GRADUATE PROGRAMS INSTITUTE

A BENCHMARKING FRAMEWORK FOR EVALUATING CLOUD SERVICES: A CASE STUDY ON AZURE APP SERVICE VS DOCKER CONTAINER

Manar ALKULL

MASTER'S THESIS

İstanbul, May 2025

A BENCHMARKING FRAMEWORK FOR EVALUATING CLOUD SERVICES: A CASE STUDY ON AZURE APP SERVICE VS DOCKER CONTAINER

Manar ALKULL

MASTER'S THESIS

Institute of Graduate Programs

COMPUTER ENGINEERING MASTER'S PROGRAM WITH THESIS

MASTER'S THESIS ADVISOR

Prof. Ercan SOLAK

BEYKOZ ÜNİVERSİTESİ

T.C.
BEYKOZ ÜNİVERSİTESİ
LİSANSÜSTÜ PROGRAMLAR ENSTİTÜSÜ
MÜDÜRLÜĞÜ

18/06/2025

Yüksek Lisans Tez Onay Belgesi

Enstitümüz Bilgisayar Mühendisliği Ana bilim Dalı, Tezli Yüksek Lisans Programı 2330140039 numaralı öğrencisi Manar Alkull 'un **"A BENCHMARKING FRAMEWORK FOR EVALUATIMG CLOUD SERVICES: A CASE STUDY ON AZURE APP SERVICES VS DOCKER CONTAINER"** adlı tez çalışması Enstitümüz Yönetim Kurulunun ../../.... tarih ve 20../.. sayılı kararıyla oluşturulan jüri tarafından oy birliği ile Tezli Yüksek Lisans tezi olarak kabul edilmiştir.

Öğretim Üyesi Adı Soyadı	**İmzası**
Tez Savunma Tarihi :18/06/2025	
1)Tez Danışmanı: Prof. Dr. Ercan Solak	…………………
2) Jüri Üyesi : Prof. Dr. Taner Eskil	...……………….
3) Jüri Üyesi : Dr. Öğr. Üyesi Feyza Erkan	………………….

Not: Öğrencinin Tez savunmasında **Başarılı** olması halinde bu form **imzalanacaktır**. Aksi halde geçersizdir.

PREFACE

ACKNOWLEDGMENT

I would like to express my sincere love and deepest appreciation to my beloved ones, whose unlimited support and affection were a constant source of strength throughout my journey.

To my father, Muhammed Mahdi Alkull, and my mother, Raida Alshami, your unconditional love and boundless support have been the foundation of everything I have achieved.

To my beloved wife, Sarah Alrefai, your presence has filled my life with endless love, warmth, and kindness.

To my precious brothers, Fatih Gul and Omar Alkull, I wish you both all the success and happiness life has to offer.

To all of you, I say: Thank you. Thank you for always being there for me, for your encouragement, and for all the beauty you've brought into my life.

I also extend my sincere gratitude to my advisor, Prof. Ercan Solak, for his continuous support and invaluable guidance throughout my studies. I deeply appreciate the time, effort, and dedication he invested in helping me achieve my academic goals.

Sincerely,

Manar Alkull

Contents

PREFACE i

Contents ii

ABSTRACT iv

ABSTRACT v

ABBREVIATIONS vi

LIST OF FIGURES vii

1. INTRODUCTION 1

2. LITERATURE SURVEY 3

2.1. Azure App Service 3

2.2. Docker 4

2.3. Benchmarking Suites 6

2.4. Apache JMeter 7

3. Cloud Benchmarking Framework 11

3.1. Characteristics of the proposed framework 11

3.1.1. Comparability 12

3.1.2. Coverage 15

3.1.3. Configurability 16

3.1.4. Expandability 17

3.1.5. Comprehensive And Clear Output 17

3.2. Implementation 19

3.2.1. Client-Server architecture 19

3.2.2. The Load Service (server side) 20

3.2.3. Client app 22
3.3. Case Study: Azure App Service Vs Docker Container Evaluation 22
3.3.1. Azure App Service 23
3.3.2. Azure Docker Container 23
3.3.3. Client App 24
4. Findings and Results 24
4.1. File Read 24
4.2. File Write 25
4.2.1. File Write Stress Test 27
4.3. Memory Read 29
4.4. Memory Write 30
4.4.1. Memory Stress Test 31
4.5. CPU Load 32
4.5.1. CPU Stress Test 34
4.6. Network stress 35
Conclusion 38
4.7. Future Steps 38
REFERENCES 40
APPENDICES 43
4.8. Appendix A: Data Blocks 43
4.8.1. Appendix A.1: example of cloud benchmarking client's configuration 43
4.8.2. Appendix A.2: Full schema of cloud benchmarking client configuration 48
BIOGRAPHY ..... **Error! Bookmark not defined.**

ABSTRACT

Bulut Hizmetlerini Değerlendirmek İçin Standart Bir Çerçeve: Azure Uygulama Hizmeti ile DOCKER Konteynerinin Karşılaştırılmasına İlişkin

Bulut bilişimde iki veya daha fazla hizmeti karşılaştırmak ve hangisinin "daha iyi" olduğuna karar vermek, birçok nedenden ötürü zorlu bir iştir. Mevcut kıyaslama çözümleri, bulutta test süreçlerini yürütme konusunda yeterli verimliliğe sahip değildir. Ayrıca, özel test durumları için yapılandırılmaları ve ortaya çıkan verilerin yapısının belirlenmesi zordur. Bulut ortamının dağıtılmış yapı, kaynak kapasitesindeki çeşitlilik, sanallaştırma, güvenlik ve maliyet gibi kendine özgü özellikleri, onu birçok yeni çalışma için ilgi çekici bir konu haline getirmiştir. Bu araştırma, bulut kıyaslama çerçevesinin karşılaştırılabilirlik, kapsam, yapılandırılabilirlik, esneklik ve kapsamlılık gibi özelliklere sahip olması gerektiğini öne sürmektedir. Bu özellikler, bulut ortamında en iyi çıktıları elde etmek amacıyla özelleştirilmiş bir kıyaslama çerçevesi önerilirken bu araştırmada dikkate alınmıştır. Önerilen çerçeve, yükü konu düğümünde çalıştıran bir hizmet olarak sunucuya ve testi kontrol etmekten ve çıktı verilerini oluşturmaktan sorumlu olan istemciye sahip istemci-sunucu modelinde tasarlanmıştır. Önerilen çerçevenin geçerliliğini test etmek için bir demo uygulaması geliştirilmiş ve Azure App Service'in performansını Azure Docker Container ile karşılaştırmak için kullanılmıştır. Test, konu düğümlerindeki (App Service ve Docker konteyneri) farklı kaynakları (CPU, Bellek, Dosya G/Ç ve ağ dahil) kıyaslamak üzere gerçekleştirilmiştir. Elde edilen veriler, Docker Konteynerinin aynı bulut katmanı için Azure'da App Service'ten genel olarak daha iyi performansa sahip olduğunu ortaya koymuştur.

Keywords: Bulut bilişim, Kıyaslama, Karşılaştırma, Hizmet performansı, Verimlilik, Yapılandırılabilirlik, Özelleştirilmiş kıyaslama çerçevesi.

Tarih: Mayıs 2025

ABSTRACT

A BENCHMARKING FRAMEWORK FOR EVALUATING CLOUD SERVICES: A CASE STUDY ON AZURE APP SERVICE VS DOCKER CONTAINER

For many reasons, comparing two or more services on the cloud and deciding which "is better" is a challenging job. The current benchmarking solutions lack efficiency when it comes to running the test process on the cloud, and they are difficult to configure for tailored test cases and specifying the structure of the resulted data. The cloud environment's special characteristics (like distributed nature, variation in resources' capacity, virtualization, security, and cost) made it an interesting subject for many recent studies. This research suggests that the cloud benchmarking framework needs to have cloud-benchmarking-related features like comparability, coverage, configurability, plasticity, and comprehensiveness. These characteristics are considered in this research while proposing a benchmarking framework that is customized to have its best outputs in the cloud environment. It's designed in client-server model where the server is a service that runs the load on the subject node, and the client is responsible for controlling the test and generating the output data. A demo application is implemented to test the validity of the proposed framework, and it is used to compare the performance of Azure App Service with Azure Docker Container. The test is conducted to benchmark different resources -including CPU, Memory, File I/O, and network- on the subject nodes (App Service and Docker container), and the resulting data revealed that Docker Container has better performance -in general- than App Service on Azure for the same cloud tier.

Keywords: Cloud Benchmarking, Performance Evaluation, Azure App Service, Docker Container, Cloud Computing

Date: May 2025

ABBREVIATIONS

AWS : Amazon Web Service

DevOps: Development Operations

SLAs : service level agreements

PaaS : Platform as a Service

IaaS : Infrastructure as a Service

API : Application Programming Interface

CLI : Command Line Interface

VM : Virtual Machine

VNet : Virtual Network

VPN : Virtual Private Network

LXC : Linux Containers

CPU : Central Processing Unit

SOAP : Simple Object Access Protocol

JDBC : Java Database Connector

CI/CD : Continuous Integration and Continuous Delivery

GUI : Graphical User Interface

IaC : Infrastructure as Code

VCS : Version Control Systems

File I/O: File Input/Output

OAuth : Open Authorization

BI : Business Intelligence

DB CRUD: Database Create, Read, Update And Delete

LIST OF FIGURES

Figure 2.1 General structure of Docker system 5
Figure 2.2 JMeter GUI 8
Figure 2.3 Response Time Percentiles in JMeter Report 9
Figure 2.4 Time Vs Threads in JMeter report 9
Figure 3.1 Client-server architecture for the proposed cloud benchmarking system 19
Figure 3.2 Cloud Architecture for Azure App Service Vs Docker Container Benchmarking 23
Figure 4.1 File read performance comparision between App service and docker container in Azure 25
Figure 4.2 File write performance comparison between App service and docker container in Azure 27
Figure 4.3 File write time per data size 28
Figure 4.4 Memory read performance comparison between App Service and docker container in Azure 29
Figure 4.5 Memory-write performance comparison between App Service and docker container in Azure 31
Figure 4.6 Memory write time per data size 32
Figure 4.7 CPU load performance comparison between App Service and docker container in Azure for single vs 20 users 34
Figure 4.8 CPU processing time per threads count 35
Figure 4.9 network stress test on app service vs container for data size 1MB vs 10MB 37
Figure 4.10 time series response time for network stress test on app service vs container for data size 1MB vs 10MB 37

1. INTRODUCTION

Over the past twenty years, the software industry has become increasingly dependent on cloud environments. This shift began when Amazon launched AWS (Amazon Web Services) in 2006, followed by Google App Engine and Microsoft Azure between 2008 and 2010 [1]. Since then, many companies started moving away from traditional on-premises servers in favor of cloud solutions, mainly because of benefits like cost savings, better scalability, and more flexibility. By the 2020s, cloud computing had become the go-to platform for software development, especially with the rise of DevOps practices, microservices, and remote work setups. Key reasons behind this shift include lower initial investment (since there's no need for physical hardware), the ability to scale up or down easily, pay-as-you-go pricing models, global accessibility, and faster product deployment.

Despite its advantages, cloud computing still comes with some challenges. Security and privacy are top concerns, as data is handled by third-party providers, which can lead to risks like data breaches or unauthorized access. Another issue is the management of service level agreements (SLAs), where vague terms might cause misunderstandings about responsibilities or expected performance. Also, moving to the cloud means organizations give up some level of control over their data, which can lead to difficulties in data governance and staying compliant with regulations. Because of these risks, careful planning and a strong level of trust in cloud providers are necessary to make sure services remain secure and reliable [2].

The variety of cloud services and cloud providers made it rather challenging to decide which cloud solution to adopt for different scenarios. To tackle this issue, benchmarking tools are used to measure and compare the services in different aspects like security, performance, configurability, reliability, and more. Most benchmarking tools were developed during or before the rise of cloud computing, which made most of them incompatible with this environment. However, some of the existing benchmarking solutions (like JMeter) are built for general purposes and have enough flexibility to be

configured to run in new environments. Then again, the cloud environment has special features and characteristics that make benchmarking services on the cloud difficult. After a literature review with analysis of current solutions, an experimental research method is conducted in this study to reveal the difficulties and obstacles that face the cloud benchmarking process and provide a suitable solution for this issue. This research will answer the questions regarding the specifications of good cloud benchmarking utilities and the methodologies and aspects to consider in these tools. Furthermore, a practical solution is offered with the proof of validity by implementing use case on Azure cloud services with revealing and analyzing the results, and proposing recommendations.

