National College of Ireland

Improving the scalability of Node.js applications in the cloud by integrating parallel processing and multi-threading, followed by a performance assessment across AWS, GCP cloud platforms.

MSc Research Project
MSc in Cloud Computing

Rohith Addula
Student ID: x22192867

School of Computing
National College of Ireland

Supervisor: Shreyas Setlur Arun

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Rohith Addula
Student ID:	X22192867
Programme:	MSc in Cloud computing
Year:	2024
Module:	MSc Research Project
Supervisor:	Shreyas Setlur Arun
Submission Due Date:	07/11/2024
Project Title:	Improving the scalability of Node.js applications in the cloud by integrating parallel processing and multi-threading, followed by a performance assessment across AWS, GCP cloud platforms.
Word Count:	XXX
Page Count:	34

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	2nd December 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Improving the scalability of Node.js applications in the cloud by integrating parallel processing and multi-threading, followed by a performance assessment across AWS, GCP cloud platforms.

Rohith Addula
x22192867

Abstract

The single-threaded, non-blocking asynchronous architecture of Node.js applications made it more suitable for handling I/O operations and concurrent requests efficiently. However, this architecture constraints the Node.js apps by limiting the scalability in handling CPU-intensive workloads. This research aims to address this with a custom load-balancing algorithm designed to handle the task distribution across the available resources by prioritising the tasks based on the resource availability. This approach ensures efficient resource utilisation and workload distribution by optimising the performance and scalability. The proposed load-balancing algorithm is integrated into a micro-service Node.js application with different endpoints enabled to perform the experiments with different workloads.

The application was deployed across Amazon Web Services (AWS) and Google Cloud Platform (GCP) as containerised and non-containerised applications to evaluate the proposed solution. The performance metrics such as CPU utilisation, memory usage and throughput are collected after every experiment with different workloads. The experimental results shows that custom load-balancing algorithm optimised scalability by 35% on average when compared with default load-balancing mechanism in Node.js application. The results shows that in containerised instances AWS outperformed GCP by 18% in handling CPU-intensive tasks showing that AWS is more suitable to handle such tasks. This research also identifies optimal deployment strategies for high performance applications in the cloud environments. This research also demonstrates the capabilities of Node.js applications in high performance computing (HPC)

1 Introduction

Cloud computing provides an extensive collection of resources that are instantly available when needed. To optimize performance, reduce costs, and save energy, it's important to use resources efficiently. Resources are critical. Cloud-based applications are rapidly expanding and enabling real-time corporate growth. Cloud apps typically experience significant user activity and concurrent requests. Single-threaded systems may struggle to handle massive requests, leading to bottlenecks and poor application performance.
Parallel computing addresses these issues by splitting up the work and completing it simultaneously across numerous processors. This distribution optimizes CPU utilization for

better application performance, allowing for more simultaneous requests and a smoother user experience.
JavaScript has rapidly gained popularity in areas such as Over-The-Top (OTT) platforms, social media, and e-commerce, driven by the rise of cloud computing. Cloud apps must manage a high volume of queries while maintaining performance. JavaScript runs in a single thread, is non-blocking, and asynchronous.. Its essential feature, the Event loop, enables it to handle asynchronous tasks efficiently. However, there are restrictions with cloud platforms. Node.js relies on the event loop to handle asynchronous activities efficiently. When asynchronous processes are carried out concurrently, such as network requests and database communications, event loops are effective in managing I/O operations. This allows programs to execute further requests without waiting for previous processes to complete. An event loop initiates a callback function to complete remaining operations after the first ones. The event loop's single-threaded nature makes it difficult to finish CPU-intensive processes. Large data processing and intricate computations may cause the thread to stop until the tasks are finished. Until the current operation is completed, the other operations will be put in line. The application's performance will be impacted by this. Optimizing Node.js applications' speed in cloud environments is essential for managing growing user requests and computationally demanding tasks. Some popular techniques to improve the scalability of a Node.js application include spinning up instances quickly through containerization or deploying multiple instances. However, even though these techniques can scale the application, they still have an impact on performance when a computation-intensive task is running. These techniques increase the application's horizontal scalability by adding more instances, but they don't enhance performance for CPU-intensive operations.

This issue can be resolved by allocating the task to the processors and threads. By putting parallel processing and multi-threading into practice,the Node.js program optimizes performance by offloading work from the main thread, allowing for more requests per unit of time. In a single instance, this method can maximize resource utilization. Analyzing the performance of the Node.js application across several cloud platforms once parallel processing and multi threading have been implemented can yield useful information for application performance optimization. Applications can be hosted on a variety of cloud providers' resources; assessing how well an application performs in various environments will assist in selecting the best environment for the needs of the business. The purpose of this research is to build multi threading and parallel processing in Node.js applications and assess how well they run across various cloud platforms.

