecture and its explanation are given which has been used in this study. Below given figure shows the overall system design along with individual components. Azure Kubernetes cluster with 2 nodes has been used to form a cloud-based formation. Later, Azure Voting Application, an example microservice application deployed. Since this study focused more on fault injection and analysis of its resultant data. The application has two services, frontend, and backend. The front is used to service users to select the specific voting and the backend service uses Redis, an in-memory key-value database to store the response of the user. Further, Prometheus and Grafana are deployed on the same cluster to capture the metrics. These metrics are then exported as CSV files and downloaded to a local machine for further analysis.

The following sections provide detailed information related to the overall methodology for the entire study carried out. It covers, the method to create an Azure Kubernetes cluster design and implementation along with the method to deploy an example application. The load generator unit has two components namely, User thread load(JMeter) and Chaos-mesh Fault injector.

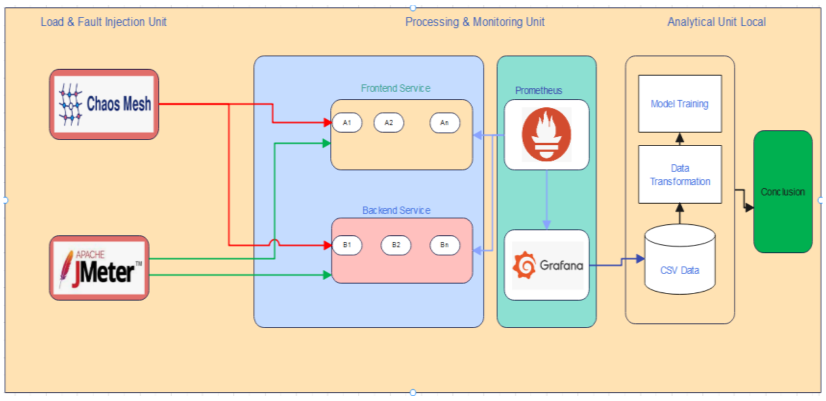


Figure 1: System Architecture

The infrastructure setup is done in three stages as in each stage and unit is deployed/configured.

STAGE 1:PROCESSING UNIT DEPLOYMENT

AZURE KUBERNETES CLUSTER DESIGN AND DEPLOYMENT

In this stage, the Kubernetes cluster is designed using an Azure managed service called Azure Kubernetes Server. In this setup, the Kubernetes master is managed by Azure. Since our experiments are of low capacity so, fewer captive VMs spawned to create a cluster. Load balancer service is also enabled to handle and evenly distribute the user thread request. The setup used with auto-scale enabled two Nodes of the *B2ms series* of Virtual machines which have *2 vCPUs*, *8GiB* RAM along with *16 GB* of temporary storage. Authentication and authorization have been enabled and it uses the Role-based access control (RBAC) service of Azure. The network configuration for this cluster is managed using *Kubernetes*.

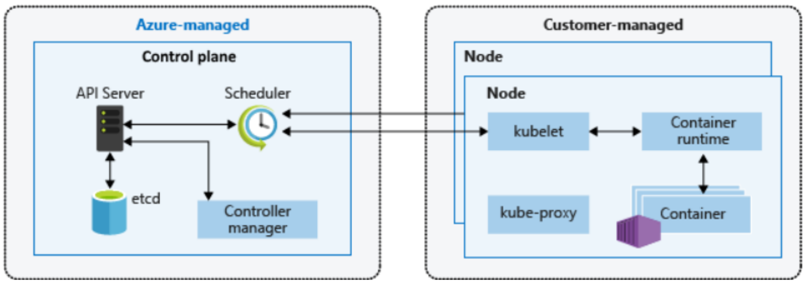


Figure 2:AKS Cluster Architecture [18].

The above diagram shows the generic high-level Kubernetes server management. Only the control plane is managed by Azure however the node scaling and number pods, containers, and other manifests are configurable at the data plane according to the requirement of experiments.

Below given is the Azure Kubernetes Cluster design used in this experiment. It has 0ne active node and scales out to one more node. The picture is showing the normal state of the cluster hence showing node scale set, auto-scale up to 2 Node maximum.

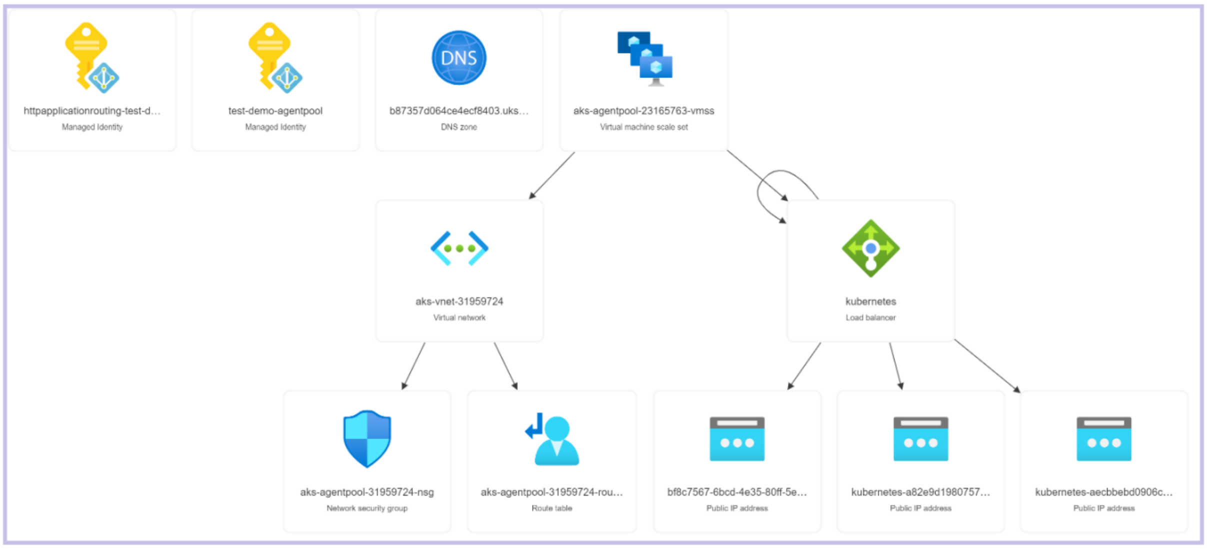

httpapplicationrouting-test-d...
Managed Identity
test-demo-agentpool
Managed Identity
DNS
b87357d064ce4ecf8403.uks...
DNS zone
aks-agentpool-23165763-vmss
Virtual machine scale set
aks-vnet-31959724
Virtual network
kubernetes
Load balancer
aks-agentpool-31959724-nsg
Network security group
aks-agentpool-31959724-rou...
Route table
bf8c7567-6bcd-4e35-80ff-5e...
Public IP address
kubernetes-a82e9d1980757...
Public IP address
kubernetes-aecbbebd0906c...
Public IP address

Figure 3: Azure Kubernetes Cluster

The picture shows that the cluster is having a different set of services running to manage the whole cluster, services like DNS Zone, Network Security Group, Routing table,vNet, Managed Identity, and load balancers. Here the deployment used network type *kubenet*, in this network plugin, *pods* receive the IP address from different logical address spaces and are not accessible outside the cluster. In other words, these addresses are not public IP addresses.

Network Address Translation is then configured at the node level and the routing table is maintained for inter-pod communication.

PROMETHEUS AND GRAFANA INTEGRATION [31][38].

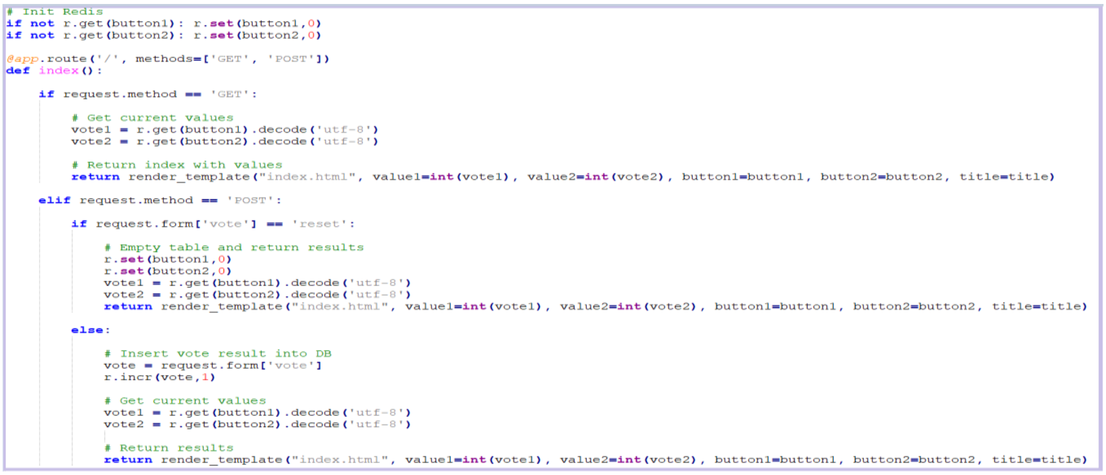
Once the cluster setup is done, we proceed with the monitoring setup. In our experiments, we have utilized the open-source monitoring tool *Prometheus*[38]. It collects the various metrics of the entire cluster. We have captured the same metrics at a separate level. It captures *CPU, Memory, and Bandwidth(Transmit and Receive)*at all three levels Cluster, Node, and Pod for the application Containers. These metrics would be utilized further for finding the unexpected spike at a certain level to find and ease the debugging. Different Grafana dashboards [31] were used for better visualization and CSV export.   To integrate the Prometheus and Grafana, a Helm chart had been used and command followed . Prometheus and Grafana come with a single helm chart which consists of Kubernetes manifestation to get deployed, and it creates respective pods. Prometheus pods deploy their demons to respective nodes and scraps, Later, Grafana had been used to export the metrics from specific dashboards as *CSV* files to local machines for further analysis. The structure of categorical data is shown in the table given below.

Level	Metrics Name
Cluster	CPU Usage
	Memory Usage
	HTTP Response
Nodes	CPU Usage
	Memory Usage
	HTTP Response
Pods	CPU Usage
	Memory Usage
	HTTP Response

Table 1: Metrics Used for Analysis

AZURE VOTING APPLICATION DEPLOYMENT

This stage starts with deploying the example application which is taken from the Microsoft site. This application uses two services to serve the user's casted vote. The application uses a button to save the vote between Dog and Cat. It uses form-based data capture and is stored in an in-memory database called *Redis*. It has 3 buttons enabled for the user. For the deployment of these, Kubernetes manifestation is created where two services Frontend and backend are created along with a Load balancer to access it over a network.



```python
# Init Redis
if not r.get(button1): r.set(button1,0)
if not r.get(button2): r.set(button2,0)

@app.route('/', methods=['GET', 'POST'])
def index():

    if request.method == 'GET':

        # Get current values
        vote1 = r.get(button1).decode('utf-8')
        vote2 = r.get(button2).decode('utf-8')

        # Return index with values
        return render_template("index.html", value1=int(vote1), value2=int(vote2), button1=button1, button2=button2, title=title)

    elif request.method == 'POST':

        if request.form['vote'] == 'reset':

            # Empty table and return results
            r.set(button1,0)
            r.set(button2,0)
            vote1 = r.get(button1).decode('utf-8')
            vote2 = r.get(button2).decode('utf-8')
            return render_template("index.html", value1=int(vote1), value2=int(vote2), button1=button1, button2=button2, title=title)

        else:

            # Insert vote result into DB
            vote = request.form['vote']
            r.incr(vote,1)

            # Get current values
            vote1 = r.get(button1).decode('utf-8')
            vote2 = r.get(button2).decode('utf-8')

            # Return results
            return render_template("index.html", value1=int(vote1), value2=int(vote2), button1=button1, button2=button2, title=title)
```

Figure 4: Application Logic [29].

The above diagram shows the core logic of the application where it has two endpoints open to *get* the status of voting and *post* the new vote. The value is set using two buttons for individual vote options. At the end it renders the index template to show the late status.

STAGE 2:LOAD & FAULT INJECTION UNIT

FAULT INJECTION MODULE

A cloud-native framework for chaos engineering called Chaos Mesh conducts chaos experiments in Kubernetes ecosystems. It helps to imitate conditions like network failures, file system failures, and Pod faults to check the system's resilience. Built on the Custom Resource Definition (*CRD)* for Kubernetes, Chaos Mesh. Chaos Mesh creates many CRD types based on various fault kinds and implements distinct Controllers for various CRD objects to manage various Chaos experiments. The three main parts of Chaos Mesh are as follows:

Chaos Dashboard: Chaos Mesh's visualization feature. We can control and see Chaos experiments using the online interfaces provided by Chaos Dashboard. Chaos Dashboard also offers an RBAC permission management system at the same time [19].

Chaos Controller Manager: It is a primary logical element of the Chaos Mesh. The scheduling and administration of Chaos experiments fall largely within the purview of the Chaos Controller Manager. Several CRD Controllers, including Workflow Controllers, Scheduler Controllers, and Controllers of various fault kinds, are present in this component [19].

Chaos Daemon: Chaos Daemon typically has Privileged permission and operates in the *DaemonSet* mode (which can be disabled). By breaking into the target Pod Namespace, this component primarily interferes with certain network devices, file systems, and kernels[19].

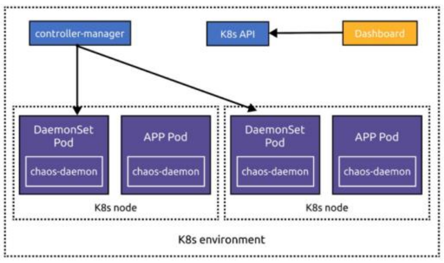
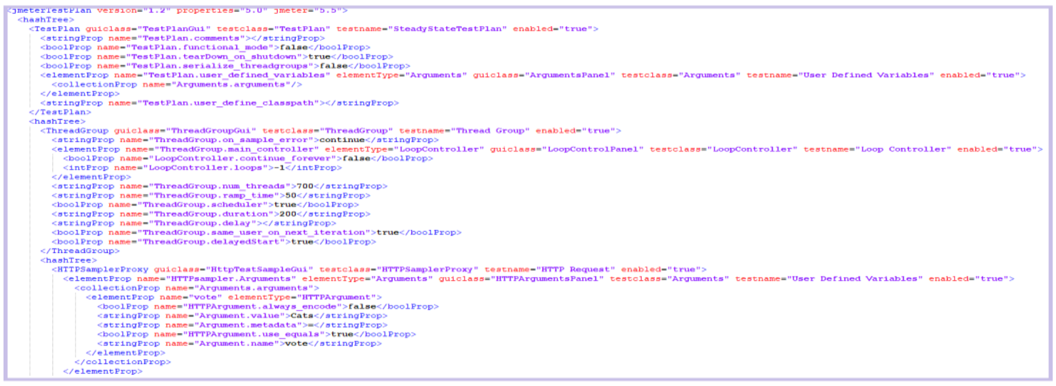


Figure 5: Chaos-Mesh Architecture [20]

Figure 5 shows the daemon set accepts commands from the controller manager and hacks into the containers cgroup, namespace, IP sets, etc. It injects according to the fault category injected as input by the user and command received from the controller manager. Installation and integration are given in Appendix A document folder which is the reference from Chaos-Mesh website [41].

