**PERFORMANCE ANALYSIS OF WEB-APPLICATIONS: MONOLITHIC VS CLOUD NATIVE**

**Introduction**

* 1. Background
  2. Problem statement
  3. Project Goals
  4. Motivation
  5. Limitations
  6. Report Structure

**Background:**

The first web server in the world was deployed at CERN in Geneva, Switzerland in 1990 and three years later, there were 500 known web servers [source]. Suffice to say they were all monolithic as the idea of SOA (Service Oriented Architecture) became popular around late 1990s according to [source] and Microservice in 2010s after the blog by Lewis and Fowler[source]. There monolithic architecture meant they were developed in a single code base with all the services tightly coupled and none of the services could deployed independently. And at the time, they were all deployed on premise. Finally, in 2023, according to [source], there were more than a billion websites, suffice to say number of web servers that host these websites are at least in millions.

This exponential explosion was facilitated by Amazon’s launch of AWS with Elastic Compute Cloud (EC2) service followed soon by Google’s Google App engine (2008) and Microsoft’s Azure (2009). The could services transformed the landscape of web-server deployment. After this point, business, individual, or organizations no longer needed to host their software on premise. More importantly, they could make whatever services they wanted to provide to end-users at very low initial investment.

From here on, the demand for better and faster service continue to rise, leading to the innovation of new type of application called cloud native. They are built using microservice architecture and take full advantage of cloud computing. In microservice architecture, applications are broken down into multiple services and are developed independently. Unlike in monolithic applications, these services can be deployed independently as well. Netflix was one of the first companies to move away from its monolithic architecture in 2009. However, at this point, the concept of microservice was not even known. In fact, it was in 2011 when the term was first used and was official announced at the 33rd Degree Conference in Krakow in 2012. And finally in 2014, the architecture was popularized by Lewis and Fowlers’s blog [source].

**Problem statement:**

[Some of the sentences in the following paragraph are one to one copy from the source. Remember to present it in a different way.]

Most of the world’s web applications are monolithic in nature. They don’t take full advantage of the cloud computing. More companies would benefit from moving toward more cloud native applications.

**Project goals:**

[Some of the sentences in the following paragraph are one to one copy from the source. Remember to present it in a different way.]

Monolithic applications were industry standard for a web application from 1960s until recently. They are still perfect for small to medium size applications that don’t have large user base. However, internet is present everywhere these days. The number of users for an application can reach from thousands to millions to hundreds of millions. On top of that, technologies and market keep on changing regularly. With these new factors, monolithic applications have started to hit bottle necks. Updating and adding new features to increasingly changing market presents a problem especially if the monolithic application is already big. It might introduce a new bug, it may take a long time to introduce a new feature, it might take a long time to test give the size of the application, on top of that a single undetected bug in production can take down the entire application. And how are you to know if you user size grows in the future. It took Netflix two years to change their monolithic applications to cloud native. Should they have stared with microservice application straight away. But what if the Netflix as an application failed if they had started as cloud native applications.

An alternative that appeases these problems is of course cloud native applications with microservice architecture. You can introduce features quickly, you decrease cognitive overhead, and multiple teams can work on the same application simultaneously give the principle of loose coupling. They can scale horizontally as well as vertically much easily monolithic applications. This sounds like exactly what we needed to solve the problem of monolithic architecture. Yet, it has its flaws. Initial design stage takes longer, is just too complicated for small to medium applications with small user base, is costly on short term.

Given these complex properties of each type of application type, to a normal software developer, it isn’t exactly obvious what is the best choice. Individuals and companies spend hours trying to understand what is best for their specific needs.

**Project Goals:**

[The following paragraph is a brain shit]

With this project, I aim to clarify what each type of application provides along with their limitations along with their performance capabilities along with technologies that best suit each of the feature in the applications. [Here insert the method that you are going to use performance test it]. I plan to develop same application using monolithic architecture and another use microservice architecture. One will be deployed in traditional style while other will using cloud native approach. These two applications will be built using exact same programming languages as to remove the idea that performance boost might have been due the underlying programming language.