2. LITERATURE SURVEY

2.1. Azure App Service

Azure App Service one of the key platforms for deploying web applications in the cloud. It is a fully managed Platform as a Service (PaaS) offered by Microsoft, which means it handles most of the infrastructure setup and maintenance for the developer. It's one of the most widely used options for hosting web apps, APIs, and backend services without needing to manage virtual machines or networking manually.

Many sources describe Azure App Service as ideal for developers who want to focus on their code rather than the underlying infrastructure. It supports multiple programming languages like .NET, Node.js, Python, and Java, which makes it quite flexible. What stands out is how it integrates tightly with other Azure services, such as Azure SQL Database, Azure Storage, and Azure DevOps, allowing for smooth deployment pipelines and scalability. Most documentation and studies emphasize its auto-scaling features, built-in security (like HTTPS and authentication), and ease of deployment via GitHub or Azure CLI. However, some researchers point out that while it simplifies deployment, it also limits low-level control over the infrastructure, which may not suit highly customized or performance-critical systems [1].

In performance prospect, Azure App Service is known for predictable behavior under moderate load, but some studies mention that cold start delays or regional differences can affect response times. It also offers different pricing tiers, and the performance varies significantly depending on the tier selected [3].

Azure App Service runs on top of Azure's virtual machine scale sets, but this is completely managed by Microsoft. The developer doesn't interact directly with VMs, containers, or load balancers. Instead, applications are deployed into an App Service Plan, which defines the region, pricing tier (e.g., Basic, Standard, Premium, Isolated), and resource allocation like CPU cores, memory, and storage. Each App Service Plan runs on a dedicated pool of compute resources (except in the Free/Shared tiers), and

scaling is handled through either manual or automatic scaling configurations. In the Premium and Isolated tiers, App Service also provides VNet integration, private endpoints, and access to dedicated instances, which is critical for enterprises requiring network isolation or hybrid connectivity.

2.2. Docker

Docker originated as an internal project under the name dotCloud, eventually transitioning to an open-source platform in 2013. Developed using the Go programming language, it was initially built upon Linux Containers (LXC) technology. Over time, however, Docker moved beyond LXC by creating its own container runtime environment and introducing a user-friendly, high-level API, which enhanced container management and usability.

Rather than emulating entire operating systems like traditional virtual machines, Docker is designed to isolate individual applications along with their supporting processes. Docker containers do not operate with a dedicated operating system; instead, they rely on the host system's kernel and infrastructure. This design results in a more efficient and lightweight execution environment. The primary concept behind Docker is to encapsulate an application with all necessary components -such as dependencies, libraries, and configuration files- into a portable container that can run consistently across any Linux-based host that supports Docker.

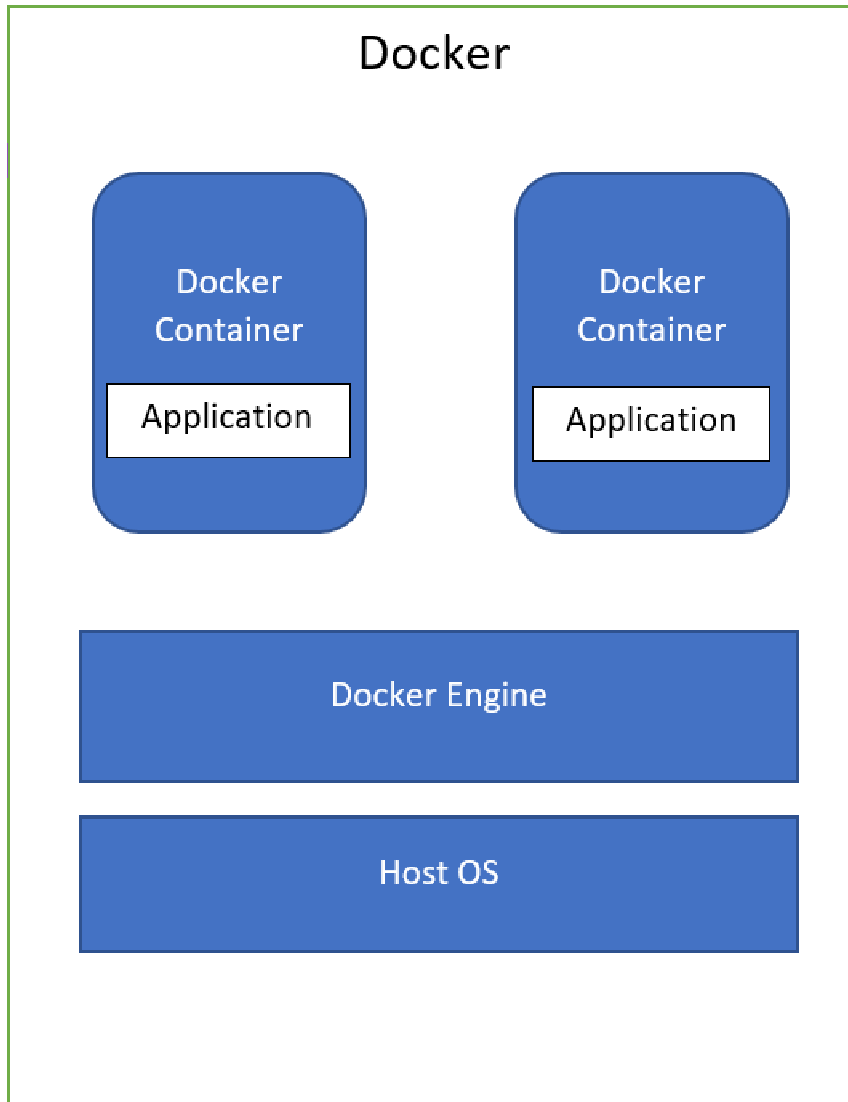

Figure 2.1 General structure of Docker system

Docker offers several important features that make it attractive for developers. One of these is cross-platform portability, which allows containers to run across different systems without modification. Another key strength is version control (similar to Git), Docker enables users to save, track, and revert container versions, as well as compare changes between them. Reusability is also an advantage: developers can use base images to create multiple container instances instead of building each one from scratch. For example, a single image of a database can be reused to launch several identical

containers. Additionally, Docker Hub provides a wide range of publicly shared container images, encouraging collaboration and reuse within the developer community.

Over time, Docker replaced its initial reliance on LXC with a custom runtime called Libcontainer, which was later succeeded by containerd, the current default in modern Docker. Docker also introduced a new model where each container runs a single application, in contrast to older systems like LXC that allowed multiple applications or full operating systems in one container. This design encourages modular application development, where large software systems are broken down into smaller, focused services. Containers are launched and managed using the Docker Engine, which executes container images created using Dockerfiles. These files automate the container-building process and help ensure consistent deployment. Despite its benefits in terms of speed, flexibility, and ease of deployment, Docker does come with some challenges. Moving existing complex applications into containers can be difficult, especially when transitioning from traditional hosting environments. However, once properly configured, Docker simplifies the development workflow and makes applications easier to distribute and scale. [2].

2.3. Benchmarking Suites

A variety of benchmarking tools have been developed to evaluate the performance of cloud systems. One well-known example is CloudSuite, which offers a collection of both batch and real-time workloads -such as memcached- and is commonly used to assess cloud behavior at the hardware level. Another tool, TailBench, brings together a set of benchmarks for services like web servers, databases, speech-to-text processing, and machine translation, while also proposing improved methods for performance measurement. Sirius focuses specifically on workloads found in intelligent assistants, such as speech recognition, to assess how efficiently machine learning tasks can be executed.

Despite their usefulness, many of these tools are based on relatively simple architectures, typically involving single-tier applications or systems with just a few layers. This doesn't accurately represent the complexity of today's cloud-native applications. Even in cases where tools include multi-tier workloads (like search engines), they often treat components as isolated services, overlooking the interdependencies that are common in real-world cloud deployments. To address this gap, more recent benchmarking frameworks have emerged with a focus on microservices architectures. For instance, µSuite examines the interactions between microservices and the underlying operating system, tracking metrics such as system calls and context switches. Some research also investigates how various factors including CPU resource limits, development frameworks, and container configurations that affect performance and scalability.

One of the more advanced benchmarking tools, DeathstarBench, is designed for large-scale systems consisting of numerous interconnected microservices. This framework enables the exploration of challenges that arise only at scale, such as network bottlenecks and service-dependency failures. It supports a diverse range of realistic application scenarios, including social networking platforms, e-commerce systems, and edge computing solutions [4].

2.4. Apache JMeter

Apache JMeter is a widely used open-source tool for performance and load testing. JMeter was originally developed for testing web applications but has grown to support many protocols, including HTTP, REST APIs, SOAP, JDBC, and others. Its popularity comes from its flexibility, ease of use, and active community. JMeter allows testers to simulate multiple users sending requests to a system, helping to identify issues such as slow response times, bottlenecks, or failures under heavy load. It uses a thread-based model, where each thread simulates a virtual user. These threads can run scenarios called "test plans," which define what kind of requests are sent, when, how often, and with what data.

One of JMeter’s strengths is its ability to create complex test scenarios through configuration of thread groups, timers, assertions, and logic controllers. For example, it can simulate hundreds of users logging into a website, navigating different pages, and performing transactions at different times. The GUI-based interface makes it easy to create these test plans without writing code, though it also supports scripting for advanced use. However, it’s recommended to not use GUI for stress point load tests.

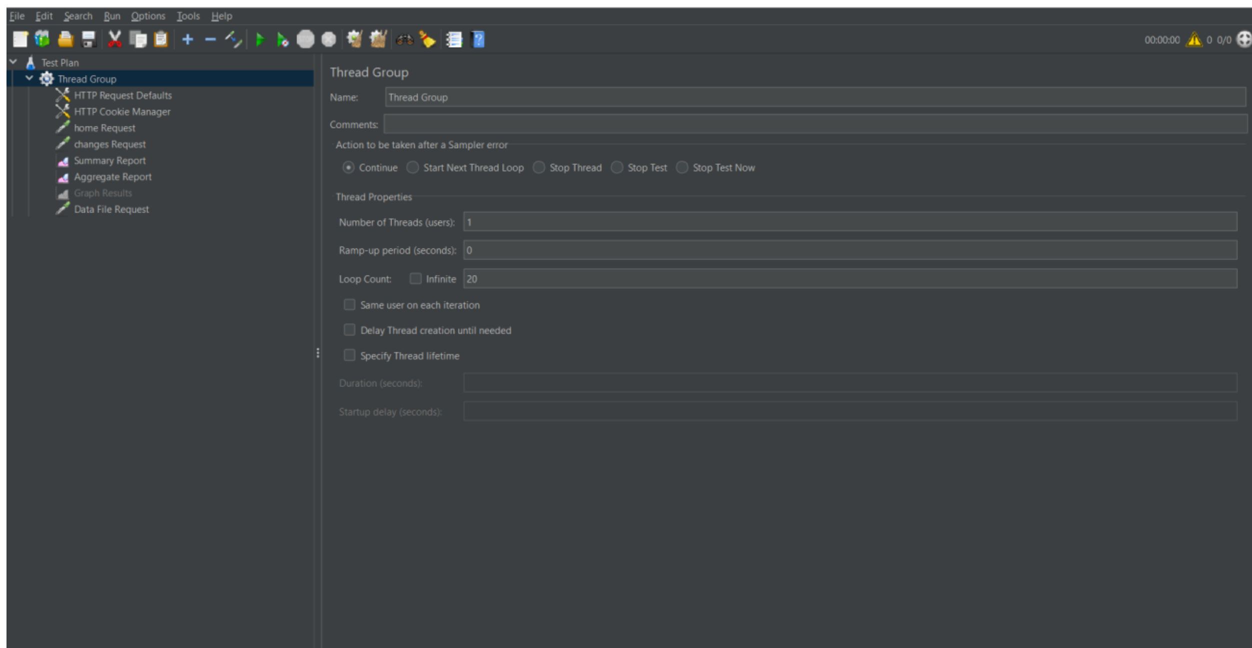

Figure 2.2 JMeter GUI

JMeter also supports integration with CI/CD pipelines and other DevOps tools, allowing automated performance testing as part of deployment workflows. JMeter is particularly useful for evaluating REST API performance, such as measuring response times and throughput under various simulated workloads. It has also the ability to generate structured report and dashboards for the runs that are proceeded. JMeter has a powerful GUI that help tester to complete the majority of test tasks easily and reliably.