The necessity to optimize resource utilization and enhance application performance by offloading CPU-intensive tasks via parallel processing and multi-threading served as the impetus for this study. Optimizing Node.js applications' performance and scalability in a cloud setting is the aim. With an emphasis on Node.js microservices, this study seeks to determine load-balancing techniques and parallel processing algorithms as well as to create effective and optimized algorithms for allocating jobs and incoming requests while making efficient use of available resources. In addition to performance optimization, this work focuses on creating algorithms that use less energy.

2 Background and Motivation

The main idea of this research is to enhance and measure the Node.js application performance in cloud platforms like AWS and GCP. Even though Node.js is highly efficient in handling I/O requests like 1000 requests per second it fails to demonstrate high performance in handling CPU-intensive tasks by blocking the main thread. In this research, a custom load-balancing logic is designed and equipped with the Node.js application to enhance the performance and analyse the performance in the cloud platforms.

3 Research Question

How can parallel processing and multi-threading techniques can help to optimize the scalability and performance of Node.js applications in handling CPU bound task and I/O operations together.
As part of this research, this study will also address the following sub-questions.
Trade-offs and Challenges:

• What are the trade-offs and challenges associated with parallel processing and multi-threading in Node.js applications, particularly within cloud environments?

Cloud Platform Variations:

• What is the impact on performance of Node.js applications after implementing parallel processing and multi-threading techniques and how does the application performs in multiple cloud environments like GCP and AWS as containerised and non-containerised applications.

4 Related Work

To improve the scalability of Node.js apps, we critically examined earlier methods and evaluated how they might further this study.
In their study "Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js," K. Lei and Tan (2014) used systematic benchmark and scenario tests examine the speed of PHP, Python, and Node.js in handling high concurrent requests and large volumes of data. According to the research findings, Node.js's non-blocking design allows it to handle I/O activities more effectively than PHP and Python. Node.js works well, nevertheless, for apps that don't need sophisticated backend processing. In a study titled "Comparison of Node.js and Spring Boot in Web Development," et al. (2023) used Apache JMeter to test the performance of Node.js and Spring Boot applications. They found that Node.js is not suitable for applications which involve multithreading and CPU-intensive tasks Node.js presents issues in managing computation-intensive operations due to its single-threaded, asynchronous, and non-blocking architecture. M. Patrou and Dawson (2020)conducted a study on the software evaluation methodology of Node.js parallelism under "Variabilities in Scalable Systems," which describes how Node.js provides modules that enable the creation of new instances (either threads or processes) to manage work in parallel without interfering with the main event loop. The research looks at several Node.js modules that facilitate multithreading and parallel processing to increase the scalability and speed of Node.js applications to conduct compute-intensive activities. This can be done by using worker threads and child process modules.

In a research conducted by Tanadechopon and Kasemsontitum (2023) to compare and evaluate the performance of API services developed in PHP, Python, Node.js and Golang. This study focuses on metrics such as response time, CPU utilisation, throughput by conducting experiments with different workloads. The results shows that Node.js demonstrated competitive performance in handling concurrent requests and Golang outperformed Node.js in handling CPU-intensive tasks. This study highlights the efficiency of different programming languages.
A study conducted by Putu Agus Eka Pratama et al.'s Pratama and Raharja (2023)compares the performance of different Node.js frameworks like Fastify, Express and Hapi across different virtualization environments like Docker, VirtualBox and Podman. This research evaluates the request latency, throughput by using Node-Bench tool. The results shows that Fastify frame works delivered good performance when integrated with Sequelize ORM in Docker and Podman. This study also suggests that developers should consider Fastify framework to improve performance of the Node.js applications.
Various modules for Node's parallel processing. The architectures of JS are contrasted in both performance and functionality. By examining how these modules manage computationally demanding activities, communicate, and share memory, the analysis sheds light on their scalability. Regarding the capabilities and performance variations of several Node.js modules designed for parallelism, the study provides insightful information. It suggests that no single module is capable of outperforming every other module in every scenario. The application's performance may differ based on the particular requirement. The concept for future research on assessing Node.js modules in distributed and more computationally demanding settings comes from this study. The performance metrics and evaluation techniques for Node.js parallelism modules may be significantly improved with the aid of this next work.
M. Patrou and Dawson (2019) study, "Scaling Parallelism Under CPU-Intensive Loads in Node.js," evaluates the effectiveness of various parallelization strategies in Node.js applications by executing CPU-intensive loads. It evaluates Node.js's multi-process and multi-thread methods as well as its scalability across different contexts. The goal of the study is to identify performance trends and provide deployment recommendations using a complex methodology that includes, among other things, Docker containers for distinct modules, separate apps for each Node.js module, and use of Node.js and WebWorker. -Threads, Cluster, and Child Process modules for parallelizing a compute-intensive operation. Along with measuring performance parameters like execution time, CPU consumption, speedup, and cache misses, it also performs statistical analysis using p-values to identify performance parallels and discrepancies. The study's findings indicate that multi-threaded approaches typically outperform multiprocessor approaches, particularly in CPU-intensive jobs where WebWorker-Threads has demonstrated the best performance to date. As workloads grow or more instances are produced, the performance differences between multi-threading and multiprocessor approaches become increasingly noticeable. The impact that operating environments have on the effectiveness of parallelization techniques is also examined in this research.
JavaScript's Web-workers made parallel processing possible by enabling asynchronous communication across worker threads. This method has various drawbacks when it comes to employing shared memory, which necessitates more work for data distribution.
In their 2016 study "GEMs: Shared-Memory Parallel Programming for Node.js,"I. Chaniotis and Tselikas (2015). investigated how to use parallel programming in Node.js, where shared-memory parallel processing is accomplished in Node.js applications using the Gen-