With this project, I aim to clarify the performance abilities of monolithic applications and cloud-native applications. To do this I will be developing and deploying same application twice, one using monolithic architecture and other using microservice architecture. Application with monolithic architecture will be deployed on a virtual machine in the cloud as to make the application monolithic and application with microservices architecture will be deployed on containers in the cloud as to make it a cloud-native application. The application will be developed using same technologies to make the comparison as fair as possible. Finally, to clarify the performance abilities of the application, I plan to perform load test on them using Jmeter with varying number of threads.

**Motivation:**

It is already known that application with large number of users or large application generally benefit from being cloud native. This seems to be the reason why most tech giants have been transitioning toward cloud native applications.

But do all the applications need to be cloud-native because of its benefits and its ability to perform better. To me it isn’t exactly obvious given the complexity of designing or transitioning to a cloud native application. I plan to load test my web applications in various scenarios in an attempt mimic different type of request given by different web application. This should give a better picture if an application needs to be cloud native or will function totally fine being monolithic.

**Limitation:**

Due to the time frame and budget for this project, the applications that I built contain 5 services. Thought this makes the applications non-trivial, the result of performance analysis conducted in these applications shouldn’t be the only thing to look at when making the decision for applications with large numbers of services. That being said, it should give general idea of performance for each type of applications.

**LITERATURE REVIEW:**

**2.1 The advantages of monolithic architecture**

**2.2 Challenges in large monolithic applications**

**2.3 Benefits of cloud-native**

**2.4 The shortcomings of cloud-native**

**2.5 Scalability**

**2.7 Security in Cloud-Native and Monolithic applications**

**2.8 Monitoring Cloud-Native and Monolithic applications**

**Monolithic Applications:**

* Monolithic applications are called monolithic because they are built using monolithic architecture, which is designed to deployed as a single, indivisible unit. In this architecture, all components of the application are tightly integrated and run in single process space. This approach simplifies development, deployment, and management processes but also couples these components closely together, making applications act as single monolithic block. Traditionally, all monolithic applications were deployed in premise but nowadays they are hosted on cloud due to cloud computing.

**Advantages of Monolithic applications:**

* Simple to develop and deploy
  + Given that whole application is built to be run on the same process. The process of developing and deploying is lot easier. The developers have to constantly test the applications functionality and ensure that they are what is required from the application, this is lot easier to do in monolithic applications because everything to do with the application can be invoked on single run.
* Ease of debugging and Testing
  + Testing is easy in monolithic because all you have to do is think about the single process. (I ACTUALLY DON’T KNOW WHY TESTING IS EASIER)
* Consistency:
  + Less duplication of the code. Easier to enforce coding practices. [FIND AN EXAMPLE]
* Less Operational Overhead
  + Things such as single database and tight coupling means that there is no need to design how different components will interact. Of course, this might lead to applications with badly structured code base.
* Small Monolithic applications are easy to maintain.
  + Small applications will not have complex design of how the different components will interact mainly because there are very few components. Tracking how they interact is much easier this way.

**Disadvantages of monolithic applications:**

* Scalability challenges
  + One of the main reasons, monolithic applications are going out of fashion is because the can only be scaled vertically, more about it later.
* Difficulty in Adopting New Technologies
  + An application has many parts each part might function differently and there might be better technologies to each part more efficiently and effectively. However, with monolithic you can only use on technology to write it.
* Increased Risk of Deployment
  + During re-deployment, sometimes it is likely that you tampered with a function that messed up the functionality of the other.
* Longer Development Time as the application grow
  + Developer need to spend more time understanding the application as a whole as slight change in code can affect other functionality.
* Limited Flexibility
  + Change to one part of the application might make it so that another part of the application needs to be updated as well as the application is tightly coupled.
* Continuous deployment becomes more challenging
  + Once again we come to the part where a single bug or change affect the whole applications.
* More challenging to make applications reliable and fault proof
  + A singular bug can take down the application.
* Team collaboration becomes more difficult as the code base grows
  + As team grows which happens if the application grows, it is very difficult to segregate the duties as the application is tightly coupled give that they run on the same process.

**Cloud Native Applications:**

* The rise of cloud native applications can be attributed to the popularity of cloud computing and the rise in demand for faster services. They are built using microservice architecture. In this architecture, applications are broken down different services. Each service is developed and deployed independently. Services communicate with each other through a developer defined APIs. Below you will find advantages and disadvantages of cloud-native applications.