For example, we run a performance test routine on a website and configured it to have 20 iteration and single thread. The following is the JMeter result dashboard that reveals a group of reports about different aspects of the test.

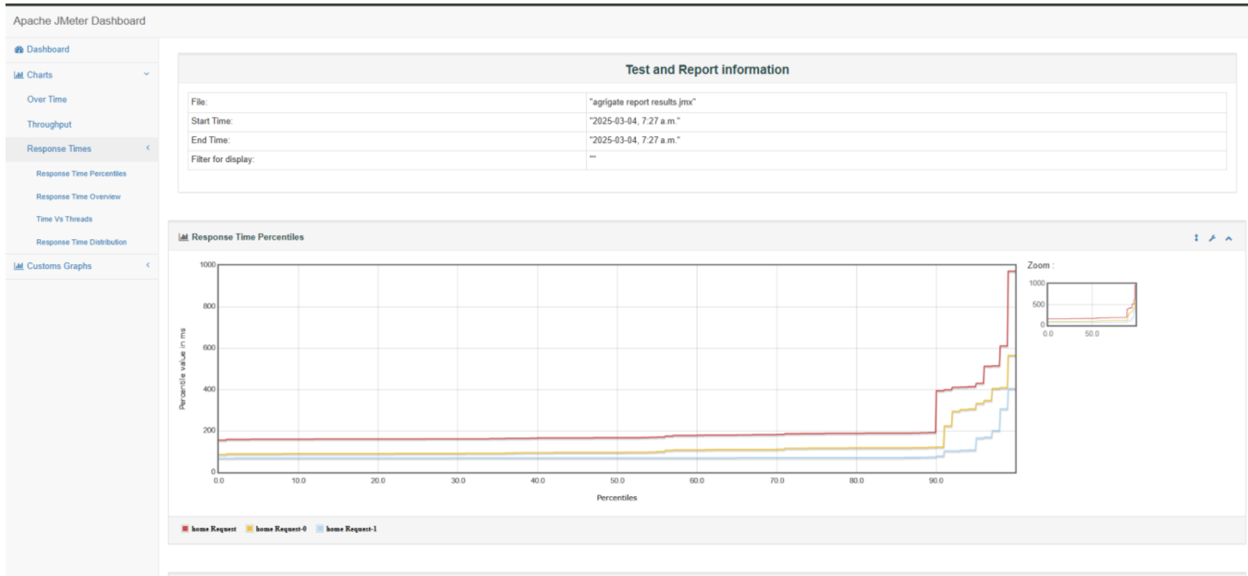

Figure 2.3 Response Time Percentiles in JMeter Report

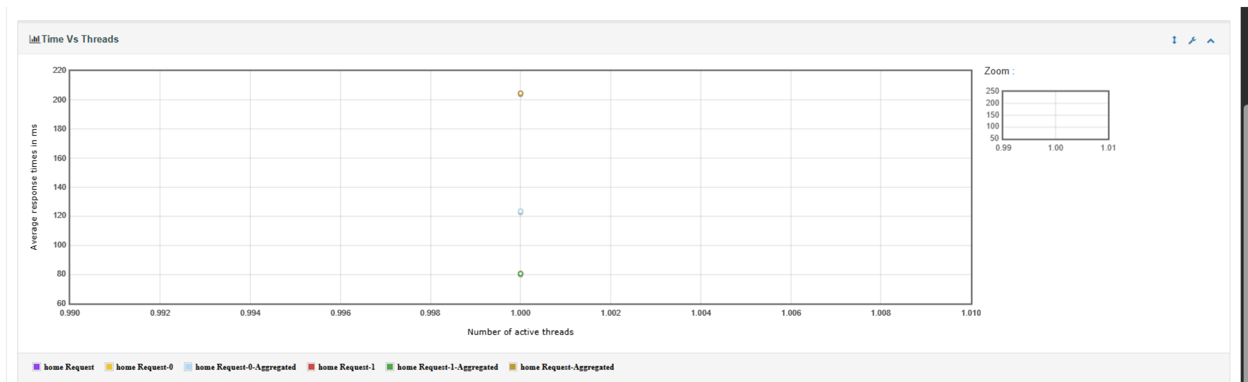

Figure 2.4 Time Vs Threads in JMeter report

Despite its advantages, Apache JMeter has certain limitations. Being a Java-based application, it can consume considerable system resources when simulating a large number of users on a single machine. To address this issue, JMeter supports a distributed testing mode, which allows multiple machines to work together as load

generators. However, JMeter can be complex to configure for specific test scenarios, often requiring significant effort to set up. Additionally, it does not natively support running automated tests across multiple nodes simultaneously with unified, structured result reports for easy performance comparison. Nevertheless, when benchmarking modern web services such as Azure App Service, JMeter remains a valuable tool. It enables the simulation of user traffic and the collection of performance metrics in a customizable and repeatable manner, facilitating reliable evaluation of service performance.

3. Cloud Benchmarking Framework

This study proposes a benchmarking framework to measure and compare different key metrics of web services on the cloud. It describes the features of this framework and the methods to achieve its purposes reliably and implement it in the real world.

3.1. Characteristics Of The Proposed Framework

The proposed benchmarking system comprehend the main features of the current software benchmarking solutions yet it is improved with additional features for more efficiency and accuracy in cloud environment where services are distributed on different nodes with different redundancy levels instead of relying on singe central server. The changeable capacity of dedicated resources is another characteristic that differ cloud systems from other conventional ones where many virtual instances share the same infrastructure and combat to consume the available resources at the time where job managers guarantee the concurrency of tasks fulfillments in the specified timeframes. Virtualization also in the cloud revealed kind of plasticity in the software deployment and resources allocation with the assistance of development operations DevOps, Infrastructure as code IaC methods, containerization where the software is accompanied with the specification of build, package, deploy, host allocation and all other specification of software infrastructure including dependencies, libraries and operation system. All of these operations are being proceed under the umbrella of version control which provide the ability to manage how the changes are applied in many aspects and especially source code version control systems VCS and tagging (in images like docker image). Furthermore, the wide accessibility of cloud system made the security a major subject in this technology. For each service provided on the cloud, there are a bunch of security standards that are applied as tools, methods and polices which are required to be set to secure satisfying level of safety in the system. The worldwide distribute of cloud system made dealing with this type of systems more complicated and involved more variables to the equation. The performance of the service differs depending on the geographical location of the warehouse where the service is hosted. These parameters are usually set when the cloud instance is created and the closer geographically to this instance better performance it gets (in general). We cannot talk about cloud services

without mentioning the material factor of it. Could service are considered relatively more expensive than conventional hosting solutions [5] and the services are provided in tires and the higher standards required the higher the costs get (and the relation is not linear) which lead us to adopt a very detailed plans on how the system will be formed on the cloud [6].

All that mentioned features gave cloud systems unique form and required -as a result- special research and studies to produce the best out of these systems.

This research proposed a benchmarking framework that is tailored to tackle the difficulties that arise as a result of these complication in cloud systems. The following, are a group of features and qualities that are necessary for any benchmarking suite that are dedicated to measure and compare cloud systems capabilities.

3.1.1. Comparability

3.1.1.1. Precise Results

The most important feature of the benchmarking tool is the ability to compare the subject nodes with reliable measurement. The instability of cloud system made it rather challenging to run the same test case on two separate nodes and compare the results. To make the result of the tests comparable, the time of the external factors must be eliminated from the comparison. This feature can be achieved by measuring the performance on the load service itself instead of loading it on the server side, and then the measurement result is sent back as a response.

3.1.1.2. Same Inputs

The benchmarking framework must guarantee also that the exact test case is run in the exact way on all subject nodes. In case of comparing the performance between two nodes, the same load must be run on both nodes with same interactions, same number of threads, load amount (data size), and the exact number of simulated users that are sending the requests.

3.1.1.3. Same Tier

Since cloud instances are created with a wide range of configuration, the benchmarking process must guarantee that the target nodes are instanced with the same hosting tiers.

For example, app service with premium tire will apparently has better performance than other instance with free tier. This also include that the initial dedicated capacities of the resources (like memory, CPU, storage, network bandwidth, etc.) must be exactly the same in the compared nodes. The other non-functional requirement is matching the cloud configurations for all subject nodes. The region where the instance is created is considered as an important factor in the availability standards. All the subject nodes are required to be located in the same cloud hosting region which is specified depending on the place of the benchmarking client and the place where the experiment is taking place in.

3.1.1.4. Security

When conducting benchmarking on cloud nodes, it is crucial to ensure that all subject environments are configured with the same minimal security standards. This consistency is necessary to guarantee fairness and comparability between the tested nodes. Even slight differences in security configurations can introduce performance variability, leading to unreliable results and invalid conclusions. In benchmarking scenarios, firewalls can add variable latency or block specific types of requests based on traffic patterns. To achieve accurate and consistent results, it is recommended to disable firewalls temporarily during the benchmarking process. Authentication mechanisms such as OAuth, API keys, or token-based systems introduce additional processing time for each request also. These delays may not be significant in individual use cases but can accumulate rapidly during high-volume load testing, hence, affecting the outcome of the benchmarking experiment. Furthermore, Antivirus software or endpoint protection systems running on a node or network follows can consume CPU, memory, and disk I/O, especially if real-time scanning is enabled. This overhead is unrelated to the actual workload being tested but can still influence the performance metrics.

3.1.1.5. Timeframe

When benchmarking cloud nodes, it is essential that the tests are conducted within the same timeframe for all nodes. This ensures that the benchmarking results are fair, accurate, and directly comparable. Cloud environments are shared infrastructures, and their performance can fluctuate depending on the time of day, day of the week, and

overall system load across the provider's network. So, Running benchmarks at inconsistent times can introduce external factors that ruin the results and lead to false conclusions about the performance of the tested nodes.

3.1.1.6. Data Compatibility

To compare the resulted data in an efficient way, this research recommend that the data is processed and visualized in a unified way with the same data processing tools. Despite the fact that all data processing tools have the same standard functionalities there are differences hidden in its implementation behind. For example, when using standard deviation of the aggregated data, some analytical tools may use corrected standard deviation as the default function while other consider uncorrected standard deviation as the default. Also, the difference in the geometric representation of data between data visualization tools leads to misinterpretation of the produced values. To avoid this inconsistent interpretation, it is recommended to use the same functions in the same mathematical tools (and libraries) on all resulted data (and for all subject nodes) in addition to assure that all resulted records have the same data structure to avoid errors in fields mapping and data typing operations.

3.1.1.7. Dedication

To eliminate the effect of external factors in the resulted benchmarking data, it is good practice to dedicate the subject nodes for the benchmarking operation and avoid running any other irrelevant tasks that are not an evaluating node. The capability of the client host also should be considered during the benchmarking process. The unstable performance of the benchmarking clint will lead to unreliable results because of resource bottlenecks or client-side error like:

- heap overflow error
- stack overflow error
- request timeout error

(which are considered external factors on the benchmarking process) that may appear as a result of not dedicating the client for the benchmarking operation. Furthermore, if all the subject nodes are run on the same environment (like same cloud service provider), it

is better to run the benchmarking client process in the same environment and connect all the nodes with the client by a devoted Virtual Private Network VPN with high bandwidth.

3.1.2. Coverage

The benchmarking process must cover as match as possible of resources and functionalities in the target nodes. For example, it may happen that node 1 has better performance than node 2 in CPU but its memory is a bottleneck that give it worse performance than node 2. For this reason, benchmarking framework must give the ability to run the loads on all resources (CPU, memory, File I/O, network, and other) and involve its key performance to the resulted data for comparison. For CPU loads the load should cover different kind of arithmetic, logical, and transferal operations. For memory loads, the test cases are scaled form small to big data size for memory read and memory write operations. For storage processes, file read and file write operations are implemented. The test is run with gradual increase from small to big chunks and the results are registered separately with enough interval between the runs. For network, since this research does not focus on security metrics in the network, the nodes and the cloud environment are configured with minimal security practices to decrease its effect on the benchmarking resulted data. Network test aims to measure how easy to access the target node and evaluate it in way that makes it comparable with other nodes. In our case, network performance is not the major target of our benchmarking, having said that, it is considered an important external factor that affect the performance of the subject node.

All the loads are run for short/long periods of time, small/big data sizes and with/without parallelism. This means, the data sizes in test cases are varied to cover all sizes levels from tiny chunks then go up till reach the maximum capacity (critical point test). The test cases also cover the number of tasks that are processed at the same time. It starts with single process (that serve single client) with single thread. Then, the scale gets increased to serve more and more parallel processes until the overload threshold is hit.