eric Messages technique. By fusing shared memory usage with message passing strategies, the GEMs increase the potential of parallel computation. This study covers various GEM kinds, including read-only, owned, portioned, atomic-LK, and atomic-STM. Web-workers can access shared memory in a controlled manner due to GEMs' internal functionality. Using shared memory improves parallel processing performance and expands Node.js application possibilities. In order to assess the potential of Generic 6 Messages in enhancing the parallel processing capabilities of Node.js applications, experiments are conducted on an Ubuntu 14.04 server. The results demonstrate that the GEMs approach performs better than the conventional Node.js parallel processing.

In their study on "Supervisory Event Loop-based Autoscaling of Node.js Deployments," et al. (2022)introduced an additional method for scaling Node.js applications: event loop-based auto-scaling. Because typical measurements like CPU use don't adequately capture system stresses, this study emphasizes the difficulty of doing so. As a metric for auto-scaling the program, This paper addresses several sorts of GEMs, such as atomic-LK, atomic-STM, read-only, owned, and portioned. Thanks to the internal functioning of the GEMs, the Web workers are able to access the shared memory in a controlled manner. Incorporating shared memory into Node.js apps will increase their possibilities and improve parallel processing performance. Prometheus is used to monitor the application and gather event loop lag measurements, whereas Kubernetes is used for container orchestration. The investigation yielded findings demonstrating that the event loop lag method may successfully scale the program with an increasing workload when performance was evaluated against traditional resource usage scaling. Nevertheless, as these analyses are conducted in Kubernetes, the outcomes may differ when conducted on other cloud platforms.

According to a 2015 study by A. Maatouki and Streit (2015) on "A Horizontally-Scalable Multiprocessing Platform Based on Node.js," Node Scala has become more well-known for its effectiveness at allocating and processing requests concurrently. Node Scala is utilized in this study to horizontally extend the application, optimizing resource utilization by splitting and requesting in parallel based on a predetermined use case. When handling huge data sets, this method showed a 74% boost in performance. As demand rises, Node Scala expands horizontally by adding more worker nodes, which improves the system's ability to process requests efficiently. This method was evaluated using a real-time application to visualize climatic data.

Applications that are effective at processing requests in parallel must be designed and developed using parallel design patterns. Organized parallel programming approaches are thoroughly examined in a study on "Algorithmic Skeletons and Parallel Design Patterns in Mainstream Parallel Programming" by Horacio González–Ve lez et al. (2020). This study emphasizes the effects of parallel programming approaches on various sectors. It investigates several parallel design patterns and algorithmic skeletons. Some of the frequent issues in parallel programming can be solved using the templates provided by the parallel design patterns. Task level parallelism, data level parallelism, and instruction level parallelism are all covered in this study.

In a research looking at ways to automate code execution and sharing between distributed systems, Oleh Chaplia et al. (2023)examine the Node.js framework for automated code scaling and execution on multiple distributed workstations. Enhancing microservices applications' scalability is the main goal of this research. The goal of this strategy is to make microservices-based applications more scalable and less complicated. For effective resource use, this framework runs shared code in a distributed environment.

Identifying the application's problems and enhancing its scalability require a performance evaluation. An innovative method known as Internal Transparent Tracing and Context Reconstruction (ITTCR) is presented in the study "Vnode: Low-Overhead Transparent Tracing of Node.js-Based Microservice Architectures" by Herve M. Kabamba et al.et al. (2023) (2024). This method allows for performance analysis of microservices applications and minimal system overhead tracing created in Node.js. This method concentrates on decreasing the amount of manual labor and resources needed to gather telemetry data in order to boost the debugging process' effectiveness. The scalability, availability, and resource efficiency of microservice designs are highlighted in this paper, along with the difficulties associated with debugging them.