**Advantages of cloud-native applications:**

* Highly scalable
* High resilience and fault tolerance
* Faster time to market
* Cost efficiency
* Flexibility and Portability
* Improved developer Productivity
* Optimized for cloud

Containerization

One of the main advantages of microservices architecture is it allows technology heterogeneity. Each service can be developed using different technologies (e.g. programming languages). This means developers can either chose a technology or a stack that is already known to work well to provide the functionality that the service provides, or they can choose to use the technology that they are most familiar with. This added with the fact that developers mostly only need to think within context of a service allows a service to be efficiently developed while reducing the risk of introducing a bug.

Second advantage of microservice is better resilience. Due to the use of containerization fault/bug is usually isolated in the service that it arises in if it doesn’t need to communicate with other services. And in the case where it is necessary that two or more services need to communicate, if proper fault tolerant mechanism is implemented, fault can still be isolated to the service it arose in. This means application as a whole will continue to function even though some of its service might not be available.

Third advantage of microservice is better maintainability and availability. It allows to take full advantage CI/CD pipelines which automate the building, testing, and deploying process. This means it is very unlikely that a bug passes through to the production if testing framework is robust. On top of this, when adding a new feature, the whole application doesn’t have to be stopped. Only the service to which new feature is added needs to be redeployed. This ultimately means new features can be brought to the market rapidly with very low risk.

**Disadvantages of cloud-native applications:**

* Complex
* Skill Gap
* Security Concerns
* Increased Operational Overhead: Initially, the transition to cloud-native can increase operational overhead. Setting up CI/CD pipelines, orchestration platforms like Kubernetes, and monitoring tools takes significant effort and time.
* Cost Management Challenges: While cloud-native applications can be cost-effective due to their efficient use of resources, managing costs in the cloud can be complicated.
* Transition and Migration Challenges: Moving existing applications to a cloud-native architecture can be challenging and resource-intensive.

Since the logic of an application must be distributed among different services, it takes longer to design a cloud native application. On top of that, if you want to take advantage of it, it is necessary to set up CI/CD pipelines and need to set monitoring tools which makes the development and deployment even longer. “An article by Taibi et al. [18], reports that microservices-based applications require nearly 20% more eort compared to the eort required for initially developing a monolithic application.”

Moving existing applications to a cloud-native architecture can be challenging and resource-intensive.

**Scalability**

Scaling is increasing ability of an application to provide better and faster service or to increase its capability to handle work (requests). It is very crucial concept if you want to develop an application that is efficient, reliable and can handle increasing number of users, data volume, and computational intensity. Scaling is usually achieved either by vertically or horizontally scaling a system.

**Vertical** scaling refers to addition of the resources (e.g. more CPUs, more Memory, more storage) in environment the application is running in to handle the increasing load.

**Horizontal** refers addition of replicas of the application/service behind a load balancer to handle the increasing load.

**Scaling monolithic applications**

* Given the fact that monolithic applications run on single process space, it is very straightforward to scale them vertically as there will be only one server that needs to be maintained. However, the same nature of the monolithic is also the reason why this type of scaling is inefficient. Since all the services are tightly coupled, the scaling needs to consider all of them. In other words, services that didn’t need scaling also get scaled as such high capability hardware is required for vertical scaling. This becomes problematic the more the application needs to be scaled because the cost of vertically scaling rises exponentially as high-capacity hardware are very expensive.
* This is where horizontally scaling comes through. Many replicas of a singular application can be run behind a load balancer which allows it to handle more load. Though same problem of inefficient resource utilization continues to persist however the cost of scaling doesn’t increase exponentially.

**Scaling cloud-native applications**

* One of the reasons the cloud-native applications have gained popularity is because of their ability to scale horizontally very efficiently. Once monitoring tools are set, it is easy to find out which service needs to handle more request, and another replica of the service is launched for specific amount of time and placed behind a load balancer. Finally, this process can be fully automated thanks to the existence of tools like docker and Kubernetes, making it more cost effective.

**Security in Cloud-Native and Monolithic applications**

Secure application generally guarantees confidentiality, integrity, and availability. Innovation of cloud native application has eased some problems but exasperated some others. Below I will try to put forth the argument about which is easier and difficult to achieve in what type of application.