Load generator Module

To simulate the user load to the deployed microservices, Apache JMeter is the choice in this experimental study. This is an open-source application developed in Java to load test the functional behavior and measure performance. It has been used tool to simulate heavy user load on the deployed microservices using the set of the HTTP request. In this module, the constant load is generated using the pods in question. The below-given diagram shows one of the examples *HTTPjmx* file setting.



```
<jmeterTestPlan version="1.2" properties="5.0" jmeter="5.5">
  <hashTree>
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="SteadyStateTestPlan" enabled="true">
      <stringProp name="TestPlan.comments"></stringProp>
      <boolProp name="TestPlan.functional_mode">false</boolProp>
      <boolProp name="TestPlan.tearDown_on_shutdown">true</boolProp>
      <boolProp name="TestPlan.serialize_threadgroups">false</boolProp>
      <elementProp name="TestPlan.user_defined_variables" elementType="Arguments" guiclass="ArgumentsPanel" testclass="Arguments" testname="User Defined Variables" enabled="true">
        <collectionProp name="Arguments.arguments"/>
      </elementProp>
      <stringProp name="TestPlan.user_define_classpath"></stringProp>
    </TestPlan>
    <hashTree>
      <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup" testname="Thread Group" enabled="true">
        <stringProp name="ThreadGroup.on_sample_error">continue</stringProp>
        <elementProp name="ThreadGroup.main_controller" elementType="LoopController" guiclass="LoopControlPanel" testclass="LoopController" testname="Loop Controller" enabled="true">
          <boolProp name="LoopController.continue_forever">false</boolProp>
          <intProp name="LoopController.loops">-1</intProp>
        </elementProp>
        <stringProp name="ThreadGroup.num_threads">700</stringProp>
        <stringProp name="ThreadGroup.ramp_time">50</stringProp>
        <boolProp name="ThreadGroup.scheduler">true</boolProp>
        <stringProp name="ThreadGroup.duration">200</stringProp>
        <stringProp name="ThreadGroup.delay"></stringProp>
        <boolProp name="ThreadGroup.same_user_on_next_iteration">true</boolProp>
        <boolProp name="ThreadGroup.delayedStart">true</boolProp>
      </ThreadGroup>
      <hashTree>
        <HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy" testname="HTTP Request" enabled="true">
          <elementProp name="HTTPsampler.Arguments" elementType="Arguments" guiclass="HTTPArgumentsPanel" testclass="Arguments" testname="User Defined Variables" enabled="true">
            <collectionProp name="Arguments.arguments">
              <elementProp name="vote" elementType="HTTPArgument">
                <boolProp name="HTTPArgument.always_encode">false</boolProp>
                <stringProp name="Argument.value">Cats</stringProp>
                <stringProp name="Argument.metadata">=</stringProp>
                <boolProp name="HTTPArgument.use_equals">true</boolProp>
                <stringProp name="Argument.name">vote</stringProp>
              </elementProp>
            </collectionProp>
          </elementProp>
```

Figure 6: jmx sample file JMeter

The above diagram is showing the configuration where it has been set with many threads, ramp-up time, and duration. Different HTTP samplers are configured to capture HTTP responses and to get the summary report.

Note: Steps for all above tools integration(File Name- Infrastructure_SetUp_Steps_All) and other codes including analytical python given in the public GitHub repository mentioned in appendix A.

STAGE 3:ANALYTICAL UNIT LOCAL

CSV FILES AND JUPYTER NOTEBOOK

In this section, the major task is to get the monitoring data from Grafana in CSV format for all the categories and perform some manual pre-processing to make it meaningful. The CSV files only contain values of consumption and time series removed as all the experiments were performed in series. Also, the outlier detection is performed based on distance from the center of the dataset hence timestamp is not significant in this scenario. Later, centroid, distance from the centroid, and other statistical calculations were done along with visualizing outliers found while comparing metrics consumption as different layers. Jupyter notebook [36] was used to execute various python packages and APIs to analyze the data. Anaconda distribution had been used to perform python programming scripts to find the outliers. *Pandas, Matplotlib, Seaborn, and SKLearn* are the libraries used to achieve the goal of this study. These libraries helped to find the outliers while comparing the dataset. K-Means clustering had been used to quantize the vectors. These outliers comparison will show which area is most impacted by the corresponding fault injection type.

Once the above set up done, Figure 7 given below shows the process flow and the activity performed at each stage. The linking among different units shown above forms a system, which is used further to perform various tests.

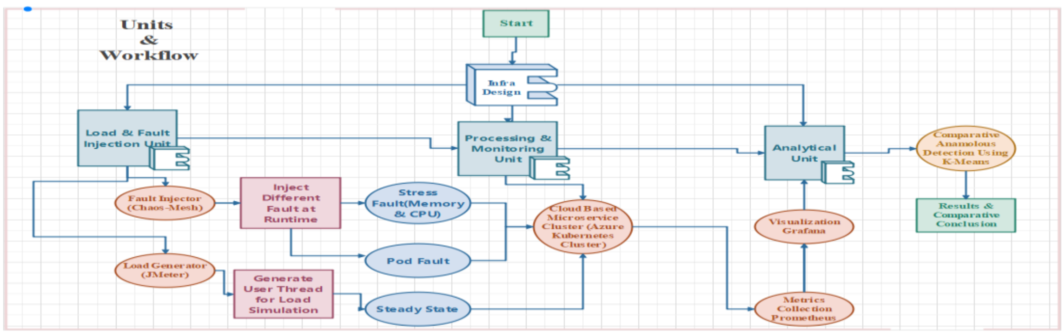

Units & Workflow
Start
Infra Design
Load & Fault Injection Unit
Processing & Monitoring Unit
Analytical Unit
Comparative Anamolous Detection Using K-Means
Fault Injector (Chaos-Mesh)
Inject Different Fault at Runtime
Stress Fault(Memory & CPU)
Cloud Based Microservice Cluster (Azure Kubernetes Cluster)
Visualization Grafana
Results & Comparative Conclusion
Pod Fault
Load Generator (JMeter)
Generate User Thread for Load Simulation
Steady State
Metrics Collection Prometheus

Figure 7: Units and Workflow

CHAPTER 3

This chapter describes the definition of important factors considered before actual experiments are performed along with different types of fault injection done on the test set done in the earlier section. Later, the metrics generated were exported into CSV and copied to the local machine for further analysis.

The web applications became more complicated as a result of the formation of distributed cloud architectures and microservices. More than ever, we are all relying on these systems, yet it has become much more difficult to anticipate if they may collapse. These outages cost businesses money loss. The disruptions make it difficult for customers to shop, do business, and complete their tasks. The cost of downtime is evolving into a KPI for many engineering teams since even brief disruptions can harm a company's bottom line.

A disciplined method for finding defects before they become outages is called chaos engineering. A system's resilience to stress can be evaluated proactively to identify flaws before they become apparent and to rectify them.

Principle of Chaos Engineering

Chaos Engineering involves running thoughtful, planned experiments that teach us how our systems behave in the face of failure[22].

The whole experiment run would follow the below-mentioned principle. It can vary according to the infrastructure setup however in general the entire test run will be performed by following the below-mentioned principles. Figure 8 illustrates the overall principles.

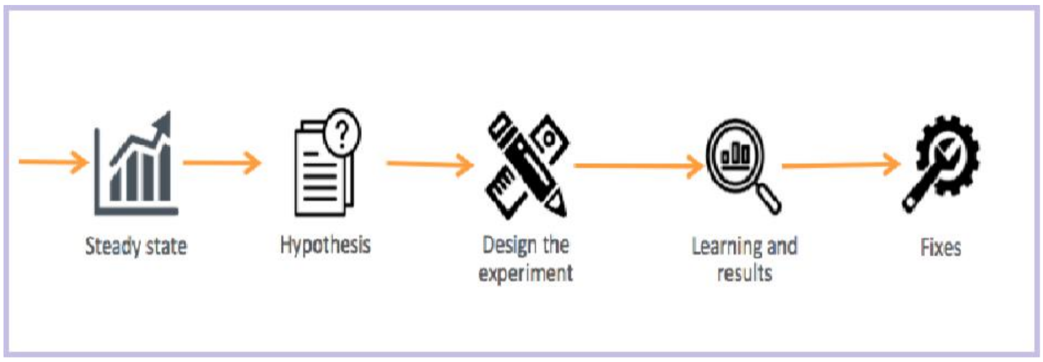


Figure 8: Principles of Chaos Experiments [23].

Steady State

In this phase, we define a measurable state that represents normal circumstances as a baseline and later different types of fault injection results would be compared to find the deviation from the hypothesis.

Hypothesis

Once steady-state metrics are set and an understanding of their steady-state behaviors done, the same can use to define the hypotheses and preferred results for the experiment. Starting with a small impact, choosing only one hypothesis at a time[23].

Design the Experiment

The experiment runs in multiple phases. It will start from small and gradually increase the intensity to identify the resiliency of the application in the stressed state. The below diagram shows the way this experiment will create a hypothesis and compare it against the steady state defined earlier.

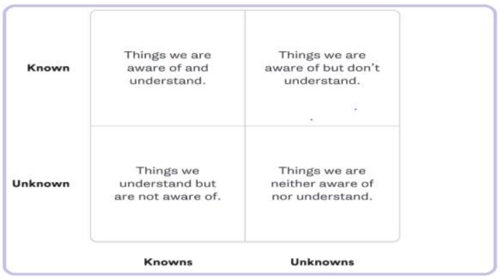

Known
Unknown
Things we are aware of and understand.
Things we are aware of but don't understand.
Things we understand but are not aware of.
Things we are neither aware of nor understand.
Knowns
Unknowns

Figure 9: Experiment Design [22].

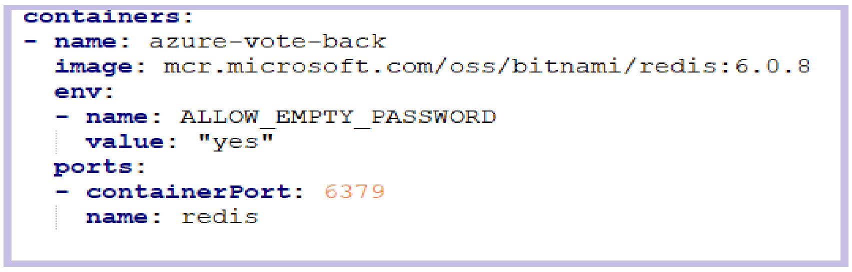
LEARNING & RESULTS

This phase is self-explanatory to draw the learnings from the above-performed steps. It would be helpful to find the potential area of vulnerability.

STEADY STATE EXPERIMENTAL RUN:

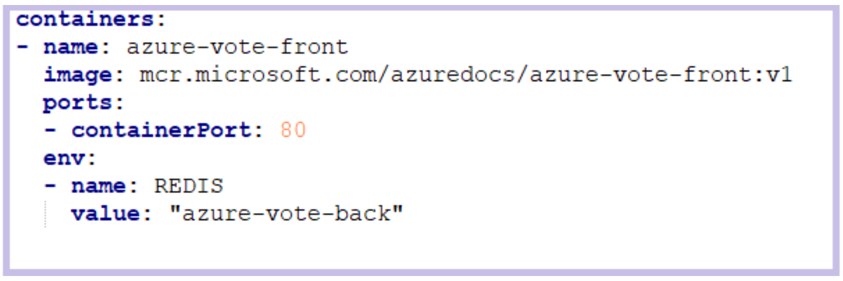
This section describes the steady state run for the application on test infrastructure. The same will be used while injecting the different faults in later sections. In this study, the purpose of the Steady state is to see the impact of fault injection when an application runs with an operationally normal load. This definition is not for comparing however just to benchmark the user load and to make sure the application is running steadily.

The application runs under a POD as a container, and it is virtualized based on *namespaces*. The metrics for the default namespace are collected at the *cluster* layer followed by *Node* and *Pod* level metrics. The response time of the API endpoints was collected as the graph from the JMeter HTTP sampler.The container configuration has been set up which can be altered as per the change in cluster capacity. In this experiment, it is set as below picture to get it running optimally.



```
containers:
- name: azure-vote-back
  image: mcr.microsoft.com/oss/bitnami/redis:6.0.8
  env:
  - name: ALLOW_EMPTY_PASSWORD
    value: "yes"
  ports:
  - containerPort: 6379
    name: redis
```

Figure 10: Backend Configuration [26].



```
containers:
- name: azure-vote-front
  image: mcr.microsoft.com/azuredocs/azure-vote-front:v1
  ports:
  - containerPort: 80
  env:
  - name: REDIS
    value: "azure-vote-back"
```

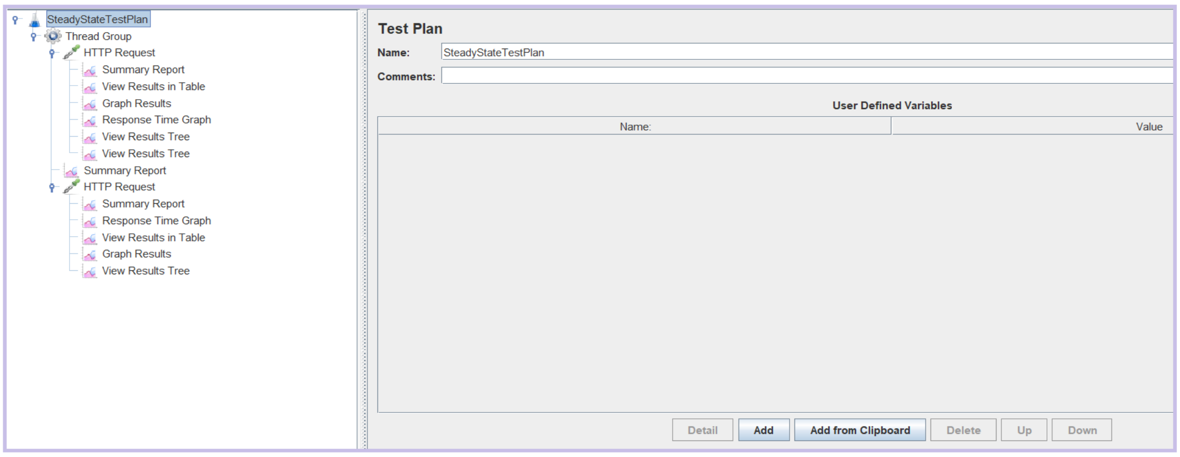
Figure 11: Frontend Configuration [26].

Both figures mentioned above show the code snippet of the container definition. Containers can be identified with the name parameter mentioned along. The image path and the port are also mentioned in both the figures.

EXPERIMENT 1:STEADY STATE DEFINITION RUN

In this section, details are given on how the load is generated on the AKS cluster using JMeter along with the parameters calculated. These parameters' values will be considered as **steady-state value**s and the same will be set when injecting fault at different layers.

Defined a JMeter test plan called “**Steady State Test Plan**”. The test plan shows all the *HTTP* requests and listeners to capture a different aspect of the test simulation.