As a result, when creating an application, it is crucial to adhere to green computing principles and take sustainability into account in every way. Maria Patrou et al.'sM. Patrou and Dawson (2022) investigation from 2022 on the article "Optimising the Energy Efficiency of Node.js Applications with CPU DVFS Awareness" offers a method for dynamically modifying CPU frequencies in response to request properties. This method, which made advantage of Node.js and user-specified frequency selection priority, significantly improved energy efficiency when compared to conventional Linux scaling governors. This method makes use of Frequency Scaling (DVFS) and CPU Dynamic Voltage on the application level, which enables the Node.js application to modify the CPU frequency in response to request demand, hence optimizing energy consumption. In order to help dynamically modify the CPU frequency, this new policy categorizes web requests according to their patterns of CPU utilization.Zhu et al. (2018)

Tilkov and Vinoski (2010) Tilkov et al.'s conducted a research by integrating the Rust programming language with JavaScript in Node.js for web applications development to enhance the performance. In this research C++ was replaced by Rust to avoid memory-related issues and avoid race conditions. Stress tests showed Rust outperforming JavaScript by up to 115 times and Node.js's concurrency model by 14.5 times without optimization. In browsers, Rust-based WebAssembly implementations surpassed pure JavaScript by 2-4 times in performance.

Table 1: Summary of Related Works

Reference	**Focus**	**Methodology**	**Key Findings**
Lei et al. (2014) K. Lei and Tan (2014)	Comparison of PHP, Python, and Node.js in handling high concurrent requests	Systematic benchmark and scenario tests	Node.js performs better for I/O-bound tasks but struggles with complex backend processing
et al. (2023)	Comparison of Node.js and Spring Boot	Performance testing with Apache JMeter	Node.js is less efficient for CPU-intensive, multithreaded applications
I. Chaniotis and Tselikas (2015)	Evaluation of Node.js for modern web applications	Performance evaluation study	Node.js is viable for real-time, low-latency applications

Table 2: Summary of Related Works

Reference	**Focus**	**Methodology**	**Key Findings**
M. Patrou and Dawson (2019)	Parallelism in Node.js under CPU-intensive loads	Evaluation of threads, cluster, and child processes	Multithreading is more efficient for CPU-heavy workloads
Bhandariet al. (2022)	Autoscaling Node.js deployments using event loop lag	Kubernetes orchestration and Prometheus monitoring	Event loop lag can be an effective autoscaling metric
PatrouM. Patrou and Dawson (2020)	Evaluation of Node.js parallelism under variable workloads	Docker-based experimentation with multithreading modules	WebWorker threads provide best performance for parallel tasks
Pratama and Raharja (2023) Pratama and Raharja (2023)	Benchmarking Node.js frameworks in virtualization environments	Node-Bench tool for latency and throughput measurement	Fastify performs best with Sequelize ORM in Docker and Podman
Tanadechopon and Kasemsontitum (2023) Tanadechopon and Kasemsontitum (2023)	Comparison of API services built with PHP, Python, Node.js, and Golang	Experimentation on response time, CPU usage, and throughput	Node.js is competitive but Golang excels in CPU-intensive scenarios
Tilkov and Vinoski (2010) Tilkov and Vinoski (2010)	Using Node.js for high-performance network applications	Functional programming with JavaScript	Node.js's asynchronous design enhances real-time performance
Maatouki et al. (2015) A. Maatouki and Streit (2015)	Horizontally scalable multiprocessing in Node.js	Use of Node Scala for parallel request handling	Achieved 74% performance boost with horizontal scaling
Sunarto et al. (2023) J.W.Sunarto and Widianto (2023)	Performance comparison of WebAssembly and JavaScript	Systematic review of benchmarks and implementations	WebAssembly significantly outperforms JavaScript in compute-intensive tasks

5 Methodology

The Research methodology involves several steps such as understanding the gaps in the previous research, designing an algorithm to efficiently distribute tasks and dynamically manage resources and allocation while performing each task, and developing a microservice app to integrate a load balancer library to perform CPU-intensive tasks. Calculating theoretical speedup using Amdahl's law. Deploying the NodeJs app in AWS and GCP cloud as containerised and non-containerised apps. Conduct experiments with different workloads and measure the performance. Monitor the CPU usage and memory usage through cloud monitoring tools. Collected the data of performance metrics. Analyse the data to identify the performance of each cloud through different experiments. Cost analysis based on the resources chosen in each cloud platform.

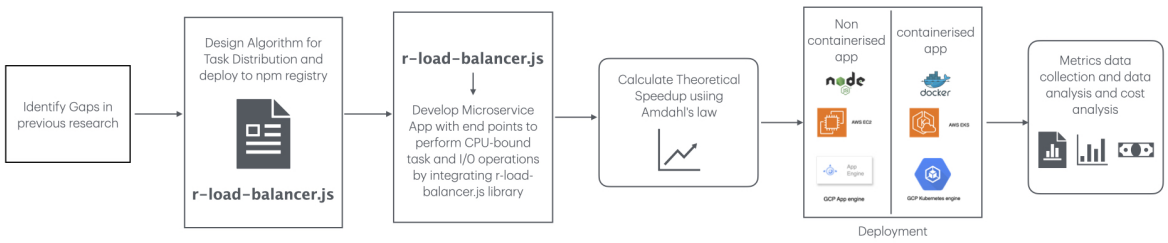

Figure 1: Methodology flow diagram

5.1 Identifying gaps from previous research

After examining previous works on improving the scalability of Node.js applications certain gaps were found and this research aims to fill the gaps by performing experiments on cloud platforms and building a custom load-balancer logic as a separate library which can dynamically allocate resources based on the load. In the earlier research libuv and napa.js are used to enable the parallel execution capabilities in a Node.js app however custom load balancer library solves the problem of dynamic resource management and in the previous research the experiments were not conducted with limited resources in this research the experiments are conducted considering the real-time scenarios of a microservice-based Node.js application in cloud platforms by deploying the applications in different services and analyze the performance.