For monolithic application, there is only one server to defend. This makes the task of defending monolithic applications easier than a Cloud-native. Given that monolithic applications have been around for a long time, there are lot of techniques to defend the applications.

But one place it shies away from cloud-native is availability. Because microservice can be implemented with circuit breaker pattern, if a bug takes down a service, other parts of the system are still available. Though it must be stressed due to less cognitive developer overhead the likelihood of bug passing through to the production is low compared to a monolithic application.

Confidentiality and integrity are very difficult to achieve in cloud-native applications. Service being independent of each other allow technology heterogeneity but it also makes the attack space larger. The process of defending has gone from one server to many, which is a problem. One of the additions to the attack space is network between the services. If a service is compromised, it can send malicious request to another service. One of the solutions to this is to use TLS to protect the communication between the different services. The problem however is how to coordinate and update the distributed authorization rules for request from users and microservices.

In conclusion, individual service within cloud-native applications can be defended in the traditional way the monolithic application is defended, the problem comes in the space where the services communicate with each other.

From: [Security in microservice-based systems: A Multivocal literature review - ScienceDirect](https://www.sciencedirect.com/science/article/pii/S0167404821000249#bib0022)

For: [Authentication and Authorization of End User in Microservice Architecture - IOPscience](https://iopscience.iop.org/article/10.1088/1742-6596/910/1/012060/meta)

**Monitoring:**

Monitoring an application is a critical aspect of providing a functional application to the end users. It involves collecting and analysing information about the application and system it is deployed in as to decipher its health and how well it is performing. Very often, it is possible to find issues, especially related to workload, within the application if it is monitored well. This can ensure that application is available most of the time.

Both types of applications can be monitored. Monolithic applications are monitored as a whole, without any break down of work performed per service basis. This, though useful in preventing system failure, doesn’t give information to aid in making decisions about services present within the application. For example, it is not possible to make an informed from monitoring a monolithic application as to which service should be optimize for better performance. Cloud-native applications, on the other hand, can be monitored per service basis, which arms administrator with more precise knowledge about the performance and health of the application as well as making informed decisions about the services.

**Cloud-Native:**

**Monitoring Cloud-Native and Monolithic applications**

**History of performance review:**

**LITERATURE REVIEW:**

**PERFORMANCE ANALYSIS:**

There have been many studies for the performance analysis of Monolithic vs Microservices architectures for various scenarios.

In 2018, using Jmeter as the benchmarking tool, Al-Debagy and Martinek conducted a performance test on an application built using monolithic and microservice architecture. There applications were developed in JHipster using Sprint Boot (java) and Angular JS frameworks while Jmeter was installed remote client. There results were as follows. During load testing there was no significant difference in performance between the different architecture. Meanwhile, during concurrency testing, monolithic architecture showed a better performance in terms of throughput by 6 % on average. There results showed an obvious winner, the monolithic architecture. However, this might have due to the fact microservices application was deployed locally where its full potential can’t be utilized.

From: <https://arxiv.org/abs/1905.07997>

In 2020, [reference] assessed the performance of two same web applications, one developed using monolithic architecture while other using microservices. Monolithic application operated on a virtual server with KVM, and microservice architecture ran in containers. The study found that microservices architecture leverages CPU and packet transmission more efficiently than monolithic architecture, meaning microservice architecture resulted in better system performance than monolithic.

FROM:

https://www.researchgate.net/publication/343805251\_From\_Monolithic\_Systems\_to\_Microservices\_A\_Comparative\_Study\_of\_Performance

**Glossary of technology:**

**3.1 Web Application:**

**Nodejs**

**Reactjs**

**3.2 Monolithic**

**Virtual Machine**

**Azure mysql database**

**MongoDB atlas**

**3.3 Cloud-native**

**AKS**

**3.4 Performance Testing**

**2.6 Data persistence layer**

**Data Persistence Layer**

* **SQL**
  + **Transaction model**
* **Scaling SQL**
* **NoSQL**
  + **Transaction Model**
* **Scaling NoSQL**
* **NewSQL**

**Glossary Of Technologies**

**Technologies**

1. Nodejs with TypeScript
   1. Nodejs is open source javaScript runtime environment that runs on V8 javaScript engine. Meanwhile, Typescript is a superset of javaScript which allows static typing unlike plain javaScript. Nodejs with Typescript combines all the features of Nodejs and Typescript to allow developers to program powerful, scalable backend systems.
2. React with TypeScript
   1. React is an open-source frontend JavaScript library which is used to build interactive UIs. It’s integration with Typescript allows it to be statically typed.