Other factor could be covered in measuring the performance of the cloud service like the time when the test is run. Varying time timeframe when the test is run expand the vision regarding the node performance since physical resources on the cloud are consumed by many virtual nodes most of time. In this case, some cloud technologies are more affected from intensive cloud usage than others during the rush hours depending on the hardware, software and virtualization level that the service is built on. For example, the performance of service that is hosted on a leased dedicated physical server in the cloud has far higher performance than a service that is hosted on a cloud shared instance in rush hours.

3.1.3. Configurability

Since recent cloud services instanced, adapted and managed by users (without a direct intrusion from service providers), cloud services are equipped with dozens of configuration parameters to shape the current stat and behavior of the service. For the same reason, the proposed cloud benchmarking framework is flexible enough to meet the different requirements of benchmarking cloud services in different environments without the need to modify the core of this framework with minimal utilization of emulation tools. All the variables of the proposed framework are easy to set, modify and expand. The configuration parameters are clear for the user, strongly typed and easy to understand with occasional need for the user to refer to the documentation to understand how to manage it. One of the recommended methods to implement this configurability is to have all the configurations are gathered in a single location with a structure that is readable and changeable by human and machines. The configuration controls all the functionalities and aspects of the benchmarking system for client side and server side and it covers:

- Methods and loads
- Input parameters
- Labels
- Synchronous and asynchronous iteration
- Target Nodes configurations

3.1.4. Expandability

It is the ability of the cloud benchmarking tool to support more benchmarking functionalities in an easy way. Even when the major metrices of benchmarking are defined and implemented, the benchmarking framework has the plasticity to add new measurement features to evaluate a new aspect of the target node. The expansion includes the availability to add new jobs and run it on the client side or server side and manage its results and append it to the standard data store in a flexible yet compatible schema.

3.1.5. Comprehensive And Clear Output

The proposed framework has 3 main kinds of outputs, logs, insights and result data.

3.1.5.1. Logs

Logs are generated data that gives information about the benchmarking process on the client side. They are generated in real time while the benchmarking process going on to give the system testers enough information about the current situation of the process. The log operations are required to comprehend all the client's and servers' activities in a clear and readable way. Clarity includes coloring the output depending on the logging level, spacing and indenting the text relating to execution depth with identifying the exact current state like iteration number, thread number, current endpoint label and timestamp. It is important for the generated log to persist locally or saved on a remote storage so the tester has the ability to refer to it anytime later easily. Logs are usually used to give user information about the currently running operation like the current state, progress, errors and warning messages.

3.1.5.2. Result data

The objective of benchmarking tools is to generate data that could be used to benchmark services' performance and compare the results. The resulted data from the benchmarking tool requires high level of compatibility with standard analytical tools. This means that the resulted data schema must be consistent and machine-readable (like

relational schema or graphs etc.). The resulted data are organized so attributes are pre-identified, typed, headed; each record is labeled, identified, and related to exact test case. The resulted data are able to be read by standard analysis and Business intelligence BI tools and has flexible schema and easy to expand in future developments.

3.1.5.3. Insights

Unlike logs data, insights data takes major part in the benchmarking result data that are used to compare cloud nodes performance. It is required form the cloud benchmarking tools also to give insights about the operations that are run on the subject nodes which play important role in evaluating the node's health. Insight in the benchmarking framework is different from that one that cloud services providers have. Despite that cloud platforms insights give information about the general status of the cloud object; these insights are centered around the target node and they are formed (structured and filtered) in a way that makes it comparable with other node's insights. The proposed framework provides structured insights about the requests that are sent to the subject nodes to measure important metrices like response time and the amount of data that is transferred and the status of these requests whether they are succeed or failed.

3.2. Implementation

As a part of the research, a benchmarking framework is implemented in a way that meets the majority of cloud benchmarking framework requirements that are mentioned in this research as a test. The following, are the specifications of the implemented system:

3.2.1. Client-Server Architecture

The implemented system has client-server architecture where the load services are hosted in the target nodes and client is in a separate node which sends HTTP requests to the target nodes and save the responses as a list of records in a single place in the client host.

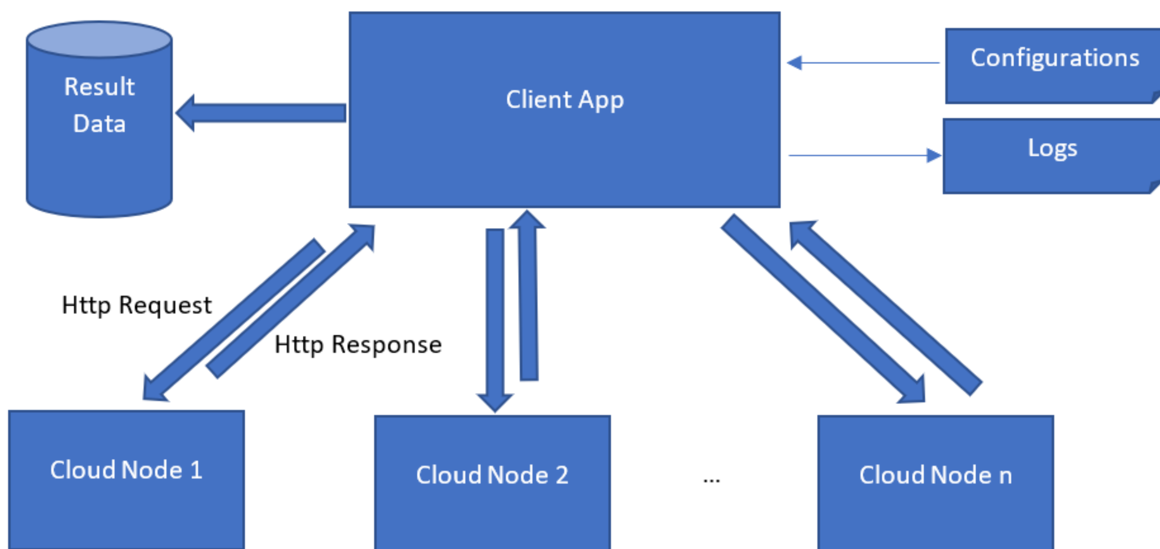

Figure 3.1 Client-server architecture for the proposed cloud benchmarking system

3.2.2. The Load Service (Server Side)

It is the service that run on the subject node to measure its resources' capabilities. It has a group of endpoints that listen to the requests from the client, execute the load, measure the performance then send back the measurement results as an http response. All subject nodes have the exact same load service (with same version) and run the same exact tests. The load web service is implemented in Dot Net Core since it's wildly supported by the all the famous cloud service providers and easy to deploy and test. Publishing Dotnet core to the cloud service is relatively easier than other web frameworks and it's possible to publish directly from the development environment or from code base by DevOps operations. Also, it's possible to build a docker image and push it to docker hub so a container could be instanced in any place in the cloud.

3.2.2.1. Endpoints

The load service exposes a set of endpoints that receive client requests, apply the specified workload, record performance metrics, and return those measurements in the HTTP response. The following are the implemented endpoints:

- **CPU load**

Accepts the number of iterations and threads as input. Iteration number decide how many times the mathematical operation (square root) is applied during the run, and threads count decide how many threads are run at the same time for the specified loops separately. This endpoint returns integer number that represent the number of milliseconds this operation took to finish the specified task.

- **Memory Read**

Accepts buffer size (bytes) as input. It generates data with the specified input size and store it in memory then start calculating the time for the process to read the data from memory and return the calculated milliseconds as http response.

- **Memory Write**

  Accepts buffer size (bytes) as input. It generates data with the specified input size and store it in memory and calculate the time of this process then return the calculated milliseconds as http response.

- **File Read**

  Requires file's block size (bytes). The endpoint generates file with the specified input size and store it in the disk storage then start calculating the time for the process to read the data from the file and return the calculated milliseconds as http response.

- **File Write**

  Requires file's block size (bytes). It generates data with the specified input size and store it in the disk storage and calculate the time of this process then return the calculated milliseconds as http response.

- **Data Transfer**

  This endpoint receives amount of data and return it -as it is- as a response. The benchmarking measurements for this test are noted in the client side and the amount of the transferred data also. The objective of this run is to calculate the period of the time between the moment when the request starts until the response received.

- **Custom load**

  This endpoint receives python code as input parameter and execute it on the server host then return the result. The result is the content of a variable in the python code that was already specified in the input. This feature makes the load service extendable and open for further custom loads which may be required on the target nodes. The resulted response of this endpoint is captured by the client and added to the benchmarking result data.

  Note: This endpoint is a vulnerability point in system's security. Since the benchmarking system is setup with minimal security configurations, this endpoint allows attackers to inject code and use this node as start point to attack other targets.

3.2.3. Client app

Benchmarking system has a single client app where the benchmarking process starts. The client app gives the tester the ability to configure how the benchmarking process goes and initialize and trigging all load processes by sending http requests to their endpoints. The client runs all the test cases on all subject nodes equally, construct the result data and generate logs in real time. In this research, the client app is implemented in Dotnet Core as console app that makes it portable and compatible with different operation systems and shows the logs formatted in real time with the ability to save it in log file also.

3.2.3.1. Configurations

All benchmarking runs are managed by configurations that are set by the tester before starting the process. The configuration file contains details about all the intended test cases to be run on the nodes with information about the target nodes too. For each endpoint, there are a list of cases that are meant to be executed by the client on all nodes. Each case contains the input parameters of the request and labels to identify the resulted data. The config file also contains information about the nodes that run the load services and listening to the requests; for each node, config defines the url and the label for that node. The full schema of the configuration file in the client is in Appendix A.2.

3.3. Case Study: Azure App Service Vs Docker Container Evaluation

The implemented benchmarking system is used to benchmark two Azure cloud technologies: Azure App Service and Azure Docker Container to test the efficiency and the usability of the proposed framework. The client was run on Azure virtual machine and those 3 objects (client, app service, and container) are connected by a dedicated high-bandwidth VPN. Both subject nodes are registered and labeled in the client's configurations.

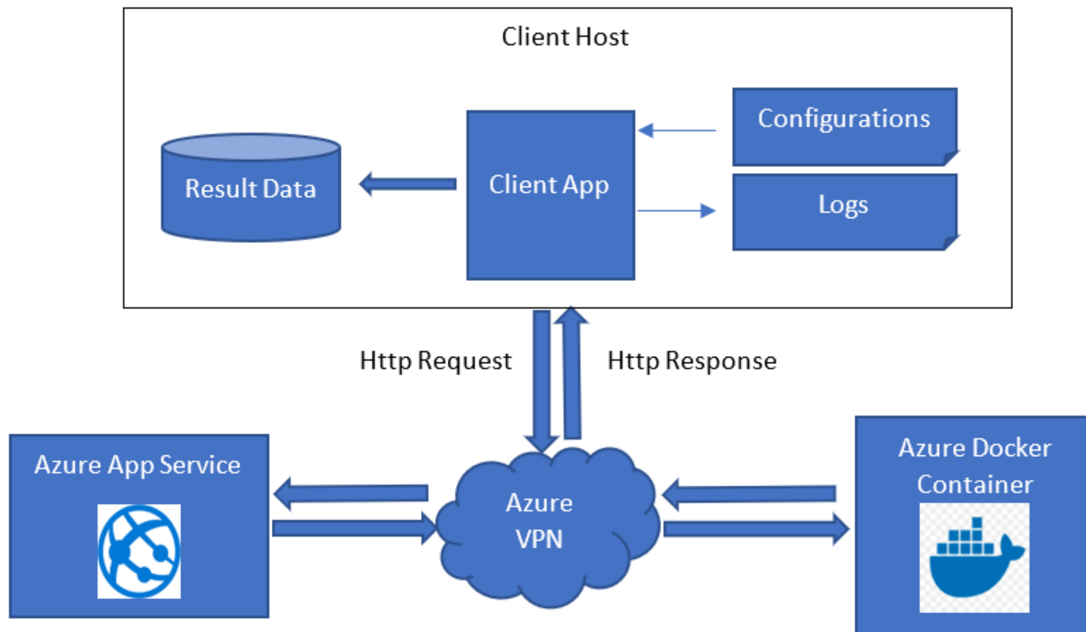

Figure 3.2 Cloud Architecture for Azure App Service Vs Docker Container Benchmarking

3.3.1. Azure App Service

An instance of Azure App Service is created in Azure platform with Tier (S2 West Europe, 5GB 2Cores). This service is dedicated to serve the benchmarking client and not used or connected to any other client. Then the load app is built and deployed to this service.