5.2 Design Algorithm for Task Distribution (r-loadbalancer.js)

After analysing the libuv and Napa.js libraries the custom load-balancing algorithm is designed to address the issues like dynamic resource allocation like memory management and CPU allocation which enables the parallel processing and multi-threading in Node.js app to perform the operations by blocking new requests and also performing the existing CPU intensive task at the same time this algorithm is designed to boost the performance and also equip Node.js applications with parallel processing capabilities. The limitation of libuv is that it is not designed to perform parallel processing of CPU-intensive tasks the main thread in the event loop will be blocked while performing CPU-bound tasks using libuv it is ideal for I/O operations which can handle thousands of requests simultaneously. On the other hand, Napa.js has limitations of managing and sharing data between workers Napa.js provides a simple API to perform parallel processing in Node.js applications. The custom r-loadBalancer.js is designed to optimise the CPU-intensive task execution while it dynamically manages the resources. This custom library continuously monitors resources and efficiently allocates tasks to each core and it efficiently handles the load between CPU-intensive tasks and I/O operations without blocking any operations. An efficient custom task queuing mechanism is implemented to handle the workloads. This library has been published in the npm registry and can be installed in any node app using the npm install r-loadBalancer.js command.

5.3 Develop microservice Node.js App

To test the custom load balancer library a Node.js application was developed using a microservice architecture with multiple endpoints to perform the experiments with different scenarios by integrating the custom load balancer library. This app is designed and developed to perform CPU-intensive tasks such as processing large files performing complex calculations like RSA Fibonacci calculation and performing I/O operations like calling an external API. This app is designed in such a way that all the scenarios are run through API endpoints while monitoring the CPU usage continuously each CPU-bound tasks are divided equally to all the cores using workers and a custom load balancer library is integrated to perform the parallel processing and dynamic resource allocation. The CPU-bound tasks designed to analyse performance are running normal Fibonacci calculations and RSA Fibonacci calculations to find factors of a large number like 2000000 which is a good amount of load to do a performance analysis and single-threaded operations are also included to run in parallel to analyse the thread blockage. Along with these another operation is performed to generate 1000000 records using worker threads. A health API is designed to return the system metrics as a response which can used to monitor the CPU and memory usage. In one experiment all the endpoints are run at the same time to analyse if CPU-bound tasks are blocking other operations. This app is deployed in AWS and GCP as containerised and non-containerised apps to perform experiments with different configurations.

5.4 Calculate Theoretical Speedup

After designing and implementing the application and load balancing algorithm an analysis is performed to calculate the theoretical speed up by the amount of code that is parallelised and the number of cores that are used to run the application. Using Amdahl's law the theoretical speed-up is calculated by using the below formula. It is essential

to do this analysis which will give an idea of performance improvement that can achieved by integrating parallel processing. However, the value may change in real-time scenarios depending on the system resource usage.

$$1/(1-P)+(P/N)$$

where P represents the amount of parallelized portion of the code, and N is the number of processing units.

5.5 Cloud configuration and deployment

To perform all the experiments the developed application is deployed in multiple cloud platforms. The application is deployed as containerised and non-containerised applications in AWS and GCP using different services. In AWS the non-containerised application is deployed in EC2 with Amazon Linux which has 4 core CPUs with 8GB memory and AWS Elastic Kubernetes Service is used to deploy containerised application which is configured with 8 core CPU and 32GB memory. The GCP app engine is used to deploy a non-containerised application which has 4 core CPUs and 8GB memory and the GCP Kubernetes engine is used to deploy a containerised app with 8 core CPUs and 32GB memory. Monitoring tools are enabled to collect the data metrics of CPU usage and memory usage according to the experiments performed.

5.6 Conduct experiments with different workloads

To evaluate the implementation of this research experiments were conducted with different workloads to observe the performance of the application after the integration of parallel processing techniques. The workloads were designed to simulate the real-time use cases where applications can face a huge load of requests and CPU-intensive tasks either separately or at the same time. To perform the experiments the exposed endpoints were used like processing large datasets, executing CPU-bound computations like RSA Fibonacci calculations, and handling high volumes of API requests.

The application was deployed in containerised and non-containerised instances in AWS and GCP cloud platforms with the same configurations of respective instances in both cloud platforms. Experiments were conducted in each instance with the same amount of workload to analyse the speed up and performance. Each experiment is monitored with native cloud monitoring tools to collect the key metrics.