**Monolithic**

1. VM:
   1. Virtual Machine are piece of software that simulate a computer system with full operating system. Multiple VMs with different operating system can be ran on one physical machine which is one of the reasons for their widespread use.
2. Pm2
   1. Pm2 is a production process manager for Node.js application. It has built-in load balancer and is equipped with an ability to keep application running forever and to reload them without down time.
3. MySQL DBMS (Database Management System)
   1. MySQL is an open-source relational database management system originally developed by MySQL AB. It is queried using SQL (Structured Query Language).
4. MongoDB Atlas: It is a fully managed cloud database service provided by MongoDB. It allows developers to deploy, operate and scale MongoDB databases without having to manage underlying infrastructure.

**Cloud-native**

1. Azure Cache for Redis
   1. It is a fully managed in-memory database that uses Redis to store and retrieve the data. Its main purpose is to be used as cache.
2. Azure Container Registry (ACR)
   1. Private docker registry to hold and manage docker images.
3. Azure Container Apps
   1. It is a fully managed application platform. Powered by Kubernetes, it allows developers to deploy apps from code or container without orchestrating complex infrastructure. It operates in serverless environment, meaning that it can be configured to scale automatically.

**Performance Testing:**

1. Azure Load Testing
   1. A fully managed load testing service for generating high scale loads and identifying performance bottlenecks. It allows the use of Jmeter script to create complex test suite. It can also collect detailed metrics of application server so long as it hosted in the Azure.
2. Ms Excel
   1. A spreadsheet software included in the Microsoft Office suite of applications. It provides functionality to comprehend and visualize data with ease.

**DESIGN AND IMPLEMENTATION:**

Same web application will be produced for each type of application.

The web application's aim is to provide users with an ability to capture the metrics for their daily activity, track mood throughout the day, capture their workout metrics, and finally, see the trend in the data that they collected. This would allow users to see what and how they had performed in the past as well as see if any activities were having good or bad effect on them over time. The main beneficiaries of the app would be people who want to optimize their health and performance. To provide this to the user, the main features will be to make a journal entry (one every day), workout entry, and a graphical representation for all the data they have accumulated so far. Given this, users will be able to make changes or be confident on what they are doing based on evidence. The app will be built using NodeJS with typescript for backend and React with typescript for frontend. The reason for this choice is because both NodeJS and React are extension of JavaScript as such using both of them will make development process lot easier.

**4.1 Monolithic Application**

**Design**

The developed monolithic application is split into three tiers – frontend, backend, and database. Frontend and backend sit of separate repository. The file structure of the repository is given below. The backend is further divided into different parts – Model, Controllers, Middlewares, and Routers, explicitly divided into different folders as shown in fig. If the application was simple, it wouldn’t have been necessary to separate it into further entities listed above, but in my case, separating them into different entities helped me develop and debug application at much faster pace. Each entity has different functionality: Model holds the codes necessary to interact with the database, controllers held API’s that handle the request from the client, middlewares checked the validity of request before passing it to the controllers, and finally routers investigated the URL to diverge them to the correct controllers.

**Hosting Model**

The monolithic application was hosted on one ubuntu server 20.04LTS x64 Gen2 with 2 vCPUs and 4GB of RAM (1x standard B2s) located in UK south. It ran pm2 to keep the server running and re-spawn it if it crashed. For database, an instance of Azure Database for MySQL flexible server was deployed on the same region (UK south) and a MongoDB cluster was deployed in EU-Ireland using mongo atlas.

**4.2 Cloud Native**

**Design**

Despite the option of being able to create separate repositories for each of the service, a mono repo was used to build and keep track of the cloud-native application. The structure of the entirety of the cloud-native repo could be seen on fig. Each service was light, and cloud have been developed without breaking down into different parts, but for the sake of keeping the code base organized, it was broken down into the same entities like in Monolithic Application. However, there was on more entity introduce for some of the services, particularly in notifications, logs, exercise, and dashboard. The new entity was either gRPC server for logs and exercise service and gRPC client for dashboard and notification service. gRPC server and client were used to allow the communication between different services. Each of the services had their own docker file as well to build their image to run them on containers.