3.3.2. Azure Docker Container

The benchmarking load app is built and composed into docker image with all libraries and dependencies layers. After that, a docker container in Azure is instantiated from the built image with the same tier of Azure App Service that made for benchmarking.

3.3.3. Client App

The client console app is set up in a node within the VPN that the other two subject nodes are set in to guarantee a stable accessibility. It's configured to run a diverse kind of tests for each subject resource (single/multi threads with many iterations) with different levels of loads. The client is configured also to identify the connected nodes with the required links and labels are set to distinct the nodes in the resulted data. The full configuration data of the client are in *Appendix A.1* .

4. Findings and Results

After running the experiment (with same exact configs, input parameters, and data sizes) on both services (Azure app service and Azure Docker container), the output data revealed the key performance differences between the two service for each aspect with the help of BI and data visualization tools.

4.1. File Read

The chart in Figure 4.1 reveal the file read performance for both app service and docker container on Azure when the file sizes are 1MB, 10MB and 100MB for single user (client thread) then for 10 users (client threads). For file size 1MB, app service has close performance to container, however, the higher file size to read the bigger difference of performance between them appears. Container had a better response time than app service and the almost all the values distributed close to the average which tell that it has stabile response time. Meanwhile, app service has bigger spread around the median and the bigger the file to read the bigger range of data distribution appears. Running many processes of reading files simultaneously reveals that container did better job than app service too. In general, the average response time for file read for service with 10

users at the same time is better when using docker container that it is in app service. Despite the response range in asynchronous run for container is bigger than it is in synchronous one, it still shows rather better (more stable) performance than app service. The increased size of the files to be read didn't affect the stability of the response time. However, we see in the Figure 4.1 that app service has far big standard deviation for response times. This means when many users run the load at the same time, some users will get their response relatively faster than others who will wait more despite they sent the request at the same time.

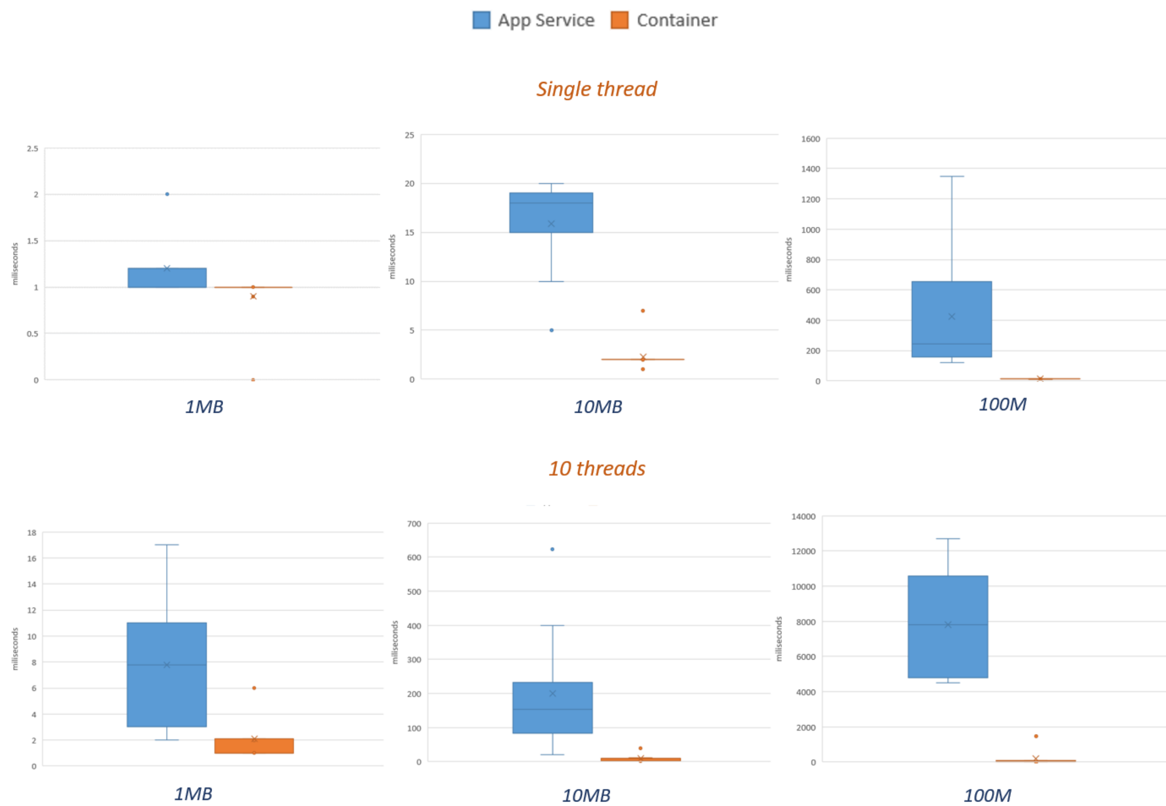

Figure 4.1 File read performance comparision between App service and docker container in Azure

4.2. File Write

A file write load was tested on both the App Service and Docker container using different file sizes: 1MB, 10MB, and 100MB. The tests were first run with a single user (10 iterations), then repeated with 10 users (client threads) running simultaneously.

With a single user, the App Service showed a wide variation in response times when writing 1MB files, with some outliers reaching 53ms and 55ms. For larger files, its performance became more consistent, averaging around 140ms for 10MB and just under 2000ms for 100MB. On the other hand, the Docker container showed very stable performance across all file sizes, with minimal variation around the average. In every case, the container significantly outperformed the App Service -at least 7 times faster- and the performance gap increased with the file size. When running with 10 users simultaneously, the App Service again showed inconsistency, especially for 1MB file writes: some requests were processed as fast as 17ms, while others took around 40ms, indicating asynchronous handling. In contrast, the container maintained stable and lower response times even under concurrent load, outperforming the App Service across all file sizes. Overall, the performance difference between the App Service and the Docker container becomes more noticeable as file size increases, both in terms of speed and consistency.

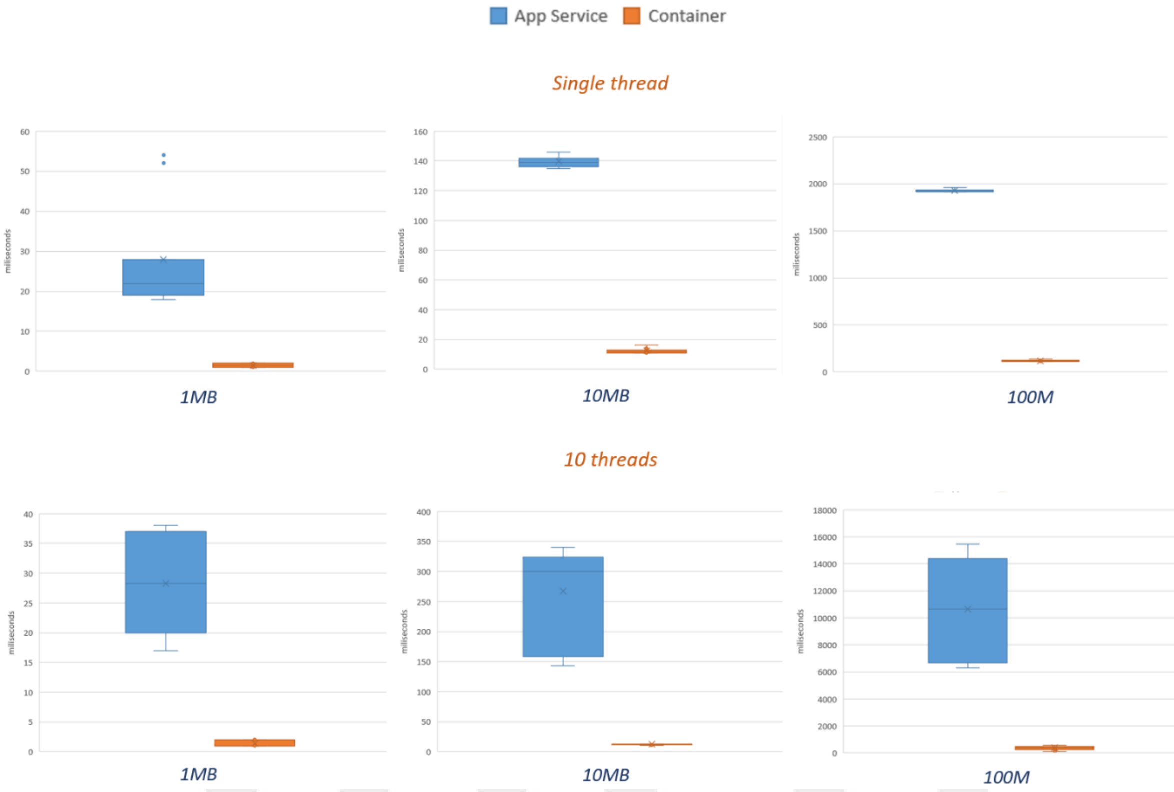

Figure 4.2 File write performance comparison between App service and docker container in Azure

4.2.1. File Write Stress Test

We re-run the file write test multiple times, and for each run, the file size increased by an effective ratio. The test (in Figure 4.3) is run on 200, 500,1000,1200,1500, and 2000 MB sequentially. App service and Container showed a linear increase in processing time for file write, but with different slope. App service kept responding with throughput relative to the size of the file that is being processed. However, all the runs that exceed the file size of 1500MB have failed. The subject App service is instantiated with tier S2, and despite the storage limit for this tier being 512GB [7], a single file write process with a size of more that 1500GB always fails, and there are many possible reasons for this. This process is triggered with a REST API request, even with increasing the request time limit in the web service configuration, there is still a time limit (in Azure configurations) for the request, and once this time is exceeded, the connection will be lost. Other reasons for this failure could be the allowed size for a

single file or the processing time limit in the operation system of the service. Also, the load service is implemented in a way that deletes the written data after the evaluation process is finished (garbage collection), so there is no accumulation of waste as a result of repeating the runs many times. Docker container revealed a linear relationship between the load file size and the time taken to write the file. This relation sustains until 2000MB of file size; after this point, all file-write attempts fail. The same possible reasons for failure that are mentioned for the app service apply to container. From these results, we indicate that Container has higher durability than App Service under intensive file write loads.

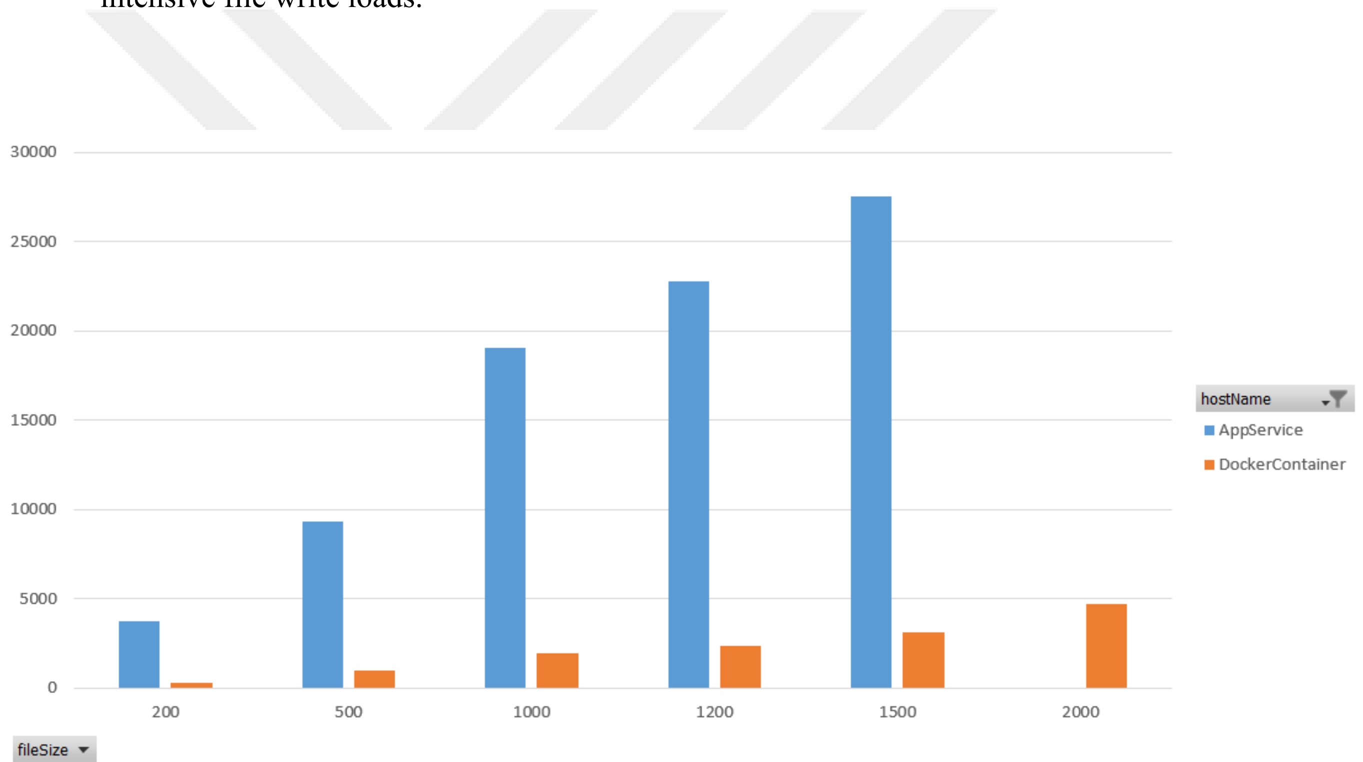

Figure 4.3 File write time per data size

4.3. Memory Read

Memory read performance showed stability for both app service and container when the run is sequential (single thread). Container performance is measured as 3 times better as app service for data sizes 10MB, 50MB, 100MB with single thread. When running 10 memory read processes simultaneously, the periods range of the process in app service is registered in a very wide scale comparing with container. The average time of app service is also 6-times (at least) higher than the average time of container.