- SteadyStateTestPlan
  - Thread Group
    - HTTP Request
      - Summary Report
      - View Results in Table
      - Graph Results
      - Response Time Graph
      - View Results Tree
      - View Results Tree
    - Summary Report
    - HTTP Request
      - Summary Report
      - Response Time Graph
      - View Results in Table
      - Graph Results
      - View Results Tree

Test Plan

Name: SteadyStateTestPlan

Comments:

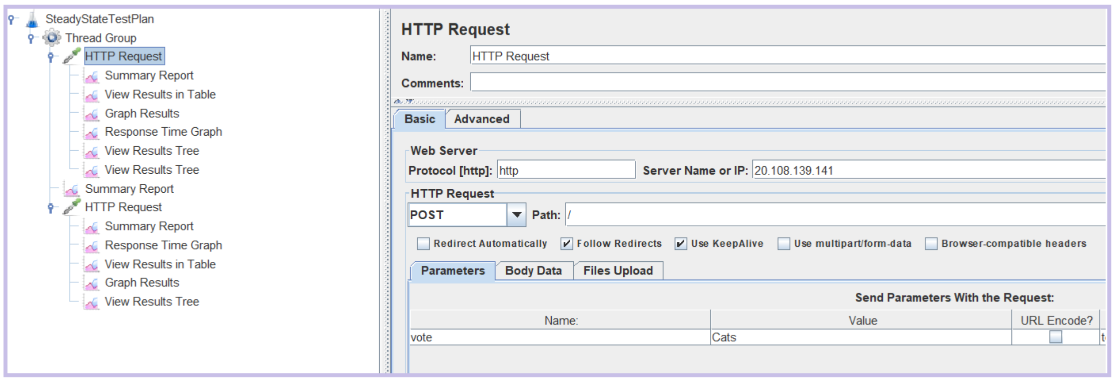
User Defined Variables

Name:	Value

Figure 12: Steady State Test Plan

Referring to Figure 12 above, there are two *HTTP request sampler*s configured that send the request to the two different endpoints of the application. These endpoints will be targeted to hit while generating the virtual users and stress the CPU and Memory at a different layer of infrastructure. The microservice is hosted on the AKS cluster and the following diagrams are showing the sampler setup for individual API endpoints. There are two endpoints in this set up which handles two voting categories. It consists of the parameter "vote" with two values "Dogs" and "Cats". These individuals are targeted in the separate sampler and user load generated at the same time successively. The hitting probability to these endpoints is random and decided at run time by JMeter, with No control at the user end.

Cats Endpoint: http://20.108.139.141/vote/Cats



SteadyStateTestPlan
- Thread Group
  - HTTP Request
    - Summary Report
    - View Results in Table
    - Graph Results
    - Response Time Graph
    - View Results Tree
    - View Results Tree
  - Summary Report
  - HTTP Request
    - Summary Report
    - Response Time Graph
    - View Results in Table
    - Graph Results
    - View Results Tree

HTTP Request

Name: HTTP Request

Comments:

Basic | Advanced

Web Server

Protocol [http]: http Server Name or IP: 20.108.139.141

HTTP Request

POST Path: /

☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers

Parameters | Body Data | Files Upload

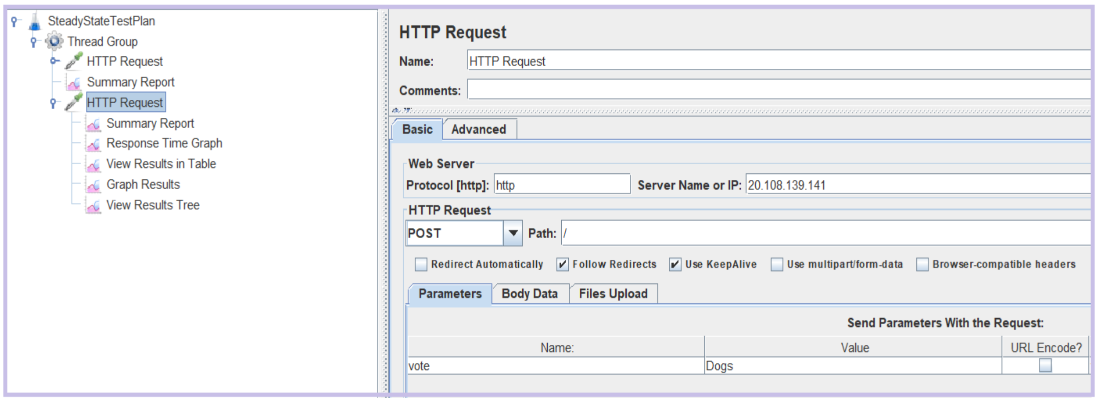
Send Parameters With the Request:

Name:	Value	URL Encode?
vote	Cats	☐

Figure 13: Cats Endpoint HTTP Sampler

The above figure shows the configuration of the sampler set to capture the different values for the Cats endpoint. It shows that the POST method is used to add the vote for Cats with the parameter "vote". The server IP is configured with a value when the test is performed, and it can be changed based on different IPs for the different clusters.

Dogs Endpoint: http:/20.108.139.141/vote/Dogs



SteadyStateTestPlan
- Thread Group
  - HTTP Request
  - Summary Report
  - HTTP Request
    - Summary Report
    - Response Time Graph
    - View Results in Table
    - Graph Results
    - View Results Tree

HTTP Request

Name: HTTP Request

Comments:

Basic | Advanced

Web Server

Protocol [http]: http Server Name or IP: 20.108.139.141

HTTP Request

POST Path: /

☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers

Parameters | Body Data | Files Upload

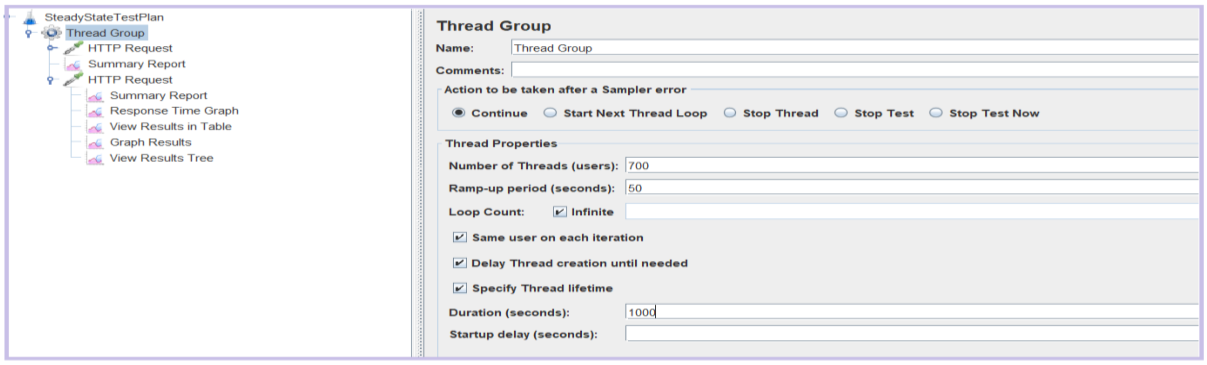
Send Parameters With the Request:

Name:	Value	URL Encode?
vote	Dogs	☐

Figure 14: Dogs Endpoint HTTP Sampler[37].

A similar configuration set up done for the Dogs endpoint, shown in Figure 14. Here the parametric value is changed to “Dogs “ from Cats. This will capture the vote given for Dogs.

Furthermore, 700 virtual user threads represent real users while creating HTTP traffic for HTTP samplers. or the application URLs. Ramp-up time for the user load is set to 50 seconds so that heavy loads are not delivered all at once to avoid unduly taxing the AKS cluster. It had been tested in many iterations to reach this number based on the capacity of the AKS Cluster. This needs to evaluate individually for each cluster set up. The value given is specific to this setup and cannot be generalized. The application performed operationally steady however there were some instances when the application was not available however it recovered soon hence, this can be considered as operational impact to end user is very minimal. The below diagram shows the setup at the thread group level. There are a few listeners also configured to capture the other parameters which can tell us about the performance of the *HTTP* request and Response.


SteadyStateTestPlan
- Thread Group
  - HTTP Request
    - Summary Report
  - HTTP Request
    - Summary Report
    - Response Time Graph
    - View Results in Table
    - Graph Results
    - View Results Tree

Thread Group

Name: Thread Group

Comments:

Action to be taken after a Sampler error: [✓ Continue] Start Next Thread Loop / Stop Thread / Stop Test / Stop Test Now

Thread Properties

Number of Threads (users): 700

Ramp-up period (seconds): 50

Loop Count: [✓ Infinite]

[✓] Same user on each iteration

[✓] Delay Thread creation until needed

[✓] Specify Thread lifetime

Duration (seconds): 1000

Startup delay (seconds):

Figure 15: Thread Group Setup [37].

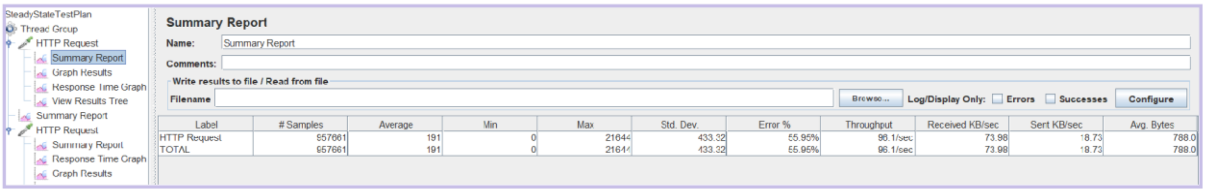
Different listeners are configured to capture and display test metrics to validate the correctness of the test run. In this experiment, the concerned Listers are “Summary Report” and “Response Time Graph”.

Summary Report [37]: It provides the aggregated values like, how many samples are, what is an average, min, max, throughput, etc.

Response Time Graph[37]: This lister captures and displays the response time of the endpoint hit at the Azure Kubernetes Cluster (AKS). The response time is captured into milliseconds for each request sent to the endpoint.

The test starts with a small set of threads and ultimately the thread setup diagram shows the final value which is considered the steady-state threshold value. Below gives a diagram that validates the test run by showing the outcomes of the listers configured for both endpoints.

The below diagram shows the summary of the entire run for both *APIs* end points consecutively.


SteadyStateTestPlan
- Thread Group
  - HTTP Request
    - Summary Report
    - Graph Results
    - Response Time Graph
    - View Results Tree
  - Summary Report
  - HTTP Request
    - Summary Report
    - Response Time Graph
    - Graph Results

Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

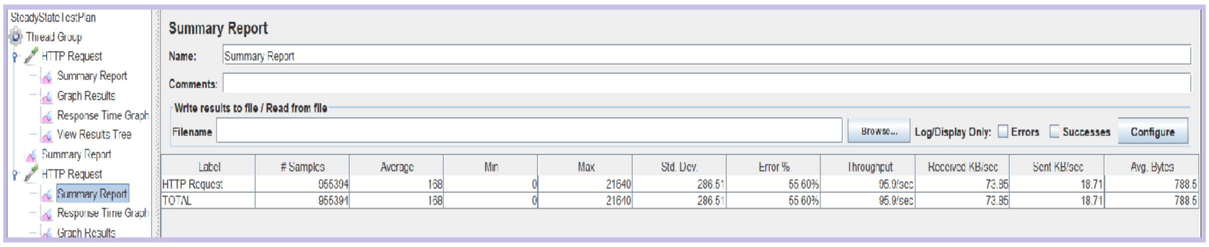
Filename

Log/Display Only: Errors / Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	957661	191	0	21644	433.32	55.95%	96.1/sec	73.98	18.73	788.0
TOTAL	957661	191	0	21644	433.32	55.95%	96.1/sec	73.98	18.73	788.0

Figure 16: Summary Report Cats endpoint

From Figure 16, it can be observed that more than 0.9 million samples hit the Cats API endpoint and it also shows that the throughput is 96.1 %. Similarly for the Dogs endpoint. The observation here is that both the summary looks almost identical, and this points that the load is distributed evenly and no unwanted spike.


SteadyStateTestPlan
- Thread Group
  - HTTP Request
    - Summary Report
    - Graph Results
    - Response Time Graph
    - View Results Tree
  - Summary Report
  - HTTP Request
    - Summary Report
    - Response Time Graph
    - Graph Results

Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename

Log/Display Only: Errors / Successes

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	955394	168	0	21640	286.51	55.60%	95.9/sec	73.85	18.71	788.5
TOTAL	955394	168	0	21640	286.51	55.60%	95.9/sec	73.85	18.71	788.5

Figure 17: Summary Report Dogs endpoint

Consequently, Figure 18 shows the HTTP response time for the Cats endpoint. Initially, the sudden spike had been observed however later it came down causing the cluster to scale out to handle the sudden increase in load. With this response time, the application was smoothly operable.

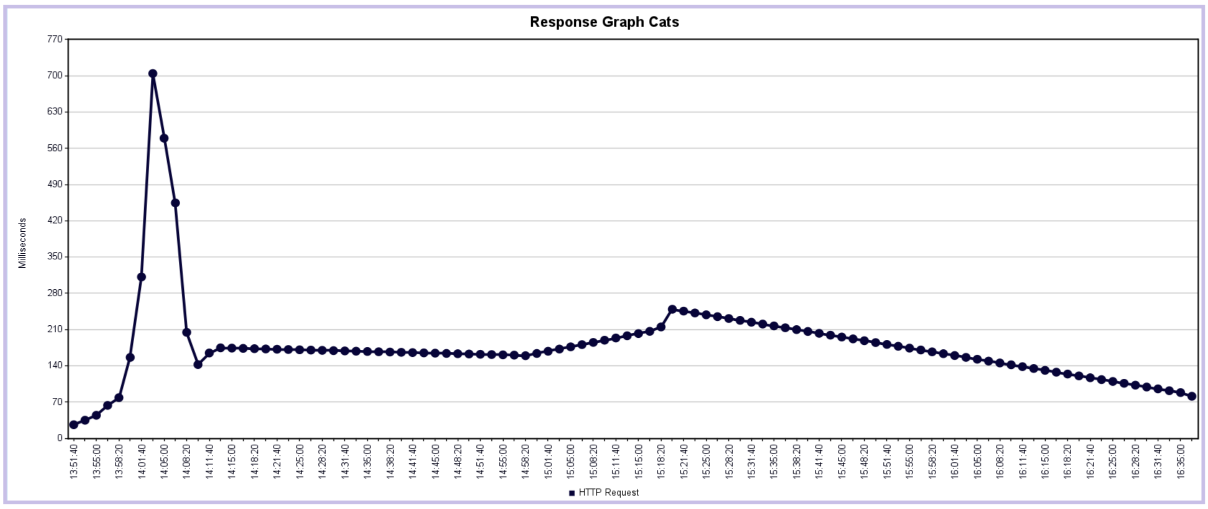
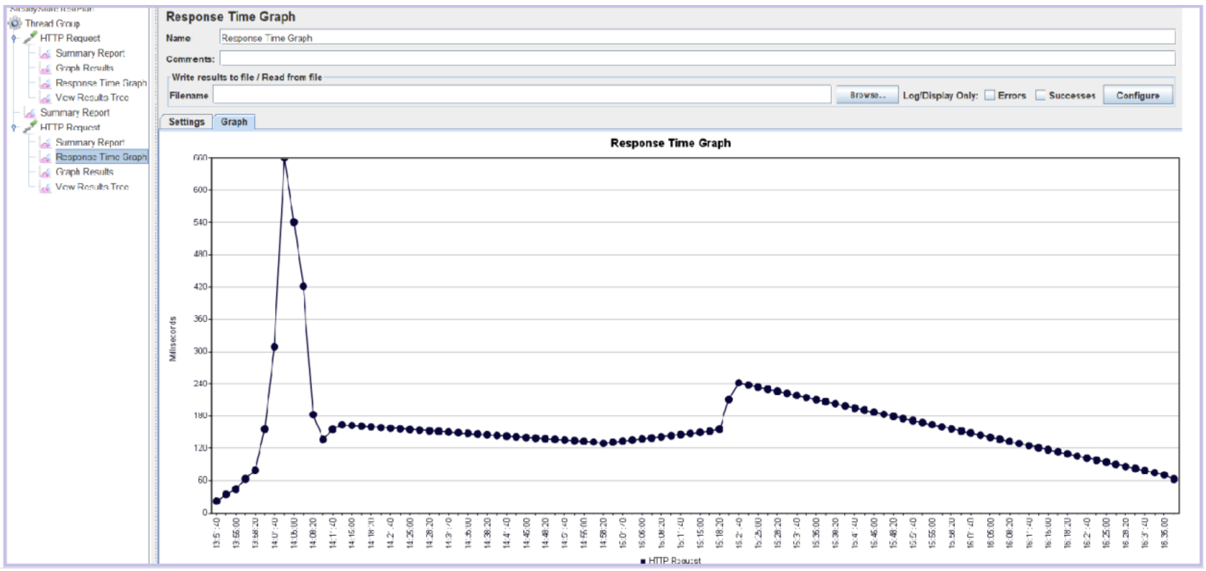


Figure 18: Response Time Graph Cats endpoint

This will be considered for the further section fault injection. This state of the application is in line with the motive of this research.