**Hosting Model**

All of the services of the cloud-native applications were hosted using Azure Container Apps. More specifically, each service was hosted in an isolated instance of an Azure Container Apps with 4v CPUs and 8GB memory. All of the services spawned with one replica which had access to 0.5 CPU and 1 GB of memory. With increasing load, 6 more replicas could spawn each with access to same amount of resources.

The services communicated with each other through either API requests or gRPC server and client. In order to facilitate the interaction between the services, Dashboard, Exercise, Logs, and Notifications service had port 8082 opened for receiving and sending gRPC requests and responses and had port 3000 open for receiving client request while user service only had port 3000 opened.

The requests were forwarded to the services using Azure’s internal Load balancer which came out of the box for each of the instance of Azure Container Apps.

And finally, each service had its own database. User service connected to an instance of mysql database hosted on Azure. Exercise service, and Logs service connected to separate mongodb database hosted in mongo atlas. While Dashboard service, that communicated with Exercise and Log service, was deployed with an instance Azure Cache for Redis to cache the data received from them in order to decrease the frequency of communication between the services. Overall, this design follows the 'Database per Service' pattern, which is a common practice in microservices architectures to ensure loose coupling and independent evolution of each service.

**Performance Testing**

Setup

To benchmark the performance of the two applications, two different instances of Azure Load Testing, one for monolithic application and other for cloud-native application, were deployed southern region of UK. The main reason for the choice of Azure Load Testing was due to its ability to use script developed for Jmeter. Jmeter is one of the popular open-source software specifically designed to load test web servers.

In Jmeter, the ratio of request type is controlled by using its Throughput Controller module. As such each load test script had a Throughput Controller module for GET, PUT, and POST request type. Further, the request to be sent to the server was randomly chosen from the request type, meaning each request under the specific request type had equal chance of being chosen.

Finally, the test scripts were designed and developed in such a way that each request would contain a cookie to let the server know which user was sending the request. This was accomplished using CSV Data Set Configuration, JSR223 PreProcessor, HTTP Header Manger, and HTTP Cookie Manager module of Jmeter. Also, if any of the GET, PUT, and POST request depended on the data previously sent by the server, they were adequately handled using Jmeter’s preprocessor and post processors lister modules. For example, a GET request of type /logs/get-log-by-id/:id with id parameter would have an id randomly chosen based on the previous request to the server. This was done for two reasons, one to create an environment similar to use cases and second to ensure that requested id was always in the database which would ensure no error was responded to the load testing server.

**Testing Strategy**

The goal behind the performance analysis was to understand if an application with specific workload type (read-heavy to update-heavy) benefited from being either monolithic or cloud native. To gain this understanding, 11 different test cases with varying HTTP request ratio were developed to be ran on the two applications developed for this project. Further, each test case contained 7 load tests, each with same request ratio but with increasing number of threads: 1, 2, 4, 8, 12, 16, and 20. This was done to simulate applications with varying number of users.

Each of the load test was ran for 4 minutes to ensure that the servers were pushed to the maximum capacity. And following each test, databases associated with the server/s were reset to only contain seed-data. In the case cloud-native, this meant the cache database was flushed.

The total number of load test that were run was 154, 77 for each application. The result of these load tests are presented in the following chapters.

On top of throughput and latency, CPU utilisation data was retrospectively collected for each workload run through the use of Azure metrics. This will be further introduced in the analysis of the first test case in the following chapters.

**PERFORMANCE TESTING RESULTS**

The content of this chapter will be presenting, analysing, and finding key trends in the data obtained from the performance testing. A graph, for both monolithic and cloud-native load test, will be plotted for comparing average throughput (x-axis) and average response time (y-axis) with increasing threads for each of the test cases. The trend of the monolithic will be presented in red while cloud-native will be in blue. Additionally, in order to give better idea of the data, a table with Response Time and Throughput for each of the thread in the test cases will be presented as well.

The load test of ratio 100:0:0 (GET:PUT:POST) was designed to simulate request load of an application that primarily receives read operations. Example of such an application is search engines, which has to retrieve large number of user comments and posts.