Figure 4.4 Memory read performance comparison between App Service and docker container in Azure

4.4. Memory Write

Similar to the results observed during memory read performance testing, the Docker container continued to demonstrate superior performance over the App Service in memory write operations as well. However, the difference between the two was not as significant as in the read tests. When performing memory registrations (writing data to memory), both App Service and container showed closer average response times, especially when running with a single thread or multiple synchronous threads. This was a noticeable contrast to the memory read performance, where the App Service consistently lagged far behind the container. Despite the closer averages, the container still maintained an edge, particularly in single-thread runs where it performed up to six times faster than the App Service. That said, as the size of the memory data being written increased, the performance difference between the container and App Service began to shrink. In other words, the more load we tried to write into memory, the more similar the performance results between both platforms became. It's also worth mentioning that although outlier values were observed in the container results, these were rare and didn’t significantly affect the overall outcome. Most of the container's response times stayed close to the average, showing a stable and predictable behavior. On the other hand, the App Service showed more variation in its response times, even though the average was close in some scenarios. This stability in the container’s performance, combined with its generally faster execution, confirms that it handles memory write operations more efficiently - especially under lighter loads or smaller data sizes.

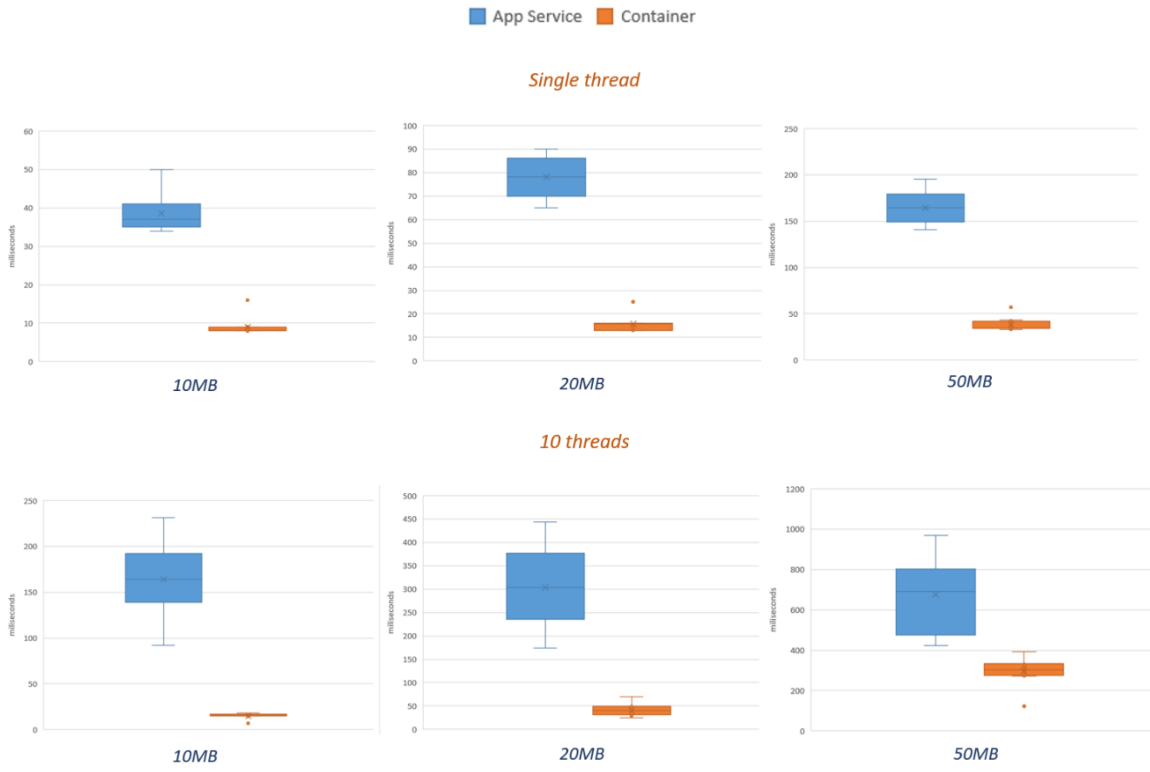

Figure 4.5 Memory-write performance comparison between App Service and docker container in Azure

4.4.1. Memory Stress Test

Other runs are conducted with increasing the size of the load data for each run. The objective is to discover the limits of available memory size particularly. The requests are run in a sequence of memory write requests, with increasing the data size every time. The app service experienced a steady increase in response time, correlated with the increase in data size. App service failed to complete the memory write operation with data sizes above 1000 MB. For all data sizes, the container has 4 times better performance than App Service. It also revealed a linear relation between the data size and the performance of memory write. Meanwhile app service had a limit of 1000 MB data size without errors, the container exceeded this threshold to provide a stable performance until the edge of 2000 MB; After this boundary, the majority of memory write operations fails.

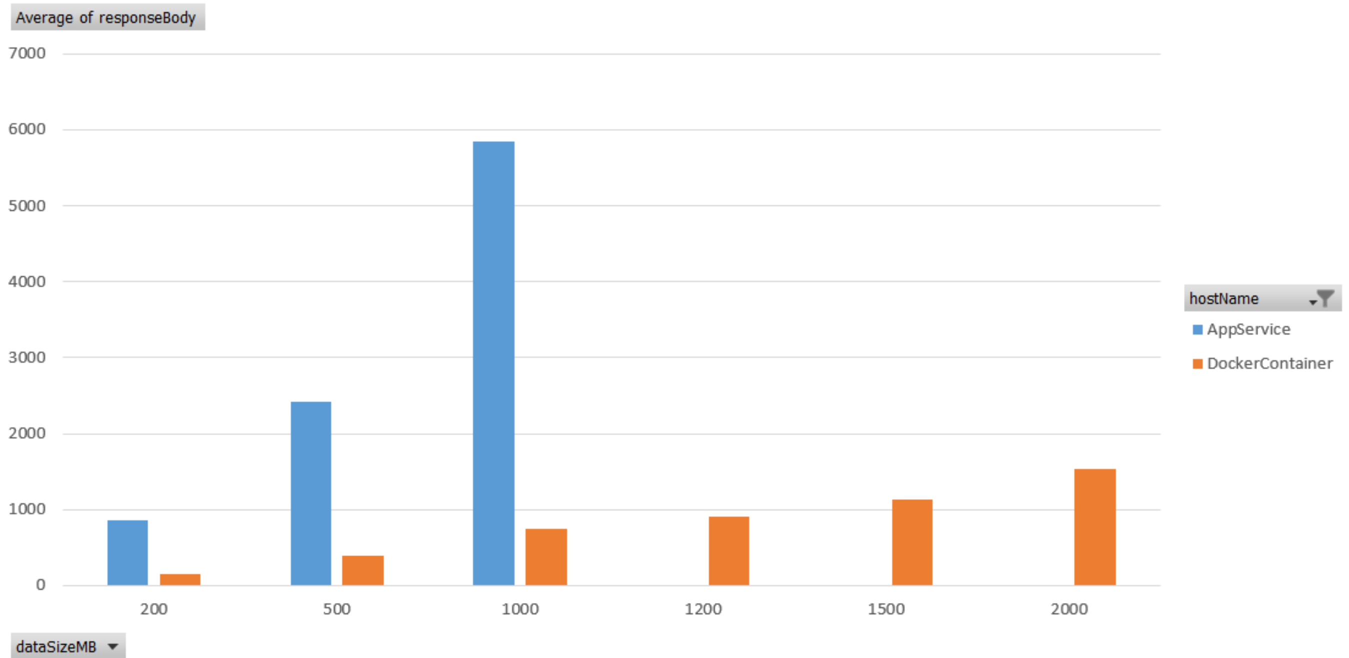

Figure 4.6 Memory write time per data size

4.5. CPU Load

The CPU load that applied is a simple mathematical operation (square root) looped 100000 times. For the first run, we run 20 client threads simultaneously and each request is handled with single thread in the server side for both app service and container. In Figure 4.7, the left side reveals that with 1 thread in the server side, the handling 20 users asynchronously took 65ms while it took only about 5ms for container. The same load is run again but with single user this time with 20 threads were run asynchronously at the same time on server. In the Figure 4.7 (right side) we see that finishing this run took 90ms for app service while it took only just under 30ms. This shows that container has lower processing time than app service whatever is it with single vs multi-users or single vs multi-threads are run on the server side. This

experiment is made in a way that shows the difference if the parallelism is applied from external factors (like users) vs the internal factors (multi-threading).

When we compare the results of two runs of the container (multi-users vs multi-threads), we notice that the difference in performance of the container had 20 users with single thread for each outweigh the running 1 user with 20 threads processing despite a single docker container is used for this experiment (no containers orchestration tools - like Kubernetes- are used or setup for this experiment).

When 1 user triggers 20 asynchronous operations, All those operations are handled by the same process. They rely on the ThreadPool, I/O scheduler, and task queues inside the runtime (in our case .NET). There are many concepts in web service affecting these results like ThreadPool, Task scheduling delays, CPU context switching overhead, Single-process resource throttling by Azure.

Azure may restrict the CPU portion per container, depending on the configuration (App Service Plan, Container Instances, etc.). So, A single container running 1 process with 20 async tasks might be CPU-starved or I/O throttled. When 20 users connect externally (via HTTP or similar), they trigger independent request handlers which can be processed more efficiently by the container’s web server or might result in parallelism at the request level, even without multiple threads (via async I/O). So, while both approaches use one container, external requests leverage the container's web server’s concurrency model, which is optimized for many simultaneous inbound requests.