Similarly, for the Dogs endpoint, Figure 19 shows the response time. It shows variation in pattern as compared to the Cats endpoint. The reason behind this is that the probability to get a hold on the vote at a specific point in time is random, Dogs endpoint may get less hit and it resulted in a little more response time than earlier.


Response Time Graph

Name: Response Time Graph

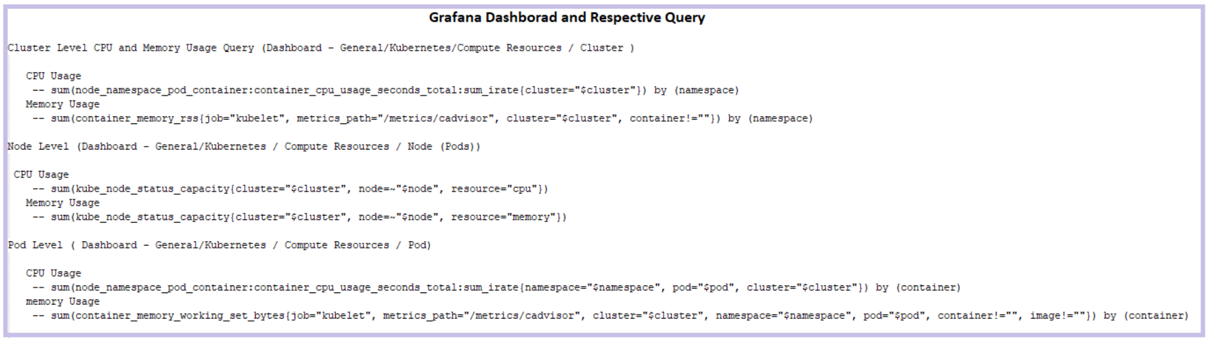
Comments:

Write results to file / Read from file

Filename:

Figure 19: Response Time Graph Dogs endpoint

Further, time to check how it was doing at different layers by observing the different metrics in Grafana. There are different Dashboards used to fetch the metrics. Figure 20 is showing the query used to capture the metrics data for further analysis. All the other test results will also follow the same query for each layer's data to keep it all uniform and try to find the pattern in different layers.


Grafana Dashborad and Respective Query

```
Cluster Level CPU and Memory Usage Query (Dashboard - General/Kubernetes/Compute Resources / Cluster )

   CPU Usage
    -- sum(node_namespace_pod_container:container_cpu_usage_seconds_total:sum_irate{cluster="$cluster"}) by (namespace)
   Memory Usage
    -- sum(container_memory_rss{job="kubelet", metrics_path="/metrics/cadvisor", cluster="$cluster", container!=""}) by (namespace)

Node Level (Dashboard - General/Kubernetes / Compute Resources / Node (Pods))

 CPU Usage
    -- sum(kube_node_status_capacity{cluster="$cluster", node=~"$node", resource="cpu"})
   Memory Usage
    -- sum(kube_node_status_capacity{cluster="$cluster", node=~"$node", resource="memory"})

Pod Level ( Dashboard - General/Kubernetes / Compute Resources / Pod)

   CPU Usage
    -- sum(node_namespace_pod_container:container_cpu_usage_seconds_total:sum_irate{namespace="$namespace", pod="$pod", cluster="$cluster"}) by (container)
   memory Usage
    -- sum(container_memory_working_set_bytes{job="kubelet", metrics_path="/metrics/cadvisor", cluster="$cluster", namespace="$namespace", pod="$pod", container!="", image!=""}) by (container)
```

Figure 20: Metrics Query Used in Grafana [31].

It can be seen that the CPU utilization and memory at the cluster level use namespace to collect the data. In this experiment, *CPU and Memory* both collected for only the default namespace as the application deployed in that.

At the Node level, query using *kube_node_status_capacity*, represents the capacity of different resources at the node, for us, two resources, *CPU* and memory collecting.

Similarly, for the Pod level, it collects data using *the sum_irate* function, which collected namespace, pod, and cluster.

Moving further based on query definition, further, the usage observation was captured. It can be observed from Figure 21 that the spike at different layers for the metrics collected. It shows the uniform usage of CPU and memory at Cluster, node, and pod level except for a few instances showing greater spike, which was due to the pause and sudden start of the test with varied user load.

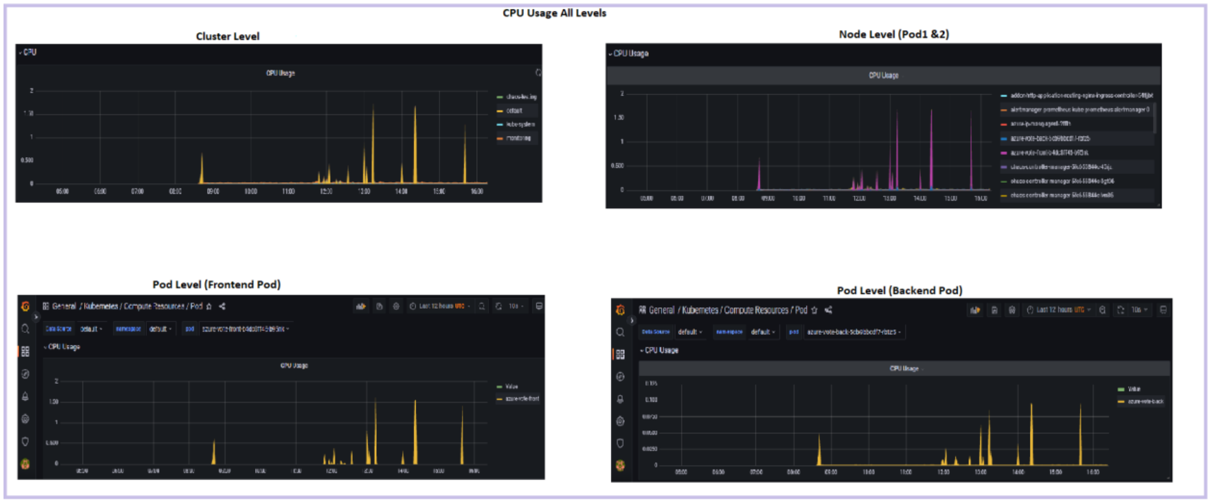


Figure 21: CPU Usage at different layers

The highest peak depicts the threshold user thread as mentioned in earlier sections. During the test blast radius method followed in which the user load gradually increased to avoid the sudden breakdown of the entire cluster.

Similarly, Figure 22 shows a similar pattern for memory usage. The observation can be made that the memory consumption at the pod level is different and it shows that the backend pod was consuming more memory as compared to the frontend pod. The reason may be cause most of the API call was for *the Post* method and the backend pod was responsible to process data and save it in the backend.

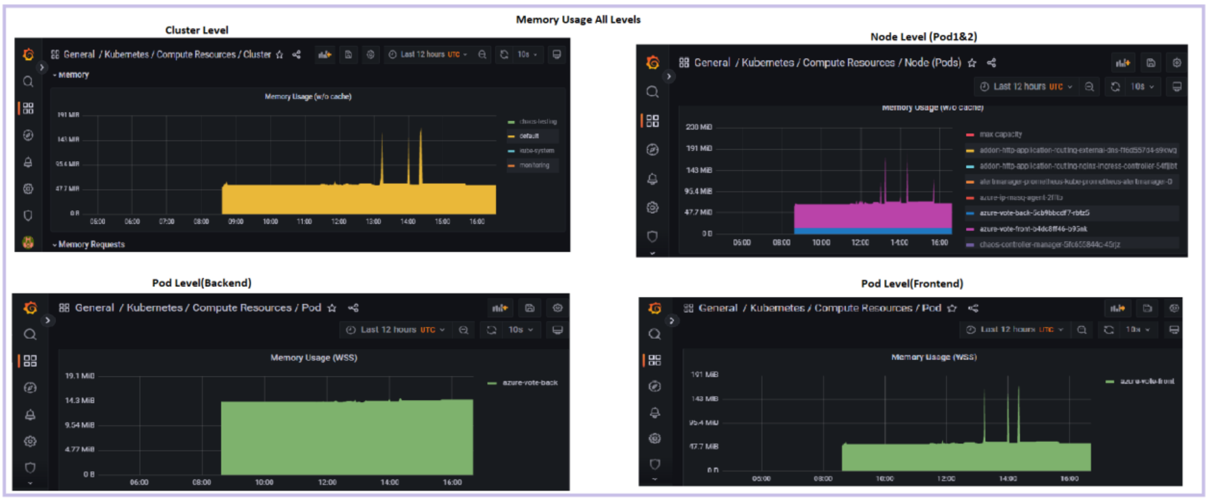


Figure 22: Memory Usage All Levels

So, at this point, it can be concluded that the user load mentioned in the Thread group configuration is the maximum and the application can sustain it operationally. Maximum capacity means the application may crash for some time however it can recover automatically as the server auto-scale itself to handle the increased load. In the coming sections when the chaos fault injection is designed, the load should not be increased the total number. So, load through JMeter should be proportional to the load injected as a fault from Chaos Mesh.

EXPERIMENT 2:FAULT INJECTION

Our primary goal is to evaluate which layer of our infrastructure shows more deviation while injecting specific faults to microservice running under a steady state. So, two types of fault injection are performed, firstly, *Stress Fault* and later *Pod Faults.* This fault injection will be targeting the CPU, Memory, and Pods.

TYPE 1 - STRESS EXPERIMENT

Since we are evaluating the impact of overall infrastructure while injecting the fault at the namespace and pod level. In this stress experiment, the fault was injected to create more load on *CPU* and *Memory* metrics.

Hypothesis: Stressing CPU usage of both pods one at a time under default namespace. Targeting both Pods individually. The CPU and Memory usage may show some spike in the consumption of the targeted pod however it should recover itself. There should not be any unexpected variation as compared to steady state metrics. Once the attack duration is finished, the pod should return to normal.

ATTACK DESCRIPTION:

Since the attack on the target Pod will occur while the application is running under a steady state, it means it would have already used some portion of resources. Hence this experiment will follow a blast radius and start from a small duration along with small resource exhaustion. In this experiment, a load of total users was identified during the Steady-state definition divided between the user load given through Chaos-mesh and JMeter. The purpose of this is to prevent the application from going into an unrecoverable state.

Here once the experiment starts, as given in Figure 23, stressor-type *memory* starts the workers and runs for the specified duration, it grows its heap size in each iteration by recreating the heap in each iteration. *Workers*, specify the number of threads going to be used to stress the memory of the cluster and *size*, which points to the occupancy limit. In this experiment, 90 % of the memory resource consumption is allowed along with 500 threads going to be used for the specified duration. It stops creating new heap allocation until the duration ends. The mode is used to specify all the eligible pods according to the criteria mentioned. Advised at this point to not run the test for a longer period otherwise cluster may crash or may become non-responsive.

Similarly for CPU stressors, as the experiment starts the *workers*' parameter shows how many threads are going to spawn and the percentage of CPU utilized. In definition file shown in Figure 23 shows the *CPU* stressor and the *load* parameter set to 90 %. Hence the spawned threads can use up to 90 % of the total *CPU.* 10 % left cause other services are also running in the cluster, and it should go into the unresponsive phase. During the initial test, it was observed that if try to push more than 700 users, the application went into unrecoverable mode and the cluster crashed completely, hence different iterations were performed however always made sure that the combination of both should not exceed the threshold value for this set, which is 700 users. Started with a small load and gradually, the resource consumption was increased until the max value reached.

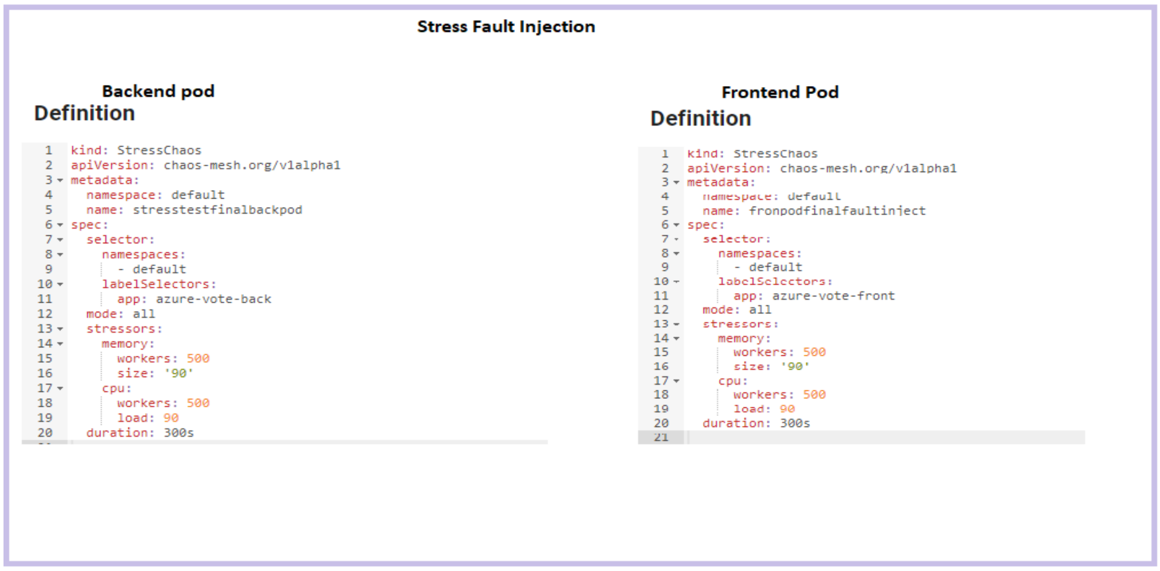
Below given are the values used to load the user threads and these values come under the steady state definition value mentioned in the steady state definition.