5.7 Performance metrics data collection and analysis

It is essential to closely monitor and collect the key performance metrics in order to evaluate performance improvement with multi-threading and parallel processing integration and analyse the performance difference in each cloud platform. This research will provide insights into the performance of both containerised and non-containerised applications. Key metrics like CPU usage, memory usage, response time are recorded to further analyse the overall performance collectively to draw conclusions. After the performance analysis cost analysis is also made in each cloud platform.

6 Design Specification

In this section we will discuss in detail about the techniques used, architecture, frameworks used and requirements to build and perform the experiments. The application is designed and developed to perform CPU intensive tasks and also handle I/O operations in parallel without blocking any operations. The backbone of this experiments is the custom load balancer library which is specifically designed to conduct this research that can optimise the resource usage and also dynamically allocate the resources based on the workload.

6.1 Techniques and Architecture

This research focuses on efficient resources utilisation and parallel processing by integrating crucial techniques. The Node.js applications usually runs on a single threaded event loop there are high chances that when a CPU intensive task is being performed all the other requests are queued and the main thread is blocked until the high computational task is completed. Worker threads and cluster modules are the inbuilt libraries of node.js which can be used to extend the capabilities of the node.js with parallel processing and multi-threading. Which will improve the performance of the application. It is essential to efficiently utilise the available resources in cloud.

The cluster module can fork multiple node instances which can run individually and communicate with the master thread back and forth via IPC. Each instance will have a node runtime and dedicated memory. Cluster will be used when process isolation is required. In this research the cluster module is used to fork child process when there is high workload incoming the forked child processes will handle all the I/O requests while the other processes handle the other computational intensive tasks

The worker threads on other hand is slightly different, worker threads are usually used to off load the computational intensive tasks from main thread and distribute the task to available cores preventing the blockage of main thread event loop. In this research worker threads are used to distribute the CPU intensive tasks efficiently to the available resources. Worker threads can share the same memory it can be used while the process isolation is not required.

6.2 Dynamic load balancing

After examining the existing libraries in node.js like libuv and Napa.js a custom dynamic load balancing algorithm is designed to efficiently distribute the tasks between the CPU cores and manage the memory. The custom library is named as r-loadBalancer.js and deployed in npm registry. This library plays a key role in this research experiments. This library is capable of distributing tasks across CPUs cleverly. This library is designed with dynamic load-balancing algorithm which can adapt to the workload by continuously monitoring the memory and CPU usage.

The adaptive task scheduling mechanism of the load balancing library determines which cores are available based on the threshold value of 10% of the time. If CPU core is idle for more than 10% of the time it will automatically allocate the task to that particular core. This adaptive mechanism can dynamic scale up and scale down the number of active workers based on the workload making sure the configured resources are utilized properly.

6.3 Memory and resource management

The algorithm includes mechanism to monitor the memory usage to ensure that tasks are allocated only when sufficient memory is available to prevent from system crashes and degradation in the performance. This will ensure that available memory is utilised efficiently. The getMemoryUsage function in the library keep track of the memory available and the dynamic load balancer will utilise this function to perform task allocation and execution with in the available resources. When the memory is low the system prioritise the I/O operations as they take milliseconds or seconds to finish the operation most of the time or reduce the task execution rate until the required amount of memory is available once the memory is available the task execution will speed up.

6.4 Cloud optimised design

The architecture of this system is designed to optimise the deployment for multiple cloud platforms. This architecture enables the horizontal auto scaling mechanism by adding more workers and processes and by adding an instance according the workload. As the resources are utilised efficiently the new instances are created only when the CPU usage is 80% and above which makes this design cost efficient.

The design includes containerisation and non-containerisation deployment strategies. The containerised app can be built once and deployed to any cloud platform using docker and Kubernetes service. This allows the engineer to customise the resources and manage them efficiently within the container. The non containerised application can be deployed to any cloud platform with predefined node environment set up available in the cloud while the resource configuration should be done before deploying the non-containerised application.

6.5 Frameworks and Libraries

Node.js: A javascript framework and primary component of this research as this research aims to improve the scalability of node.js application enhancing it's non-blocking, event driven architecture with parallel processing

Express.js: A web framework used to develop the micro-services and enable the REST API endpoints. Express.js is simple to use and implement the micro-service architecture in the node ecosystem and efficient in handling I/O requests.

r-load-balancer.js: This is the custom load balancer library designed specifically for this research and deployed in npm registry. This library can be utilised by other node applications to improve the performance and enable parallel processing capabilities.

big-integer Library: This library is used to handle large number computations, such as those required for RSA-based Fibonacci calculations. It provides reliable methods for performing arithmetic operations on numbers that exceed JavaScript's standard integer size.

Docker: Docker is used to containerise the application and build a docker image which can then be deployed to any cloud platform with Kubernetes service. Docker is efficient in building the images that can support multiple chip set architectures.