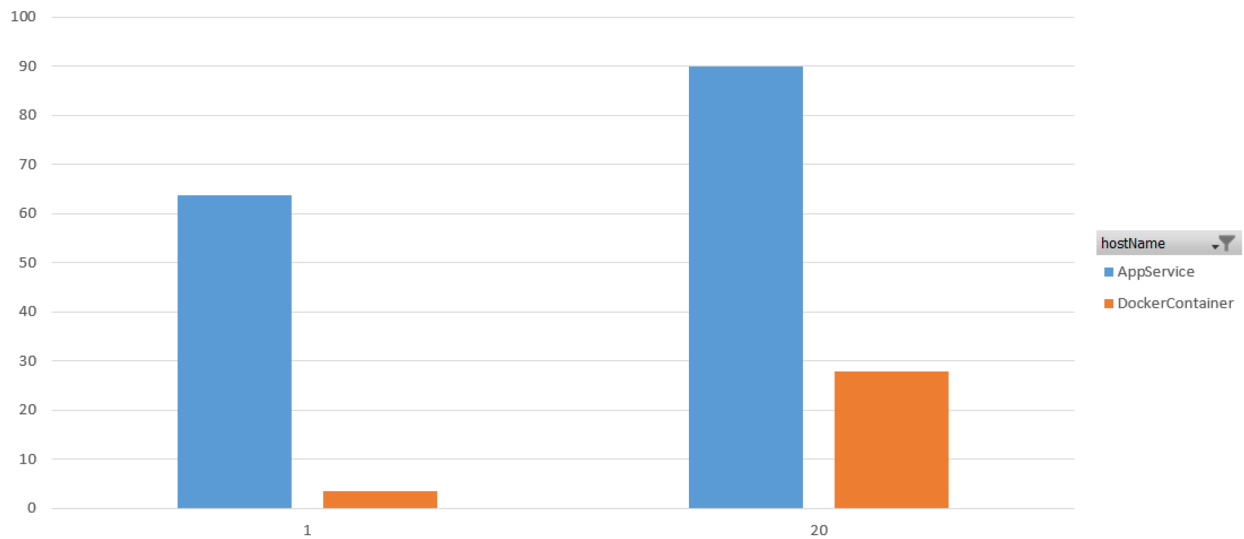

Figure 4.7 CPU load performance comparison between App Service and docker container in Azure for single vs 20 users

4.5.1. CPU Stress Test

A sequence of CPU load runs is measured with an increased amount of load in each run. The Figure 4.8 reveals the increase in the processing time of App Service and Docker Container when increasing the number of simultaneous processes (threads) in the threads pool. App Service processing time increased continuously with the increase of the number of load threads. In some stages (5000 to 5500, 7000 to 7500), a change in the pattern appeared. After the processing time reaches a specific level, it experiences a sudden decline (better performance) despite the steady increase of the load. This happened because of the expansion of the CPU thread pool capacity dedicated to this service. In Azure App Service, there are worker tasks that monitor the percentage of CPU resource usage and increase/decrease the capacity depending on the load. In our case, every time thread management processes get close to being overloaded, resource management job increases the capacity of this process, so the processing time decreases. The container showed a 25% better performance during the runs in general. In Figure 4.8, we notice a decrease in processing time for the container when increasing the load

from 8500 to 9000. No failure was noticed for all runs, and the resources (for app service and container) had a continuous expansion as the load went higher.

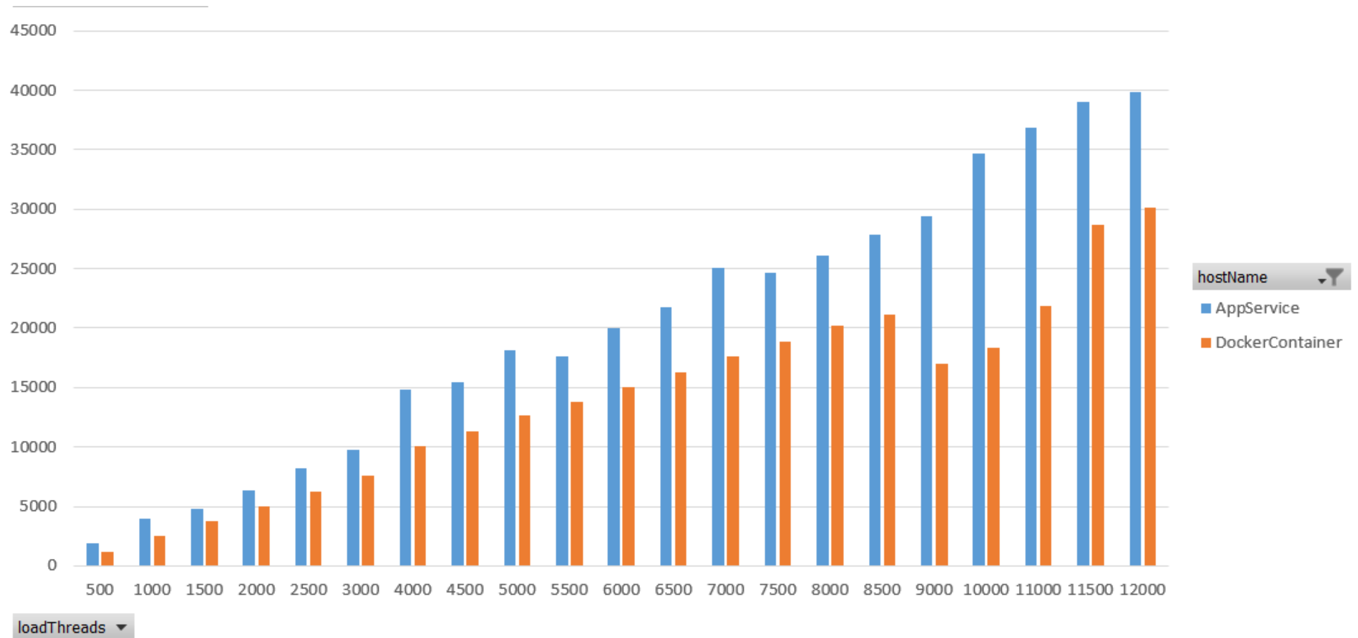

Figure 4.8 CPU processing time per threads count

4.6. Network Stress

Hosting services in cloud systems are setup differently depending on the technology behind that service and the environment, so, one of the major metrices to consider when benchmarking services on the cloud is accessibility. It may happen that two nodes have same resources capacity allocation on the cloud but are implemented in different technologies which one of them requires extra data processing than another which lead to longer response time.

In this experiment we send a request with load that has a specific size of data, and we get response with the same amount of data. We measure the period between the moment when the request is sent and the moment when the response is received (unlike the other experiments when period calculation is occurred at the server side and the period is sent as a response). The objective of this run is to measure the level of accessibility of the

subject cloud services and compare the response time of them. It's important to mention here that the result data is related also to the environment condition of the client including network bandwidth and request/response processing time in the client side, however, following the specifications that mentioned before (characteristics of benchmarking framework) like high bandwidth, dedicated network, high performance and dedicated client host, repeating the experiment, and other conditions play important role in eliminating the effect of the external factors on the result data. In Figure 4.9 and Figure 4.10 we see the result of runner network stress load on app service and container with data size of 1MB and 10MB separately with single thread. Since there are outlier values, interpreting the performance from time sequence chart (Figure 4.10) is easier than doing so from box plot (Figure 4.9). In Figure 4.10 we see the response time of running the load on the target node registered by the time. With small data sizes (1MB) app service and container did a relatively stable performance and container showed a slightly better performance in general. With data size of 10MB, the response recorded more fluctuation in performance and wider range around the median. We cannot tell the real reason behind the extreme outlier values that appeared, because it may be due to a network stall form the client side or server side. In general, these runs test the effect of the ecosystem of the target node rather than measuring the key performance of the node itself.

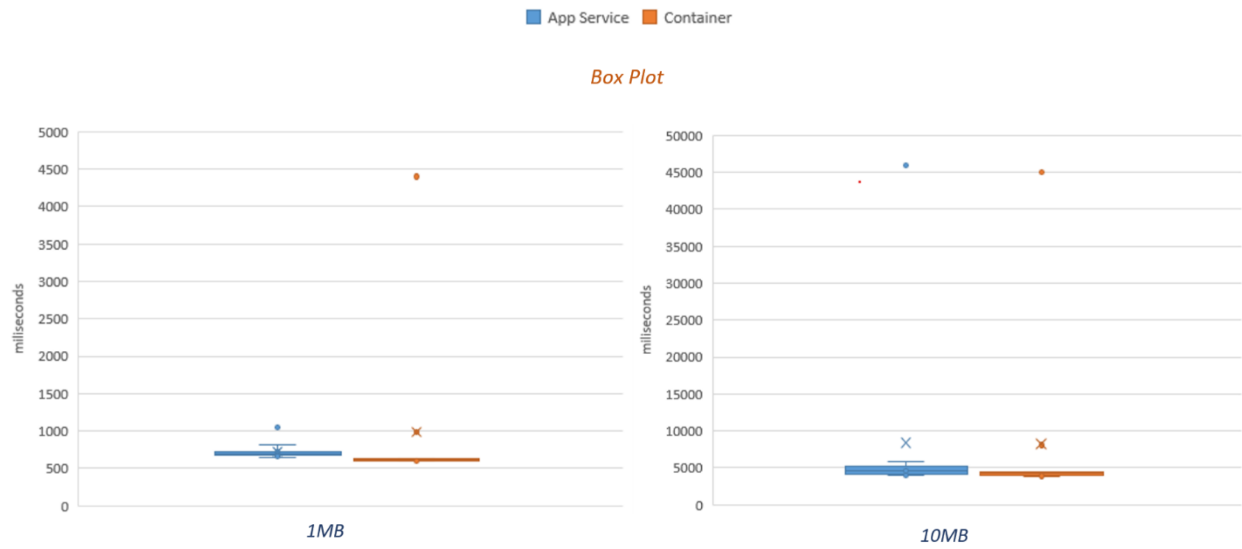

Figure 4.9 network stress test on app service vs container for data size 1MB vs 10MB

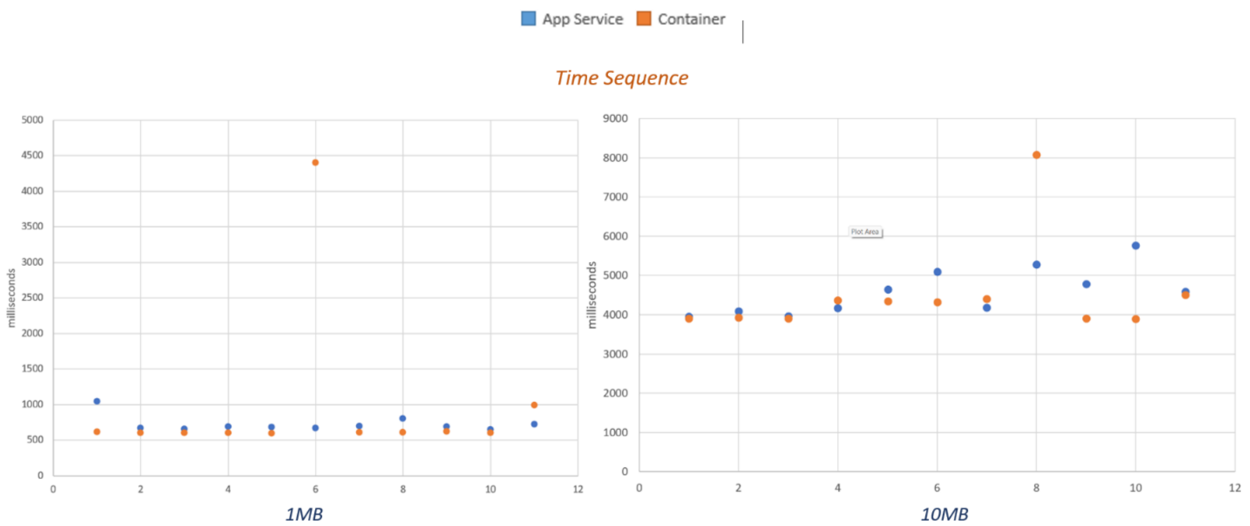

Figure 4.10 time series response time for network stress test on app service vs container for data size 1MB vs 10MB

Conclusion

This study is concerned with evaluating and comparing software performance in cloud environments. Virtualization, decentralization, global distribution, and cost; made cloud services a challenging subject to evaluate. The currently provided solutions are either missing the required compatibility with the conditions of the cloud environment or they are generalized in a way that makes it unaffordable to customize in terms of both time and effort. This study proposed a benchmarking suite that has specific properties and methods to evaluate and compare cloud services efficiently. Dedication, simplicity, compatibility, precision and timing are the most important requirements to achieve fair comparison between cloud services. Furthermore, the ability to cover the major resources in the subject node, in addition to configurations that support using multiple users in multiple iterations in the test process, are essential features in the cloud benchmarking framework. This research also recommends that structured data with clear and formatted logs and insights outputs are important factors in this context. The extendibility is also considered by offering a flexible way to add new test routine and send it to be executed on the subject node in the cloud.

A detailed description of a concrete implementation of the framework is proposed that includes all the recommended features. A practical test is conducted to benchmark the performance of Azure App Service against Azure Docker Container in terms of CPU, memory, storage, and network capabilities. Azure Docker container revealed superiority against Azure App Service in CPU, memory, file I/O, and accessibility performance for multithreaded tasks. However, a close performance with a slight tendency to containers is registered for loads that are executed with single threads.

4.7. Future Steps

This research is not final, and there are many further steps could be taken in the context of this research.