STEADY STATE LOAD

Experiment Run	Number of Threads(User)	Ramp-Up Period(Seconds)	Duration(Seconds)
Steady State Load	200	50	500

Table 2: Steady State Value Used for Stress Fault

The stress Test definition file is below for both the pods along with memory and CPU load. Figure 23 shows that the fault was injected with a maximum capacity of both the pods along with a steady state (200 + 500= 700). It is the final definition file that was executed after following multiple small impact radii to avoid unrecoverable damage to the infrastructure and application.



```yaml
kind: StressChaos
apiVersion: chaos-mesh.org/v1alpha1
metadata:
  namespace: default
  name: stresstestfinalbackpod
spec:
  selector:
    namespaces:
      - default
    labelSelectors:
      app: azure-vote-back
  mode: all
  stressors:
    memory:
      workers: 500
      size: '90'
    cpu:
      workers: 500
      load: 90
  duration: 300s
```

```yaml
kind: StressChaos
apiVersion: chaos-mesh.org/v1alpha1
metadata:
  namespace: default
  name: fronpodfinalfaultinject
spec:
  selector:
    namespaces:
      - default
    labelSelectors:
      app: azure-vote-front
  mode: all
  stressors:
    memory:
      workers: 500
      size: '90'
    cpu:
      workers: 500
      load: 90
  duration: 300s
```

Figure 23: Stress Fault Injection Definition File

Figure 24 is showing how the stress test started from a small value and gradually increased to the max under which application would not break. Here, it is considered that the load should not increase beyond the threshold defined in the steady state definition. The different color dots depict the individual fault injection run and the corresponding time.

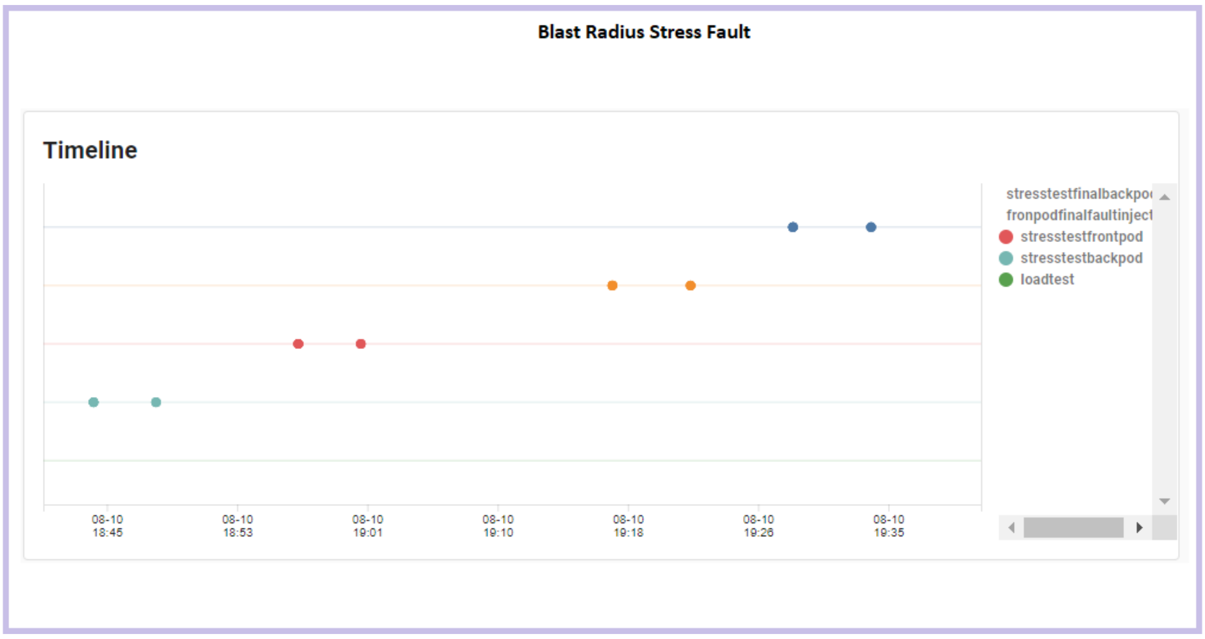


Figure 24: Blast Radius Stress Fault

Results and Analysis: As per the goal of this entire study, the collected metrics are reviewed to find if any unknown information is captured. Once, the stress started, the individual pods start consuming more CPU and Memory from the underlying hardware. Here the container definition does not set the limit of consumption hence, the container keeps increasing the capacity consumption as per the increasing load during the test run. This can be observed as the spike in response to the API endpoint and also the graph-based observation.

Figure 25 shows the results of the first test performed in many iterations. It can be observed that the spike for smaller stress has the least impact however as it increases it shows the respective hike.

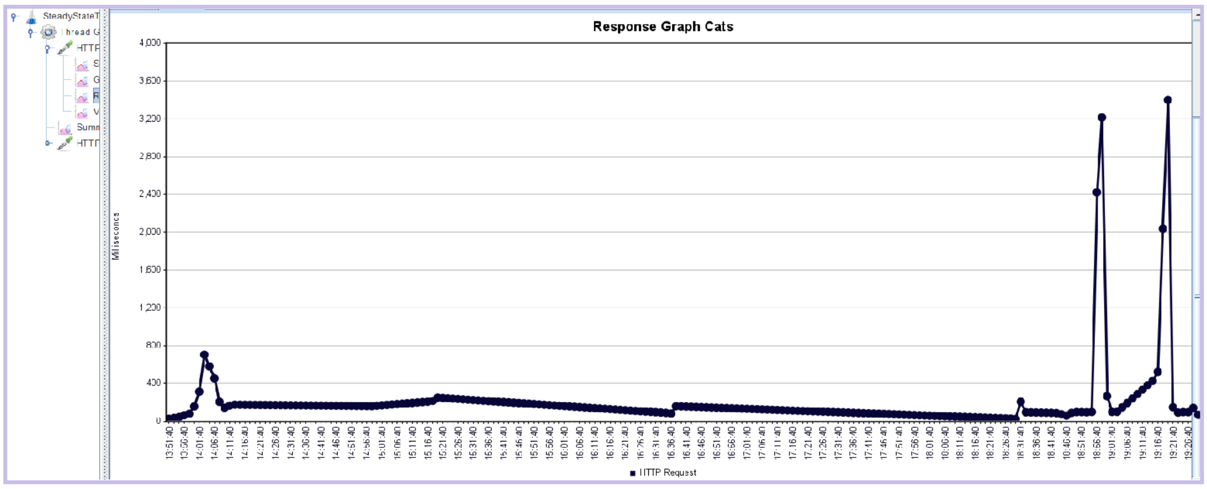


Figure 25: HTTP Response

A sudden spike is observed in response from the *API* as expected in the hypothesis. Once the fault injection duration finished the HTTP response came back to its normal timing as if were before injection. The end data points in the above figure points the same.

Let's evaluate how it impacted the CPU and Memory Usage at Cluster, Node, and Pod levels.

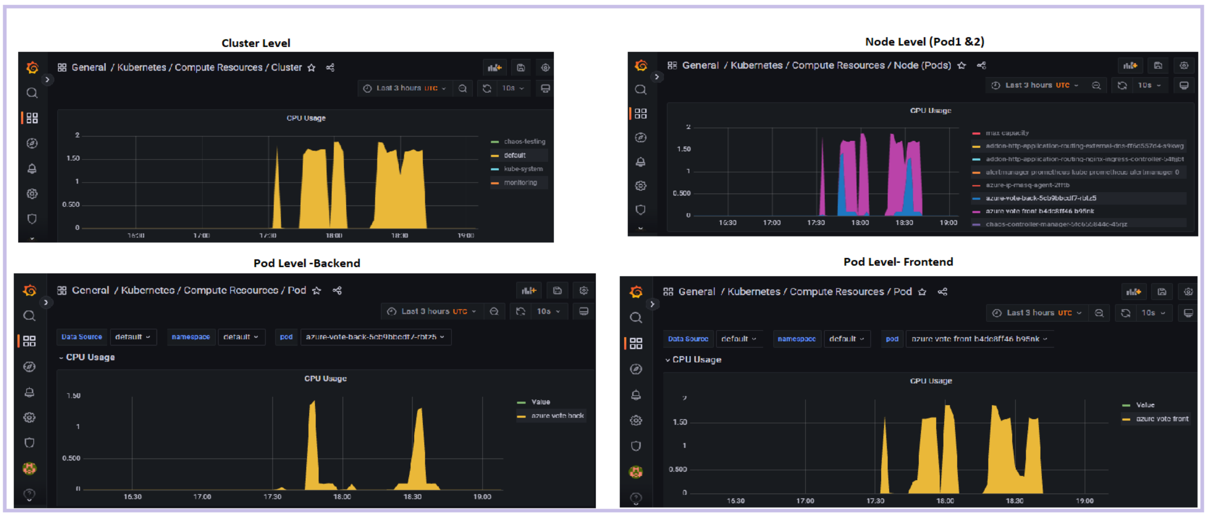


Figure 26: CPU Usage All Levels

Figure 26 depicts that there was a significant load handled by the front-end pod at the pod level. This seems to be an interesting observation that can give us some information as to why this is different however if it is compared with the steady state, the backend pod given in Figure 21 was more actively consuming the resources due to *the post* method of HTTP. However, in this observation, the front pod is impacted most due to unknown reasons.

Let’s observe how was memory consumption across the layers. Below given figure shows the usage at all the layers.

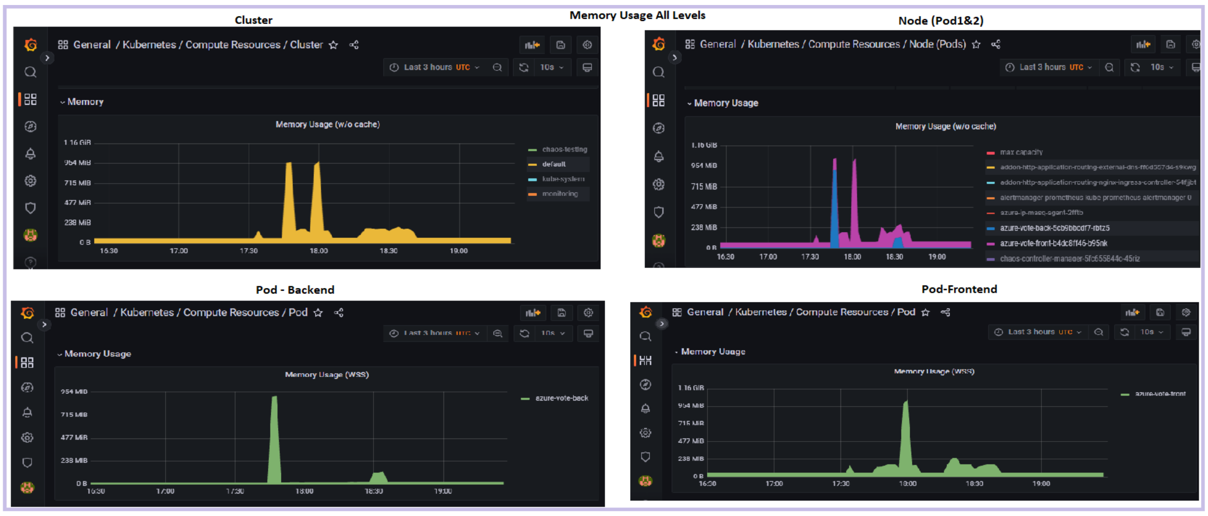


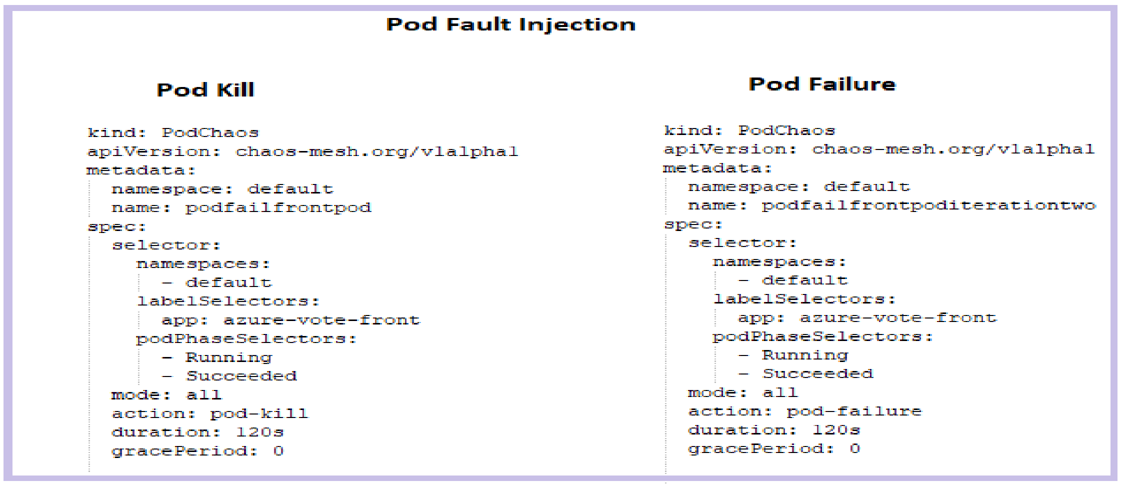
Figure 27: Memory Usage All Levels

The clear observation was as expected however memory consumption is following the same pattern as CPU utilization. It can also be observed that the frontend pod is utilizing more memory as compared to the backend however it should be reciprocal to this.

EXPERIMENT 3: POD FAULT

POD FAILURE AND POD KILL:

In this experiment, two definition files were created with pod failure and the other with Pod Kill. Figure 3.3.13 depicts the same for this experiment. The hypothesis is given further.


Pod Fault Injection

Pod Kill

```
kind: PodChaos
apiVersion: chaos-mesh.org/v1alpha1
metadata:
  namespace: default
  name: podfailfrontpod
spec:
  selector:
    namespaces:
      - default
    labelSelectors:
      app: azure-vote-front
    podPhaseSelectors:
      - Running
      - Succeeded
  mode: all
  action: pod-kill
  duration: 120s
  gracePeriod: 0
```

Pod Failure

```
kind: PodChaos
apiVersion: chaos-mesh.org/v1alpha1
metadata:
  namespace: default
  name: podfailfrontpoditerationtwo
spec:
  selector:
    namespaces:
      - default
    labelSelectors:
      app: azure-vote-front
    podPhaseSelectors:
      - Running
      - Succeeded
  mode: all
  action: pod-failure
  duration: 120s
  gracePeriod: 0
```

Figure 28: Pod Fault Injection Definition File

Hypothesis: When Pod failure is injected, the expected outcome is that the specified pod should not be available for the duration given in the definition file, on the other hand, once Pod Kill is injected, the specified pod is killed for that duration. The expected hypothesis is that if Replica set is configured then, it should restart the killed Pod and server the incoming request else it will show an error, and the pod never gets started if such mechanism is not in place. The Pod must recover from failure and automatically recreate the killed pod.

ATTACK DESCRIPTION:

Since the attack creates an interruption in the pod lifecycle while the application is running under a steady state. The cloud-based distributed application is meant to be auto-recovered to handle the situation. In this experiment each scenario, pod failure, and Pod kill are executed in multiple iterations, The pod failure makes sure the specified pod should not be available until the experiment duration ends however in terms of Pod Kill, Kubernetes terminates all the containers which are part of the specified pod. Kubernetes sends *SIGTERM* to the processes before killing for specified timing, in this experiment, this is known as *gracePeriod*. The value here is set to 0, which means as soon as it sends the signal, it kills the pod immediately without waiting. To kill the process *SIGKILL* signal is sent, and it is terminated. This value could be configured in a different scenario however in this experiment, it is expected to keep it killing and see if any deviation in any other metrics due to this continuous kill command.

Note: The scenario may differ in the case of a *stateful* set, cause the stateful set always maintains the order of deployment, and their difference can be observed however that is not the scope of this paper.