For this request type, the cloud native applications response time remained consistently low with an average of 19 milliseconds even with the extreme increase in throughput between different number of threads with an average of 435 request/sec overall. Its performance started to decrease in a manner that could be significant at 20 threads where it received a response time of 24 milliseconds with a throughput of 801 which was a decrease in performance compared to 16 threads where its response time was 19 and throughput was 800. However, just these datapoints can’t be used to conclusive state that cloud native application would reach saturation point at 20 threads. It is necessary to conduct few more load test with more threads.

Monolithic application, on the other hand, had an average response time of 69.28 milliseconds with an average throughput of 251.22 requests/sec. Its maximum throughput, 393 request/sec, was achieved at 16 threads while its best performance was achieved at 12 threads where the throughput was 391.95 request/second with response time of 65 ms. Any additional threads after 12 would result in depreciating performance. This is primarily due to the fact that it was close to running out of CPU resources as can be seen in the following figure.

A table with numbers and letters

Description automatically generated

The figure shows that CPU utilisation of the cloud-native application was significantly higher than monolithic application’s which is justifiable because cloud-native application performed significantly well for all the different number of threads. What is interesting within this graph is that cloud-native application was able to use more of its CPU resources at the lower number of threads compared to monolithic application. For example, cloud native application was able to use 1.27 CPU cores at 4 threads while monolithic application was only able to use 0.65 CPU cores even though it had access to more of it. Essentially meaning that cloud-native application can better utilize the resources available to it compared to monolithic application in this scenario.

**COULD BE USEFUL.s**

As can be seen on figure \_\_\_, for this request type, the “winner” is cloud-native application. Cloud-native applications response time remains consistently low with an average of 19 milliseconds even with the extreme increase of throughput meanwhile the monolithic application’s response time continues to increase with an average of 69.28 and reaches a bottleneck around 400 requests per second.

This leads me to conclude that monolithic applications aren’t suitable for

monolithic application reached the bottleneck around 400 throughput with response time increasing exponentially. This type of application seems to perform well with small number of users as it can be seen both cloud-native and monolithic applications have close response time when the throughput is low (low throughput meaning, small number of users).

The obvious winner of the comparison is cloud-native applications, its response time remained consistent with the increasing load. This was likely because as the load increase the cloud-native application was able to spwan more replica of itself to handle the incoming load. Meanwhile, monolithic applications performance degrades as the throughput increases.

The load test of ratio 98:1:1 (GET:PUT:POST) was designed to simulate request load of an application that primarily receives read operations but also includes some update and insert operations. Example of such an application is local bus transportation system where user will constantly access the bus routines, and admins might make slight change or add different routines every so often. A table with numbers and letters

Description automatically generated

Once again, cloud-native application outperformed monolithic application. Monolithic application had an overall average response time of 67.57 milliseconds with moderately increasing throughput till it reached a bottleneck at 16 threads while cloud-native application had an average response time of 19.28 milliseconds with significant increase in throughput from 1 thread to 20 threads. Monolithic application received a maximum throughput of 380.35 with response time of 68 ms for 16 threads. Meanwhile, cloud-native received maximum throughput of 864.21 request/sec with response time of 24 millisecond.

CPU utilization of the cloud native is higher than that of the monolithic application. The similar behaviour to the scenario one is shown by the CPU in this scenario.

With this it can be clearly said that monolithic application performs worse than cloud native application, but it is also using less CPU.

This load test is designed for applications that don’t need addition of new data by users, but users constantly access the data and may make update to the existing data. An example of such an application would be applications tracking old projects.

In this scenario, the overall average response time of monolithic application was 43.71 milliseconds comparatively low compared to the previous two test scenarios, but its average throughput of 251.39 was very close to the previous two scenarios. Cloud-native application’s average response time, on the other hand, increased slightly reaching 21.14 milliseconds though its average throughput of 417.90 request/sec. As it can be seen in figure \_\_\_ if more threads were added to load test of cloud-native application, we might have been able to see its bottleneck.

Looking at the CPU utilization, the cloud-native application on average used 2.256 CPU cores while monolithic application used 1.001 CPU cores. Cloud-native application used 2.253 times more CPU resources than monolithic application to achieve 1.66 times better average throughput and 2.170 times better average response time.

In this scenario,

A table with numbers and letters

Description automatically generated