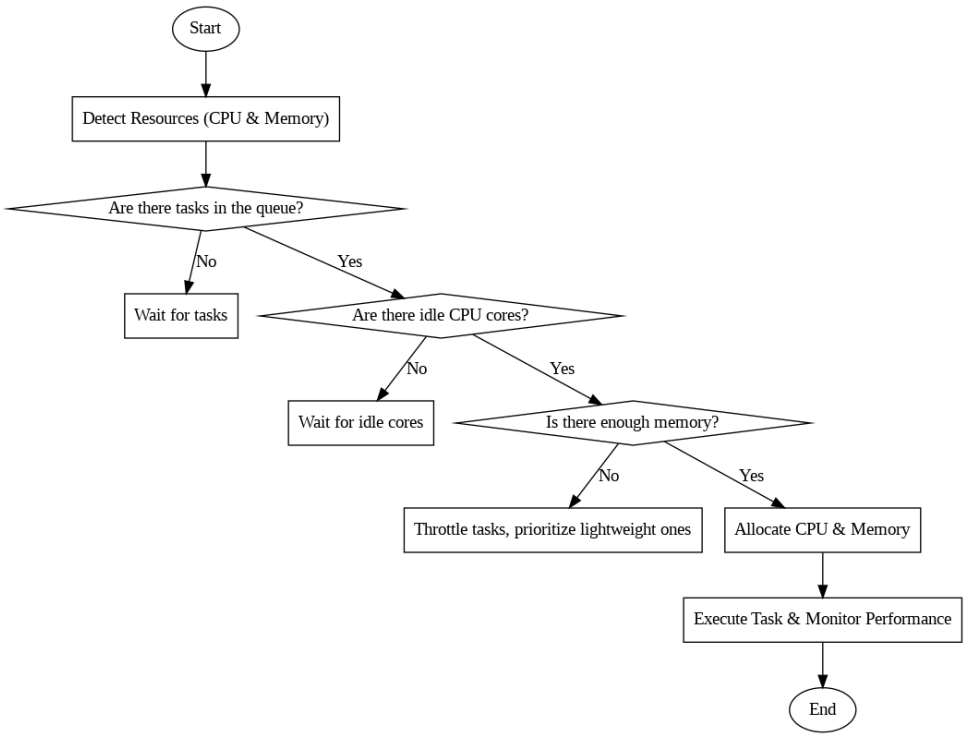

Figure 2: Methodology flow diagram

6.6 Algorithm Description

The foundation of this research is the algorithm developed for a custom load balancing library that can dynamically distribute workloads and manage resources efficiently. Let us discuss the key features of this custom load balancing algorithm in detail. The above figure 2 represents the flow and decision-making process designed in the custom load-balancing algorithm.

1. **Start:** When the application is initialised the process begins to detect the availability of the system resources, especially CPU and memory availability.

2. **Check task queue:** Once the resource checks are done system will start to analyse the task queue to check if there are any pending tasks in the queue. If there are no tasks present system will wait for the tasks to be added to the queue.

3. **Check for idle cores:** If there are tasks present in the queue system will start to identify idle cores the logic is designed in such a way that if any core is idle for more than 10% of the time the system will consider that core to allocate the pending task. If there are no idle cores system will hold the tasks in queue until idle cores are available

4. **Check for memory availability:** If idle cores are detected the system will analyse for memory availability. Here's where the dynamic decision-making design is integrated to throttle the CPU-bound tasks and prioritise the lightweight tasks until the resources are released from CPU-bound tasks.

5. **Allocate Resources:** If there are idle cores and enough memory available the system will start to allocate the required amount of memory and CPU to the tasks waiting in the queue.

6. **Execute and monitor:** Once the tasks are assigned the execution starts and the performance of the system is monitored continuously to ensure efficient resource utilisation.

7. **End:** The process then exits once the execution of the assigned task is completed. The system will continue to look for the next task and adjust the resources based on the workload.

This strategy helps to efficiently utilise the resources available and ensures that the system doesn't get overwhelmed with the workloads. The key features of the algorithm are resource monitoring, task scheduling, adaptive load balancing and memory management. This algorithm continuously monitors the system CPU and memory usage. Node.js built-in os module is used to collect the system health metrics. By continuously monitoring the resources they are efficiently utilised without leaving the cores idle for a very long time. All the incoming tasks are scheduled in a queue the algorithm identifies the available resources and assigns the tasks to the worker threads once the idle cores are detected. If there are no idle cores then the tasks continue to wait in the queue. The adaptive load-balancing mechanism dynamically adjusts the resource availability based on the workload. As the workload increases the number of workers will also increase. The algorithm efficiently handles the memory usage by dispatching the tasks from queue only when enough memory is available preventing the system crashes.

Algorithm for RSA-Based Fibonacci Calculation: To simulate CPU-intensive tasks an RSA-Based Fibonacci calculation is designed. The algorithm for this computation task is as follows. The big-integer library is used to compute larger Fibonacci numbers efficiently while the node.js int can not handle large numbers. Once the Fibonacci number is computed, it is used as a base in a modular exponentiation operation. This step is crucial for RSA-like computations where modular arithmetic is necessary. Once the Fibonacci number is calculated the same number is used as a base in a modular exponential operation which is a crucial step where the modular arithmetic is necessary. In this computational problem, the base is the Fibonacci number and the modulus is another large Fibonacci number.