Table 3 shows the load supplied while running the pod faults to the cluster. In these experiments, no thread load was generated by fault injection hence max capacity load was given to gauge the performance.

STEADY STATE LOAD

Experiment Run	Number of Threads(User)	Ramp-Up Period(Seconds)	Duration(Seconds)
Steady State Load	700	50	200

Table 3: Steady Load for Pod Fault

Figure 29 shows the blast radius strategy followed here and there were many iterations used to do it in a controlled way to avoid unresponsiveness of the cluster.

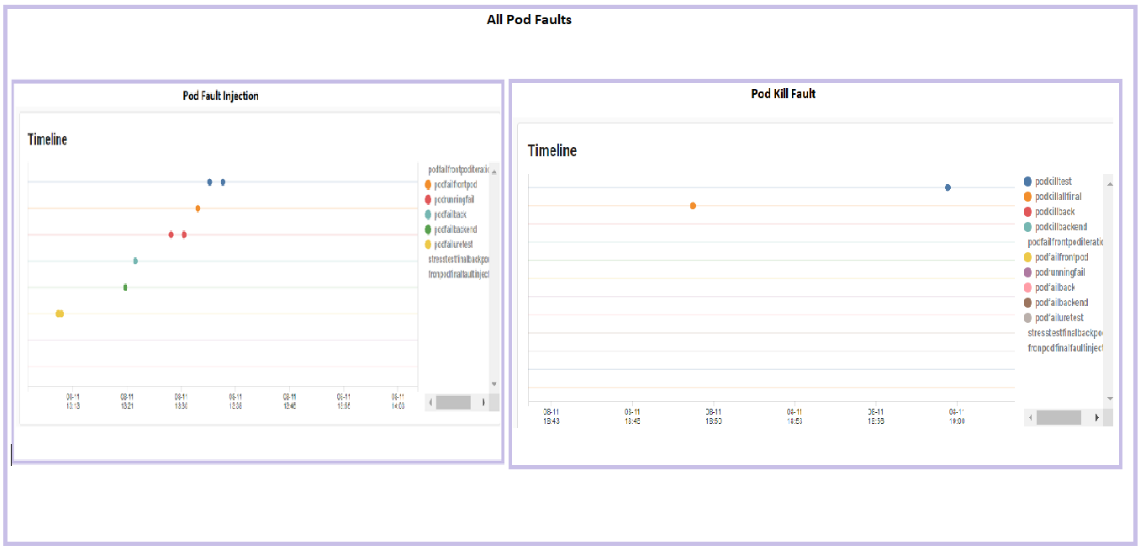


Figure 29: Pod Fault Injections

It can be observed that a significant number of iterations were performed with different duration and then the metrics were captured for further analysis.

Results and Analysis: As per the goal of this entire study, the collected metrics are reviewed to find if any unknown information is captured. Once, the pod faults started individually, it was observed that during POD failure, the application recovers once the experiment duration finished however during this period, it shows error, and the application was not accessible. On the other hand, when POD Kill is introduced, the application never comes back when both the main and replica pod gets eligibility to be killed else it also recovers if we pass the failed status out of eligibility criteria while configuring the pod kill. It means if Replica Set is available, it restarts. Hence the results of the test are according to the hypothesis done before fault injection.

Let's observe if any impact on the response, CPU, or memory consumption during this period. Figures 30 and 31 are showing the results of the tests performed in many iterations. It can be observed that the spike for smaller stress has the least impact however as it increases it shows the respective hike. Graphs are showing the downward trend until the next iteration starts. Two spikes are shown in both the first depicts when the pod failure was introduced, and it became steady after receiving the initial spike.

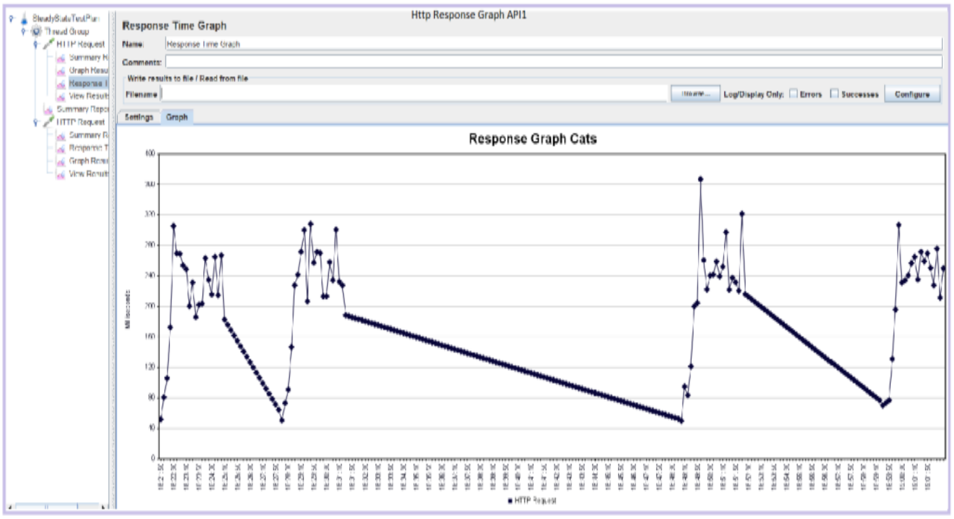


Figure 30: HTTP Response Cats Endpoint

The figure below shows little deviation as compared to the first endpoints in terms of the response which can be bigger if the scale of the test or duration can be increased. This proves that terms of response from the endpoint do not impact much observed however this may change in the case of tests performed on a larger scale.

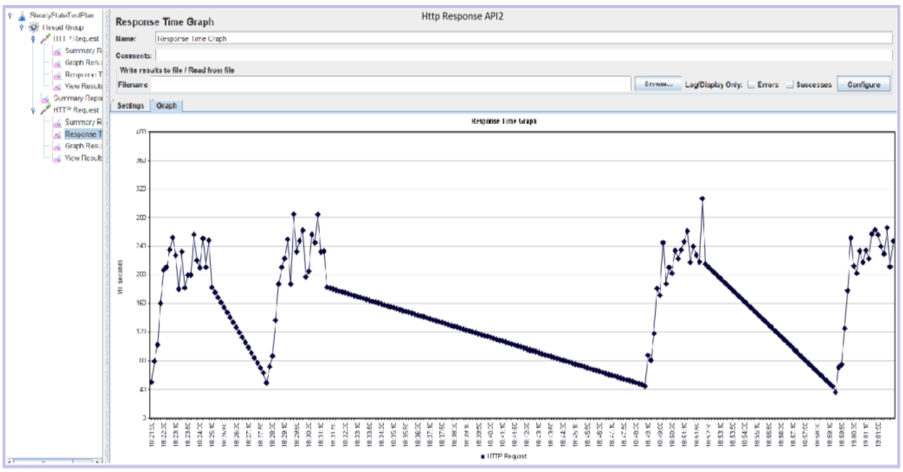


Figure 31: HTTP Response Dogs Endpoint

However, if these graphs are compared with the Stress fault trend mentioned in Figure 25, it shows a huge deflection. It means that the pod failure impacts the response instantly as the pod was not available and the demon keep looking for restoration, this could be the reason behind it.

Now let's observed the impact on the CPU and Memory utilization at a different layer of the cluster. The following section shows the individual impact on CPU and Memory for Pod Failure and Pod Kill. Figure 32 is showing the trend for CPU at different layers for pod fault. If compare with Figure 26 which showed the CPU in stress fault injection, here the trend is more uniform, and not much spike is observed at any layer. This proves that the impact of pod failure at the CPU level is uniform however as compared to the stress fault given in Figure 26, it is significant.

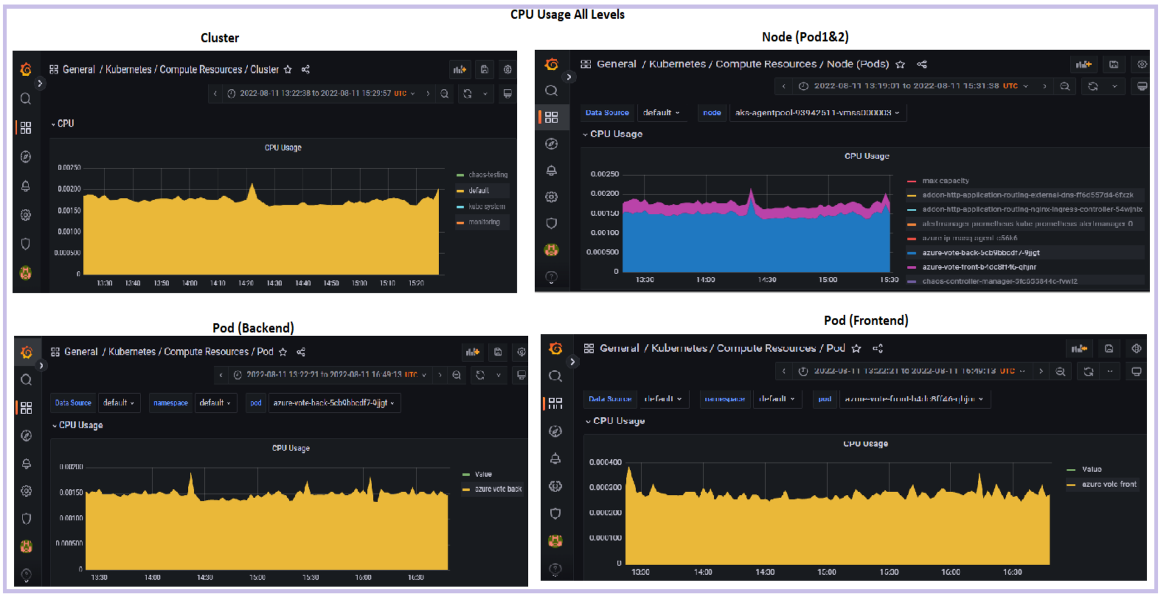

CPU Usage All Levels
Cluster
Node (Pod1&2)
Pod (Backend)
Pod (Frontend)
General / Kubernetes / Compute Resources / Cluster
General / Kubernetes / Compute Resources / Node (Pods)
General / Kubernetes / Compute Resources / Pod
CPU Usage
chaos-testing
default
kube-system
monitoring
azure-vote-back
azure-vote-front

Figure 32: CPU Usage for Pod Failure

Similarly, if we compare the below give figure's Memory usage with Figure 27 of the stress test, it can be understood that these failures injected in pods, the impact is significant as it keeps trying to restore however because the grace is zero, it instantly removes and does not get time to re-create.

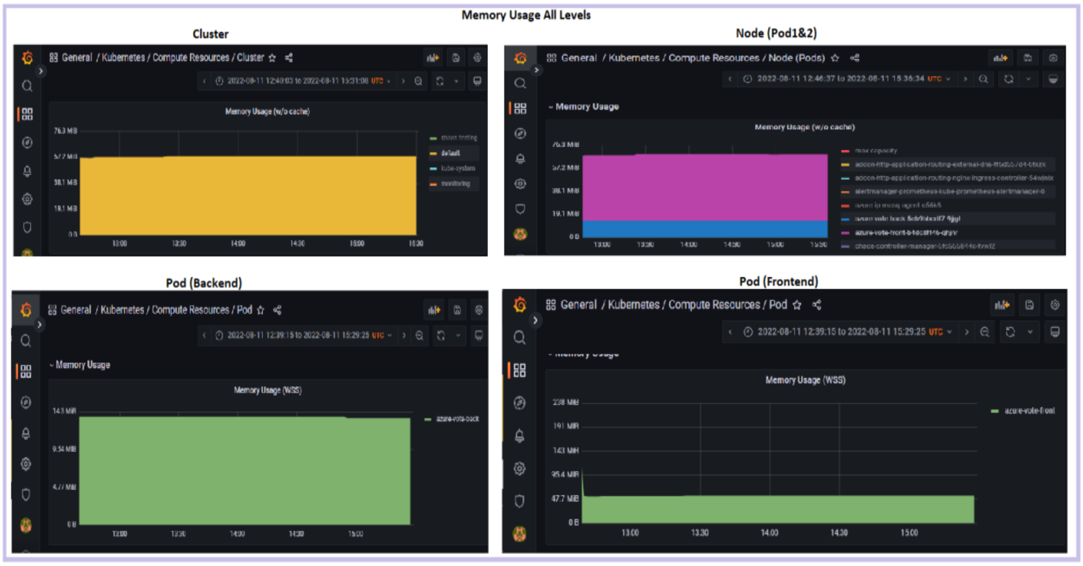

Memory Usage All Levels
Cluster
Node (Pod1&2)
Pod (Backend)
Pod (Frontend)
General / Kubernetes / Compute Resources / Cluster
General / Kubernetes / Compute Resources / Node (Pods)
General / Kubernetes / Compute Resources / Pod
Memory Usage (w/o cache)
Memory Usage (WSS)
azure-vote-back
azure-vote-front

Figure 33: Memory Usage for Pod Failure

Moving further with other types of Pod failure, called *Pod Kill*. Figures 34 and 35 are showing the consumption of CPU and Memory respectively while performing *pod kill* fault injection. These figures show that the consumption is completely different that pod failure. It can be observed that the utilization somehow follows a similar trend as in Stress failure mentioned in previous sections.

Logically, if the pod is killed, it should not more on the node and the consumption would not be applicable however the reality is just the opposite.