More measurements could be taken on more services on cloud like including Azure container orchestration (Ex, Kubernetes) in the benchmarking routine, or comparing the same service (like Docker container) between different cloud service providers (like

Google Vs Azure Vs IBM) and interpreting the concluded results. The conducted research could be taken for a wider scope of configurations like the variety of data scale layers, number of users, and iterations, initiating the same runs at different times, and so on.

This research is focused on measuring the main resources of the computation node (CPU, Memory, storage, and data transfer); however, this research could be taken further by supporting more resources to evaluate like DB operations (CRUD and I/O), operating system operations, and more.

REFERENCES

[1] "App Service overview," microsoft, 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/app-service/overview.

[2] Marek Moravcik, Pavel Segec, Martin Kontsek, Jana Uramova, Jozef Papan, "Comparison of LXC and Docker technologies," IEEE, 2021.

[3] R B Suryawan, R Ferdiana, Widyawan, "The Comparison of Cloud Migration Effort on Platform as a Service," iopscience, 2022.

[4] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Brian Ritchken, Brendon Jackson, Brett Clancy, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," ACM, 2019.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, Matei Zaharia, "A view of cloud computing," ACM, 2010.

[6] R. S. K. Srinivasa, "CHARACTERISTICS, APPLICATIONS AND USE CASES OF CLOUD COMPUTING," SSRN, 2021.

[7] Sergio Di Meglio, Luigi Libero Lucio Starace,Sergio Di Martino, "Starting a New REST API project? A Performance Benchmark of Frameworks and Execution Environments," CEUR-WS, 2023.

[8] Daniel Silva, João Rafael and Alexandre Fonte, "Toward Optimal Virtualization An Updated Comparative Analysis of Docker and LXD Container Technologies," MDPI, 2024.

[9] amara Dancheva, Unai Alonso, Michael Barton, "Cloud benchmarking and performance analysis of an HPC application in Amazon EC2," doi, 2023.

[10] "Security considerations for Azure Container Instances," Microsoft, 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/container-instances/container-instances-image-security.

[11] M. G. Toshe Petrovski, "Container vs Function as a Service: Impact on Cloud Deployment for Real-World Applications," IEEE, 2024.

[12] S. Alzide, "13-Cloud Computing Evolution, Challenges, and Future Prospects-2024," doi, 2024.

[13] "Apache JMeter," [Online]. Available: https://jmeter.apache.org.

[14] "Use containers to Build, Share and Run your applications," Docker, [Online]. Available: https://www.docker.com/resources/what-container.

[15] "LXC," creativecommons, 2025. [Online]. Available: https://linuxcontainers.org/lxc.

[16] Harshad Kasture, Daniel Sanchez, "TailBench A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications," IEEE.

[17] Tapti Palit, Yongming Shen, Michael Ferdman, "Demystifying Cloud Benchmarking," IEEE.

[18] S. Kelkar, "Challenges and Opportunities with Cloud Computing," ISSN.

[19] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, John Wilkes, "Autopilot workload autoscaling at Google," ACM ISBN, 2020.

[20] Marco Barletta, Marcello Cinque, Catello Di Martino, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer, "Mutiny! How does Kubernetes fail, and what can we do about it," IEEE, 2024.

[21] Yu Gan,Yanqi Zhang,Dailun Cheng,Ankitha Shetty,Priyal Rathi,Nayan Katarki,Ariana Bruno,Justin Hu,Brian Ritchken,Brendon Jackson,Kelvin Hu,Meghna Pancholi,Yuan He,Brett Clancy,Chris Colen,Fukang Wen,Catherine Leung,Siyuan Wang,Leon Zaruvinsky,Mateo Espinos, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," ACM, 2019.

[22] Vladimir Podolskiy, Anshul Jindal, Michael Gerndt, "IaaS Reactive Autoscaling Performance Challenges," IEEE, 2018.

[23] "Service limits in Azure AI Search," Microsoft, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/search/search-limits-quotas-capacity. [Accessed Jun 2025].

APPENDICES

4.8. Appendix A: Data Blocks

4.8.1. Appendix A.1: example of cloud benchmarking client's configuration


```
{
  "fileWriteOperationConfigs": [
    {
      "note": "/////////////////for 1 user: File size(MB)=1,10,100 (10
iterations)//////////////////////////",
      "fileSize": 1,
      "threads": 1,
      "iterations": 10
    },
    {
      "fileSize": 10,
      "threads": 1,
      "iterations": 10
    },
    {
      "fileSize": 100,
      "threads": 1,
      "iterations": 10
    },
    {
      "note": "/////////////////for 10 user: File size(MB)=1,10,100 (1
iteration)//////////////////////////",
      "fileSize": 1,
      "threads": 10,
      "iterations": 1
    },
    {
      "fileSize": 10,
      "threads": 10,
      "iterations": 1
    },
    {
      "fileSize": 100,
      "threads": 10,
      "iterations": 1
    }
  ],


  "fileReadOperationConfigs": [
    {
      "note": "/////////////////for 1 user: File size(MB)=1,10,100 (10
iterations)//////////////////////////",
      "fileSize": 1,
      "threads": 1,
      "iterations": 10
    },
```

```
    {
      "fileSize": 10,
      "threads": 1,
      "iterations": 10
    },
    {
      "fileSize": 100,
      "threads": 1,
      "iterations": 10
    },
    {
      "note": "/////////////////for 10 user: File size(MB)=1,10,100 (1
iteration)//////////////////////////",
      "fileSize": 1,
      "threads": 10,
      "iterations": 1
    },
    {
      "fileSize": 10,
      "threads": 10,
      "iterations": 1
    },
    {
      "fileSize": 100,
      "threads": 10,
      "iterations": 1
    }
  ],

  "memoryReadOperationConfigs": [
    {
      "note": "/////////////////for 1 user: dataSizeMB=10,20,50(10
iterations)//////////////////////////",
      "dataSizeMB": 10,
      "threads": 1,
      "iterations": 10
    },
    {
      "dataSizeMB": 20,
      "threads": 1,
      "iterations": 10
    },
    {
      "dataSizeMB": 50,
      "threads": 1,
      "iterations": 10
    },
    {
      "note": "/////////////////for 10 user: dataSizeMB=10,20,50(1
iteration)//////////////////////////",
      "dataSizeMB": 10,
      "threads": 10,
      "iterations": 1
    },
    {
      "dataSizeMB": 20,
      "threads": 10,
```

```
        "iterations": 1
      },
      {
        "dataSizeMB": 50,
        "threads": 10,
        "iterations": 1
      }
    ],


    "memoryWriteOperationConfigs": [
      {
        "note": "/////////////////for 1 user: dataSizeMB=10,20,50(1
iteration)//////////////////////////",
        "dataSizeMB": 10,
        "threads": 1,
        "iterations": 10
      },
      {
        "dataSizeMB": 20,
        "threads": 1,
        "iterations": 10
      },
      {
        "dataSizeMB": 50,
        "threads": 1,
        "iterations": 10
      },
      {
        "note": "/////////////////for 10 user: dataSizeMB=10,20,50 (1
iterations)//////////////////////////",
        "dataSizeMB": 10,
        "threads": 10,
        "iterations": 1
      },
      {
        "dataSizeMB": 20,
        "threads": 10,
        "iterations": 1
      },
      {
        "dataSizeMB": 50,
        "threads": 10,
        "iterations": 1
      }
    ],

    "cpuStressOperationConfigs": [
      {
        "note": "/////////////////for 20 user, 1 thread
//////////////////////////",
        "threads": 20,
        "iterations": 1,
        "loadIterations": 1000000,
        "loadThreads": 1
      },
      {
```

```
      "note": "/////////////////for 1 user, 20 threads
//////////////////////////",
      "threads": 1,
      "iterations": 1,
      "loadIterations": 1000000,
      "loadThreads": 20
    }
  ],


  "networkStressOperationConfigs": [
    {
      "note": "/////////////////for 1 user: dataSizeMB=1,10,50 (10
iterations)//////////////////////////",
      "dataSizeMB": 1,
      "threads": 1,
      "iterations": 10
    },
    {
      "dataSizeMB": 10,
      "threads": 1,
      "iterations": 10
    },
    {
      "dataSizeMB": 50,
      "threads": 1,
      "iterations": 10
    },
    {
      "note": "/////////////////for 10 user: File size(MB)=1,10,50 (1
iteration)//////////////////////////",
      "dataSizeMB": 1,
      "threads": 10,
      "iterations": 1
    },
    {
      "dataSizeMB": 10,
      "threads": 10,
      "iterations": 1
    },
    {
      "dataSizeMB": 50,
      "threads": 10,
      "iterations": 1
    }

  ],

  "Hosts": [
    {
      "BaseUrl": "https://cloudbenchmark20250420213840.azurewebsites.net",
      "Name": "AppService"
    },
    {
      "BaseUrl": "https://cloudbenchmark41125484658495.azurewebsites.net",
      "Name": "Container"
    }
  ],
```

```
  "SleepAfterHost": 1,
  "SleepBeforeIteration": 10
}
```

4.8.2. Appendix A.2: Full schema of cloud benchmarking client configuration

```json
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "properties": {
        "cpuStressOperationConfigs": {
            "items": {
                "properties": {
                    "iterations": {
                        "type": "integer"
                    },
                    "loadIterations": {
                        "type": "integer"
                    },
                    "loadThreads": {
                        "type": "integer"
                    },
                    "note": {
                        "type": "string"
                    },
                    "threads": {
                        "type": "integer"
                    }
                },
                "required": [
                    "iterations",
                    "loadIterations",
                    "loadThreads",
                    "note",
                    "threads"
                ],
                "type": "object"
            },
            "type": "array"
        },
        "fileReadOperationConfigs": {
            "items": {
                "properties": {
                    "fileSize": {
                        "type": "integer"
                    },
                    "iterations": {
                        "type": "integer"
                    },
                    "note": {
                        "type": "string"
                    },
                    "threads": {
                        "type": "integer"
                    }
```

```
            },
            "required": [
                "fileSize",
                "iterations",
                "threads"
            ],
            "type": "object"
        },
        "type": "array"
    },
    "fileWriteOperationConfigs": {
        "items": {
            "properties": {
                "fileSize": {
                    "type": "integer"
                },
                "iterations": {
                    "type": "integer"
                },
                "note": {
                    "type": "string"
                },
                "threads": {
                    "type": "integer"
                }
            },
            "required": [
                "fileSize",
                "iterations",
                "threads"
            ],
            "type": "object"
        },
        "type": "array"
    },
    "Hosts": {
        "items": {
            "properties": {
                "BaseUrl": {
                    "type": "string"
                },
                "Name": {
                    "type": "string"
                }
            },
            "required": [
                "BaseUrl",
                "Name"
            ],
            "type": "object"
        },
        "type": "array"
    },
    "memoryReadOperationConfigs": {
        "items": {
            "properties": {
                "dataSizeMB": {
                    "type": "integer"
```

```
                },
                "iterations": {
                    "type": "integer"
                },
                "note": {
                    "type": "string"
                },
                "threads": {
                    "type": "integer"
                }
            },
            "required": [
                "dataSizeMB",
                "iterations",
                "threads"
            ],
            "type": "object"
        },
        "type": "array"
    },
    "memoryWriteOperationConfigs": {
        "items": {
            "properties": {
                "dataSizeMB": {
                    "type": "integer"
                },
                "iterations": {
                    "type": "integer"
                },
                "note": {
                    "type": "string"
                },
                "threads": {
                    "type": "integer"
                }
            },
            "required": [
                "dataSizeMB",
                "iterations",
                "threads"
            ],
            "type": "object"
        },
        "type": "array"
    },
    "networkStressOperationConfigs": {
        "items": {
            "properties": {
                "dataSizeMB": {
                    "type": "integer"
                },
                "iterations": {
                    "type": "integer"
                },
                "note": {
                    "type": "string"
                },
                "threads": {
```

```
                    "type": "integer"
                }
            },
            "required": [
                "dataSizeMB",
                "iterations",
                "threads"
            ],
            "type": "object"
        },
        "type": "array"
    },
    "SleepAfterHost": {
        "type": "integer"
    },
    "SleepBeforeIteration": {
        "type": "integer"
    }
},
"required": [
    "cpuStressOperationConfigs",
    "fileReadOperationConfigs",
    "fileWriteOperationConfigs",
    "Hosts",
    "memoryReadOperationConfigs",
    "memoryWriteOperationConfigs",
    "networkStressOperationConfigs",
    "SleepAfterHost",
    "SleepBeforeIteration"
],
"type": "object"
}
```