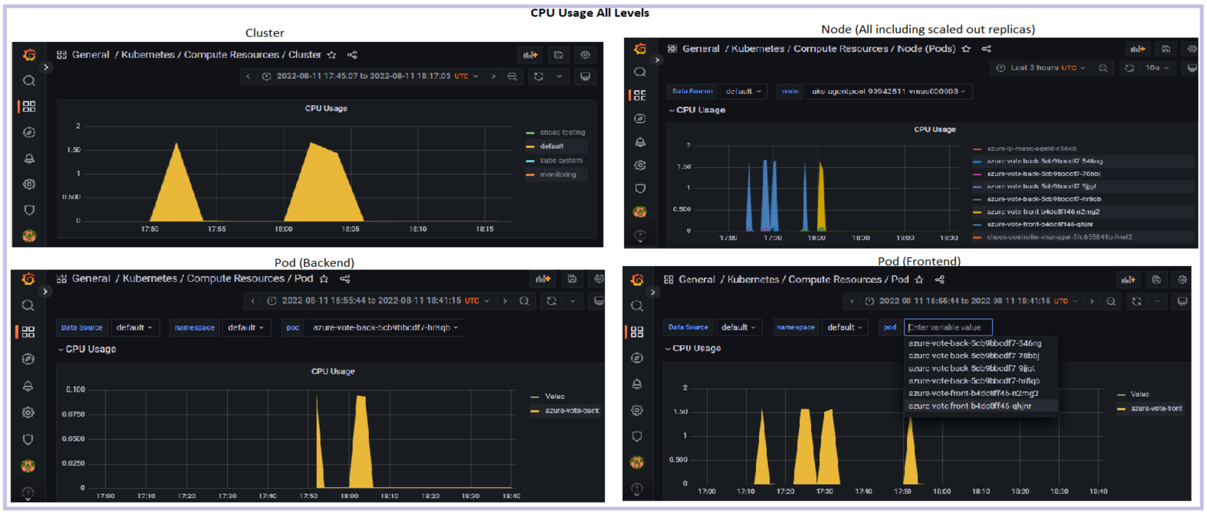

CPU Usage All Levels
Cluster
Node (All including scaled out replicas)
Pod (Backend)
Pod (Frontend)
General / Kubernetes / Compute Resources / Cluster
General / Kubernetes / Compute Resources / Node (Pods)
General / Kubernetes / Compute Resources / Pod
CPU Usage
chaos-testing
default
kube-system
monitoring
Value
azure-vote-back
azure-vote-front

Figure 34: CPU Usage for Pod Kill

Once the pod is killed, due to autoscaling of pods, it tries to recreate/restart and the resource-hungry pod start consuming the resource that is the reason, it is showing the valley in CPU utilization. Similar applied to memory consumption as shown in the below figure

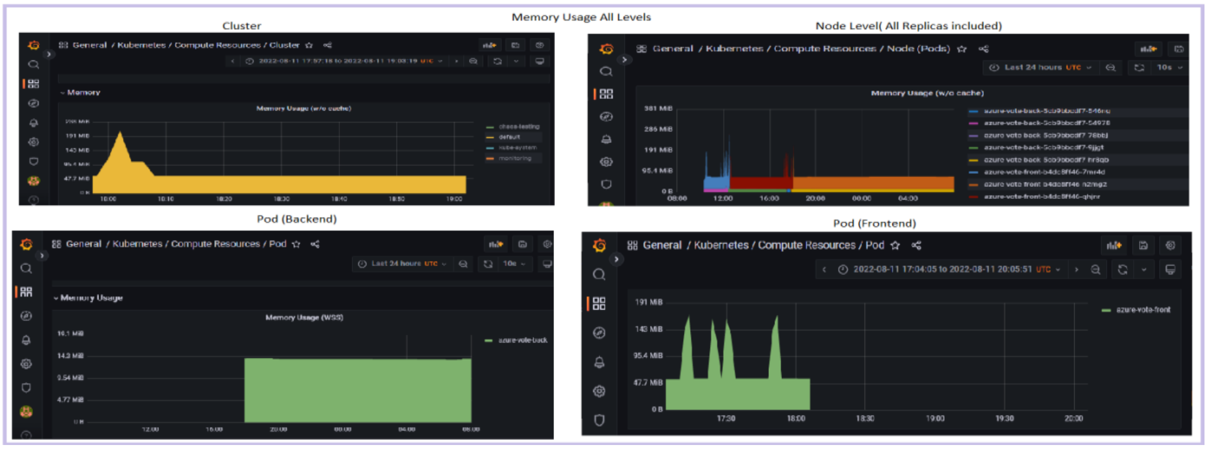


Figure 35: Memory Usage Pod Kill

The concluding remark of all these consumptions is that the Horizontal Pod Autoscaling properties of the Kubernetes ecosystem make it scale horizontally to avoid failure and make the application highly available. That is what was observed when Pod failure or Pod Killed injected intentionally, the cluster tried to auto scale, which leads to more consumption.

All these metrics were collected and exported as CSV to a local machine for further analysis to get the outlier when two layers were compared to identify the impact of these types of faults. The next chapter is going to describe it all in detail.

CHAPTER 4

RESULTS AND ANALYSIS

The observations of the outcomes and the full study effort are presented and covered in depth in this chapter. The first section covers the in-depth examination of the experimental data. The research activity is critically evaluated in the second section, which attempts to determine whether the goals that were established at the outset of the study endeavor have been achieved. Additionally, it describes the constraints that existed during the experiments. Finally, in the final section, discussion on the effects this study has had on the contemporary workforce.

ANALYSIS OF RESULTS

In this section, the metrics collected will be examined to attempt and determine any patterns that would indicate which layer is most affected. *K-Means* clustering algorithm, a vector quantization technique, is used for this. One thing to keep in mind is that before training a model, it must be identified that the whole data set going to be divided into a subgroup, know the number of groups (K). Later identify data points that are occasionally outside of their particular cluster as outliers or anomalies. The intriguing aspect about this is that it should be able to define the outliers on its own. Typically, an outlier/anomaly is defined as a data point that is far from the centroid (center point) of its cluster. We may also specify the number of data points that constitute an outlier [25]. For both types of faults inserted, two distinct layers are contrasted with one another. Data cleaning was carried out manually by retaining metrics' values in the same CSV file so that they could be compared and evaluated if any outlier values were detected. The expected outcome is that fault injection will be significantly determined by the number of outliers at any layer comparison.

Figure 36 on the next page, is showing the steps to achieve the task to find the outlier in the divided data set. First thing is to load the cleansed data to the data frame and standardize the variable to have a mean of 0 and a deviation of 1. This step is significant as the variables are distributed with wide ranges and this will have a huge impact on distance calculations. Secondly, data visualization steps are taken, which will be helpful to find the appropriate *K* value. At this point, it is important to know that if the analysis is done with more than two features, then Principal Component Analysis (PCA) method should be considered to find the *K* however in this experiment, the comparison is between two parametric values, hence two features, directly drawing the plot to find the *K*[25]. Further steps are clear once get the value of K, just fit all the variables. Once the model fit is done, again the cluster plot is drawn which will show data points and the cluster it belongs to. Black dots are used to show the centroid of the cluster. These steps visually show that there might be some data points that are far away from the centroid, that can be considered outliers. And this experiment will see the more outliers in comparison, the most impact layer. Once it is done the final step is to calculate the distance of each data point from the centroid of the respective clusters.

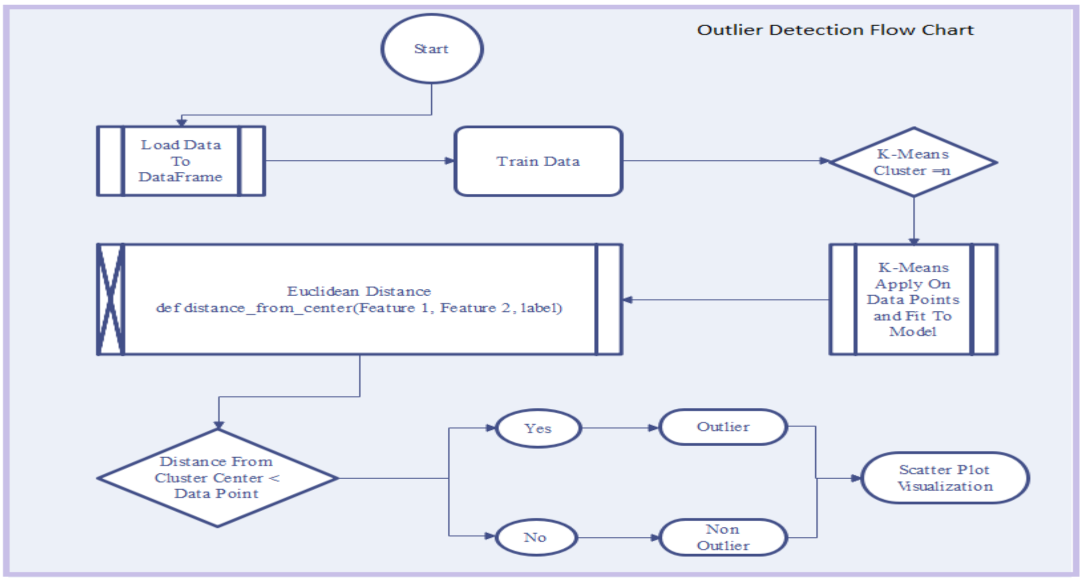

Outlier Detection Flow Chart
Start
Load Data To DataFrame
Train Data
K-Means Cluster =n
K-Means Apply On Data Points and Fit To Model
Euclidean Distance
def distance_from_center(Feature 1, Feature 2, label)
Distance From Cluster Center < Data Point
Yes
No
Outlier
Non Outlier
Scatter Plot Visualization

Figure 36: Outlier Detection Flow Chart

To calculate the distance of each data point, function *distance_from_center (Feature 1, Feature 2, label)* is used. There is a new column *label* defined as well which holds the results matching the clustering result [25]. Once the distance is calculated, it is stored in a variable called *distance.*

Euclidean distance Calculation formula varies how many data clusters are formed while using K-Means [30].

$$d(P,Q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

Finally, the data frame is sorted to find the data points most distant from the center. For more user-friendly, scatter plots are used to show visually.

Let's start with the evaluation of Stress Fault metrics collected in *chapter 3*. The analysis is categorized into two parts. First is stress fault for the resource *CPU and Memory*, secondly, the same resources are analyzed for Pod fault. Table 4 shows the resource type and corresponding layer comparison to find the maximum impacted layer. It illustrates that there were 3 layers of comparison performed for each, *CPU and Memory*.

Fault Type	**Resource Type**	**Layers**
Stress fault	CPU Usage	Cluster Vs Node
Stress Fault	CPU Usage	Node Vs Pod
Stress Fault	CPU Usage	Cluster Vs Pod
Stress Fault	Memory Usage	Cluster Vs Node
Stress Fault	Memory Usage	Cluster Vs Pod
Stress Fault	Memory Usage	Node Vs Pod

Table 4: Features Analysed for Stress Fault

The flow chart mentioned in Figure 36 had been followed to find the respective outliers while comparing two layers. Here to highlight, that since at Node and Pod layers, more than one service can appear as in this experiment, there were two services used, backend and frontend. Hence, individual services have taken a while compared with the cluster. The Node to Node or Pod to Pod comparison is always made sure that the same services are compared to get more meaningful results. Figure 37 depicts the outliers in different layers. It can be observed that the number of outliers is more wherever cluster and pod.

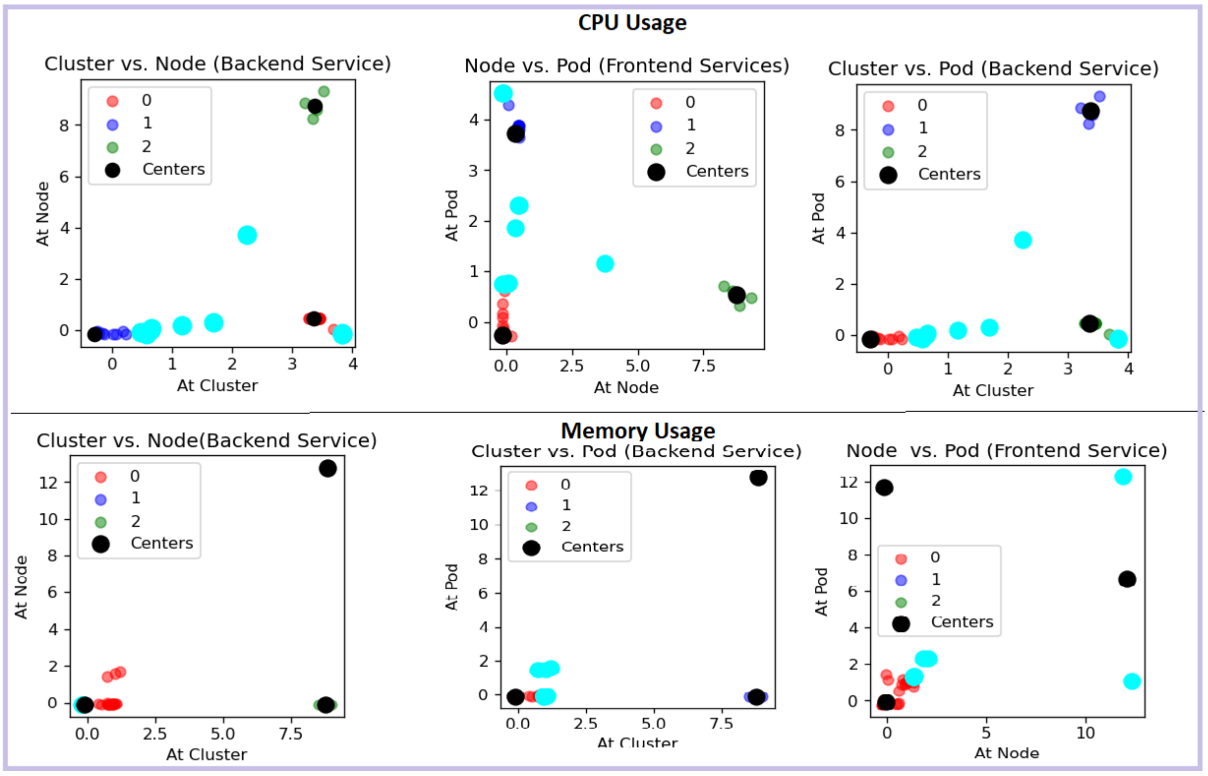


Figure 37: Impact Analysis (Stress Fault)

data compared. Cluster Vs Pod is also pointing to more outliers, it points that the cluster is pointing towards more outliers which can be the point of investigation however in the context of this research, it may be possible that the other services, like Prometheus, Grafana, and Chaos-mesh sharing the same hardware and that can be the cause of the higher *CPU* usage in this case however it is just an assumption at this point. In a conclusion for this, it can be said that the cluster is more impacted, and the investigation should start from there to see if the application in question is using more resources or if any other services running for different apps consuming.

Similarly, if we look for memory usage during stress fault, it can be observed that cluster vs node is showing the least outlier however node vs pod is showing maximum impacted area. In this case, the debugging should start from the application pod and then move to the node. There might be possible that the pod configuration is not appropriate as in this experiment there is no resource consumption limit set in the definition file mentioned in Figure 23. In this scenario, Pod may try to scale out and consume maximum resources while injecting the stress fault. Hence this information will be helpful to identify the configuration mistake and its repercussion.

Moving further for Pod fault analysis, Table 5 shows the resource type and the layers compared. There were two different types of faults injected as Pod fault mentioned in section experiment 3. Collected logs are used as per the steps mentioned earlier in this chapter.

Fault Type	Resource Type	Layers
Pod fault	CPU Usage	Cluster Vs Node
Pod fault	CPU Usage	Node Vs Pod
Pod fault	CPU Usage	Cluster Vs Pod
Pod fault	Memory Usage	Cluster Vs Node
Pod fault	Memory Usage	Node Vs Pod
Pod fault	Memory Usage	Cluster Vs Pod

Table 5: Features Analysed for Pod Fault

Figure 38 below is showing the outlier in different layers. Considering CPU usage in the mentioned figure, it can be seen that the Cluster Vs Node shows the least outliers however Node Vs Pod and Cluster Vs Pod show significant deviation. Cluster Vs Node shows the least outliers however Node Vs Pod and Cluster Vs Pod show significant deviation.

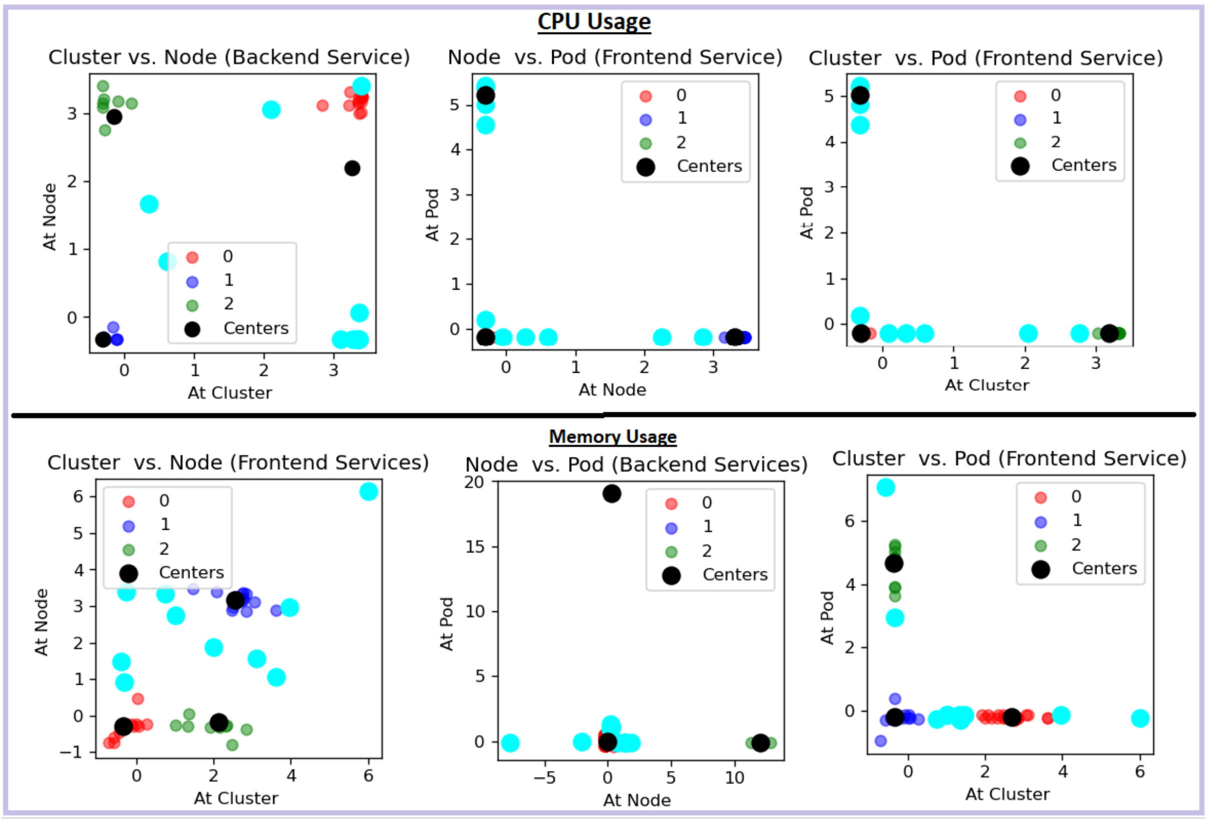


Figure 38: Impact Analysis (Pod Fault)

If further narrow down then it can be observed that node and Pod seem most impacted and especially wherever the Pod comparison involved showing more outliers, hence the debugging should involve checking at Pod level configuration and later node. Furthermore, in terms of memory usage, it clearly shows that the Node Vs Cluster shows most of the outlier, a possible reason may be the underlying hardware. Node Vs Pod in the above figure shows limited outliers hence if we find the relation, the cluster level seems more impacted. One of the obvious answers could be the other services running cumulatively as cluster it is showing most impacted.

In summary after comparing both types of faults, which somehow point to either cluster or pod configuration need to be reviewed.

Impact of The Research

This experimental research is proposing a direction to reduce the debugging time of cloud-based microservices in case of any failures. Also, it provides flexibility to test many more permutation combinations of parameters to predict more accurate results. These all will help in the real-world operation to find the impact in a certain area and resolve it quickly. Even though the scope and the experimental application were not as per the real-world application however this research put light on a new aspect of troubleshooting the cloud-based microservices which is a pain area the case when the application is complex, and the infrastructure is huge in terms of servers and connected in a different availability zone. Adopting the method followed in this research can improve the Turnaround Time (TAT) of the operations or administrative teams that are responsible for monitoring and maintaining the infrastructure. This will be a great benefit to an organization as cloud infra is prone to fault however for end users, it matters a lot which organization provides almost zero downtime to serve its customer and this methodology will be a great addition to this goal.

CHAPTER 5

CONCLUSION

CLOSING STATEMENT

In this experimental study, it was tried to address the reduction in debugging time when any issue was faced in the operation of a cloud-based microservice. Since maintaining and troubleshooting the enterprise-level microservice application is a challenging and cumbersome job to do. Many organizations are investing a lot in this area to make applications always ready and to have the least downtime. There are many monitoring tools and anomaly detection systems present in the market however these are used to mainly monitor hardware level consumption or predict the anomalies based on historical logs. In this study, tried to figure out the method through which we can narrow down the area of troubleshooting by using outlier detection based on the K-Means clustering algorithm.

This study started by studying related work in this area, the goal had been set to find the layered outliers. The activity started by creating the Kubernetes cluster on a public cloud and choosing Azure Kubernetes Cluster (AKS), which provides managed control plan. Furthermore, the monitoring tools, Prometheus and Grafana had been integrated to monitor the cluster along with different dashboards provided by Grafana. Later, to emulate the user load and run-time fault injection, Chaos-Mesh and JMeters had been used. Using JMeter, user threads spawned to the target cluster and at the same time, the run time different fault injected (Stress and Pod Fault) using Chaos-Mesh to mimic the real-time environment. After these experimental runs, the metrics (CPU, Memory, and HTTP Response) were collected in CSV format and exported to the local machine for analysis.

Conclusively, several metrics were observed, and then the K-Means clustering algorithm was used to find the outliers. The data set is divided into 3 subsets and the Euclidean distance is calculated from the centroid of the cluster. Data points with high distance values are considered outliers and considered as problematic points. Finally, by referring to figures 37 and 38, it was concluded the method can identify the most impact layer and can give direction to troubleshoot the issue further. The proposed method that can be adopted is mentioned in figure 36 once the relevant metrics are collected.

FUTURE WORK

This study fulfils most of the expected tasks mentioned at the start however there are a few areas where the scope is limited in this study.

Firstly, in this study, the microservice application was having only two services, more services may create the scenario more complex while monitoring and analyzing the data. Also, if the service with persistent volume is introduced then the complexity would be more.

Secondly, this study focused only on CPU and Memory utilization even though these are the primary metrics that can cause severe impact, still few more parameter additions can give a better-pinpointed layer of impact and that can lead to a great reduction in troubleshooting time.

Finally, lots of manual work is involved and these can be automated using Terraform to deploy and integrate the different tools and for analysis of the data, unsupervised multivariate algorithms can be explored.

BIBLIOGRAPHY

[1] De, S. (2021). A Study on Chaos Engineering for improving Cloud Software Quality and Reliability. [online] IEEE Xplore. doi:10.1109/CENTCON52345.2021.9688292.
[2] Noor, A., Jha, D.N., Mitra, K., Jayaraman, P.P., Souza, A., Ranjan, R. and Dustdar, S. (2019). A Framework for Monitoring Microservice-Oriented Cloud Applications in Heterogeneous Virtualization Environments. 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). doi:10.1109/cloud.2019.00035.
[3] Torkura, K.A., Sukmana, M.I.H., Cheng, F. and Meinel, C. (2020). CloudStrike: Chaos Engineering for Security and Resiliency in Cloud Infrastructure. IEEE Access, 8, pp.123044–123060. doi:10.1109/access.2020.3007338.
[4] Jiang, Y., Zhang, N. and Ren, Z. (2020). Research on Intelligent Monitoring Scheme for Microservice Application Systems. 2020 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS). doi:10.1109/icitbs49701.2020.00173.
[5] S. De, "A Study on Chaos Engineering for improving Cloud Software Quality and Reliability," 2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON), 2021, pp. 289-294, doi: 10.1109/CENTCON52345.2021.9688292.
[6] Shkuro, Y. (2020). Observability challenges in microservices and cloud-native applications. [online] Medium. Available at: https://medium.com/@YuriShkuro/observability-challenges-in-microservices-and-cloud-native-applications-72857f9d03af [Accessed 17 Jul. 2022].
[7] Gunja, S. (2021). What is chaos engineering? [online] Dynatrace news. Available at: https://www.dynatrace.com/news/blog/what-is-chaos-engineering/ [Accessed 17 Jul. 2022].
[8] The New Stack. (2020). Debugging Microservices in the Cloud. [online] Available at: https://thenewstack.io/debugging-microservices-in-the-cloud/ [Accessed 17 Jul. 2022].
[9] λ.eranga (2021). Chaos Engineering with Chaos Mesh. [online] Rahasak Labs. Available at: https://medium.com/rahasak/chaos-engineering-with-chaos-mesh-b040169b51bd [Accessed 17 Jul. 2022].

[10] Nane. Kratzke. About microservices, containers and their underestimated impact on network performance. arXiv preprint arXiv:1710.04049(2017)., 2017.

[11] Cotroneo, D., De Simone, L., Liguori, P. and Natella, R. (2022). Fault Injection Analytics: A Novel Approach to Discover Failure Modes in Cloud-Computing Systems. IEEE Transactions on Dependable and Secure Computing, [online] 19(3), pp.1476–1491. doi:10.1109/tdsc.2020.3025289.

[12] Amaral, Marcelo & Polo, Jorda& Carrera, David &Mohomed, Iqbal &Unuvar, Merve&Steinder, Malgorzata. (2015). Performance Evaluation of Microservices Architectures Using Containers. 27-34. 10.1109/NCA.2015.49.

[13] Sauvanaud, C., Kaâniche, M., Kanoun, K., Lazri, K. and Da Silva Silvestre, G. (2018). Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned. Journal of Systems and Software, 139, pp.84–106. doi:10.1016/j.jss.2018.01.039.

[14] Zhang, Y., Zheng, Z. and Lyu, M.R. (2014). An Online Performance Prediction Framework for Service-Oriented Systems. IEEE Transactions on Systems, Man, and Cybernetics: Systems, [online] 44(9), pp.1169–1181. doi:10.1109/TSMC.2013.2297401.

[15] Mi, Haibo& Wang, Huaimin& Zhou, Yangfan&Lyu, Michael & Cai, Hua. (2013). Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems. Parallel and Distributed Systems, IEEE Transactions on. 24. 1245-1255. 10.1109/TPDS.2013.21.

[16] Zhang, Y., Jiang, W. and Deng, X. (2019). Fault diagnosis method based on time domain weighted data aggregation and information fusion. International Journal of Distributed Sensor Networks, 15(9), p.155014771987562. doi:10.1177/1550147719875629.

[17] Du, Q., Xie, T. and He, Y. (2018). Anomaly Detection and Diagnosis for Container-Based Microservices with Performance Monitoring. Algorithms and Architectures for Parallel Processing, pp.560–572. doi:10.1007/978-3-030-05063-4_42.

[18] mlearned (2019). Concepts - Kubernetes basics for Azure Kubernetes Services (AKS) - Azure Kubernetes Service. [online] Microsoft.com. Available at: https://docs.microsoft.com/en-us/azure/aks/concepts-clusters-workloads.

[19] chaos-mesh.org. (n.d.). Chaos Mesh Overview | Chaos Mesh. [online] Available at: https://chaos-mesh.org/docs/ [Accessed 19 Jul. 2022].

[20] chaos-mesh.org. (2021). Chaos Mesh Remake: One Step Closer toward Chaos as a Service | Chaos Mesh. [online] Available at: https://chaos-mesh.org/blog/chaos-mesh-remake-one-step-closer-towards-chaos-as-a-service/ [Accessed 19 Jul. 2022].

[21] Wikipedia. (2022). t-statistic. [online] Available at: https://en.wikipedia.org/wiki/T-statistic#:~:text=The%20t%2Dstatistic%20is%20used [Accessed 23 Jul. 2022].

[22] www.gremlin.com. (n.d.). Chaos Engineering: the history, principles, and practice. [online] Available at: https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-and-practice/.

[23] Belgium, O. (n.d.). An introduction into the world of Chaos Engineering - Tim Verte. [online] ordina-jworks.github.io. Available at: https://ordina-jworks.github.io/cloud/2020/12/14/chaos-engineering.html [Accessed 23 Jul. 2022].

[24] Kubernetes. (n.d.). Resource Management for Pods and Containers. [online] Available at: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.

[25] Dino, L. (2022). Outlier Detection Using K-means Clustering In Python. [online] Medium. Available at: https://towardsdev.com/outlier-detection-using-k-means-clustering-in-python-214188fc90e8 [Accessed 13 Aug. 2022].

[26] GitHub. (2022). Azure Voting App. [online] Available at: https://github.com/Azure-Samples/azure-voting-app-redis/blob/master/azure-vote-all-in-one-redis.yaml [Accessed 15 Aug. 2022].

[27] Tien, C., Huang, T., Tien, C., Huang, T. and Kuo, S. (2019). KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches. Engineering Reports, 1(5). doi:10.1002/eng2.12080.

[28] Agrawal, Bikash &Wiktorski, Tomasz & Rong, Chunming. (2017). Adaptive real-time anomaly detection in cloud infrastructures. Concurrency and Computation: Practice and Experience. 29. e4193. 10.1002/cpe.4193.

[29] GitHub. (2022). Azure Voting App. [online] Available at: https://github.com/Azure-Samples/azure-voting-app-redis/blob/master/azure-vote/azure-vote/main.py [Accessed 20 Aug. 2022].

[30] itsmycode.com. (2022). *Calculate Euclidean Distance in Python - ItsMyCode*. [online] Available at: https://itsmycode.com/calculate-euclidean-distance-in-python/#:~:text=irrespective%20of%20dimensions.- [Accessed 15 Aug. 2022].

[31]GrafanaLabs.(n.d.). Dashboards. [online]Available at: https://grafana.com/grafana/dashboards/ [Accessed 21 Aug. 2022].

[32] Pandas (2018). Python Data Analysis Library — pandas: Python Data Analysis Library. [online] Pydata.org. Available at: https://pandas.pydata.org/.

[33] Matplotlib (2012). Matplotlib: Python plotting — Matplotlib 3.1.1 documentation. [online] Matplotlib.org. Available at: https://matplotlib.org/.

[34] seaborn (2012). seaborn: statistical data visualization — seaborn 0.9.0 documentation. [online] Pydata.org. Available at: https://seaborn.pydata.org/.

[35] Scikit-learn.org. (2010). 2.3. Clustering — scikit-learn 0.20.3 documentation. [online] Available at: https://scikit-learn.org/stable/modules/clustering.html#k-means.

[36] Jupyter (2019). Project Jupyter. [online] Jupyter.org. Available at: https://jupyter.org/.

[37] Apache Software Foundation (2019). Apache JMeter - Apache JMeterTM. [online] Apache.org. Available at: https://jmeter.apache.org/.

[38] Prometheus (n.d.). Prometheus - Monitoring system & time series database. [online] prometheus.io. Available at: https://prometheus.io/.

[39] zr-msft (n.d.). Quickstart: Deploy an AKS cluster by using the Azure portal - Azure Kubernetes Service. [online] docs.microsoft.com. Available at: https://docs.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-portal?tabs=azure-cli [Accessed 22 Aug. 2022].

[40] TECHCOMMUNITY.MICROSOFT.COM. (2021). Using Azure Kubernetes Service with Grafana and Prometheus. [online] Available at: https://techcommunity.microsoft.com/t5/apps-on-azure-blog/using-azure-kubernetes-service-with-grafana-and-prometheus/ba-p/3020459 [Accessed 22 Aug. 2022].
[41] chaos-mesh.org. (n.d.). Install Chaos Mesh using Helm (Recommended for Production Environments) | Chaos Mesh. [online] Available at: https://chaos-mesh.org/docs/production-installation-using-helm/ [Accessed 22 Aug. 2022].

APPENDIX A

Code And Kubernetes Manifest Files

A.1 Git main repository

Kumar, S. (2022). *Msc_Thesis_Repo_AllDocs*. [online] GitHub. Available at: https://github.com/sk-85/Msc_Thesis_Repo_AllDocs [Accessed 22 Aug. 2022].