6.7 Requirements

1. **Hardware Requirements:** The application is designed to run efficiently on machines with multiple CPU cores and a substantial amount of RAM. To conduct the experiments a 4-core CPU with 8GB RAM and an 8-core CPU with 32 GB RAM has been selected in AWS and GCP.

2. **Cloud Platform Requirements:**

    - The app can be deployed on AWS and GCP. It requires configurations for both containerised (using Docker) and non-containerised deployments.

- It utilises cloud monitoring tools, such as AWS CloudWatch and GCP Monitoring, to track performance metrics like CPU and memory usage.

3. **Software Dependencies:**

   - Node.js (version 18.x or higher)
   - Express.js (To build the micro-service application for the experiments)
   - big-integer Library (to handle large numbers during the experiments)
   - r-loadbalancer.js Library (for dynamic task distribution and resource management)

7 Implementation

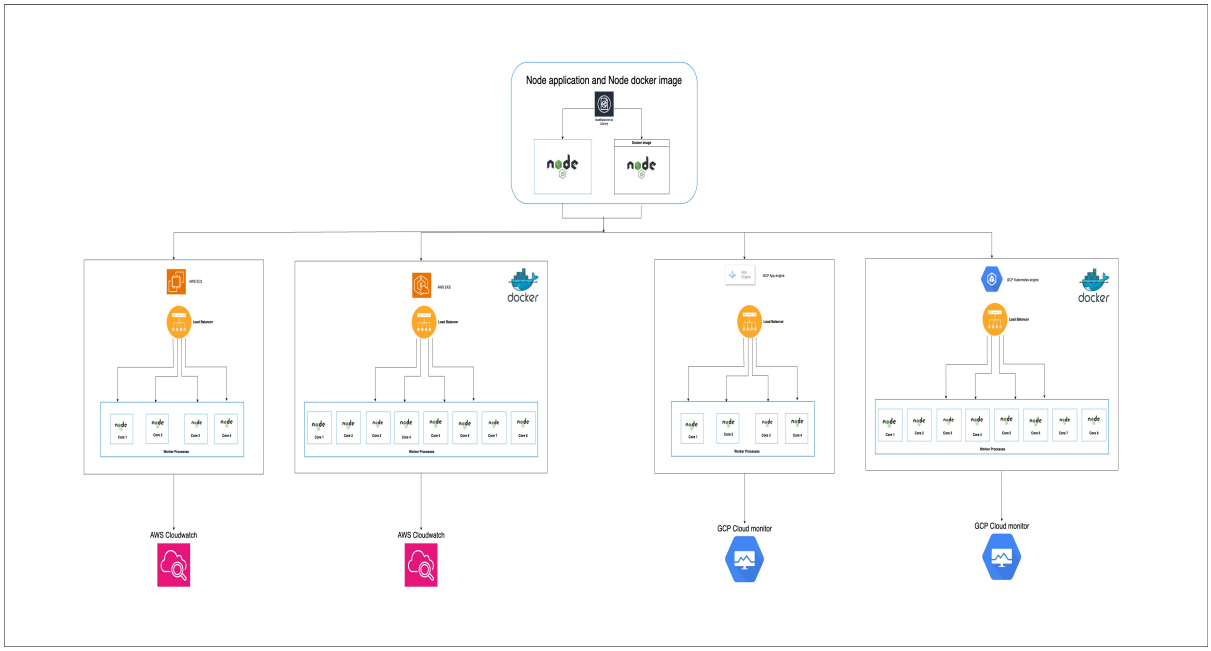

Figure 3: Architecture diagram

The implementation phase is crucial for translating theoretical analysis into practical execution in order to evaluate the hypothesis. The Implementation can be divided into multiple phases from development to deployment. The r-loadBalancer.js development phase a custom library with dynamic resource allocation and task distribution algorithm, a scalable Node.js application with microservice architecture development integrating the r-loadBalancer.js library. Deploying the application as containerised and non-containerised application in AWS and GCP cloud platforms. This implementation is aimed at evaluating the idea of optimising the node.js application performance by enabling the parallel processing and multi-threading techniques which will boost the performance of node.js applications. The above figure 3s illustrates the architecture of the experiment setup and evaluation. Let us discuss in detail about each phase.

7.1 Library and app development

The primary goal is to design and develop a library that enables parallel processing and multi-threading in node.js applications and develop a micro-service architecture-based node.js application to integrate the load balancer library and write REST API which will be used to perform all the experiments with different workloads.

The library is designed after carefully analysing the existing node.js libraries for parallel processing like libuv and Napa.js. The r-loadBalancer.js library aims to fill the gaps like dynamic resource allocations and dynamic task prioritization which is not available in both the libraries.

Core Functionality: The core functionality of the implementation lies in the custom load balancer library. This library is developed using cluster module and worker threads which are inbuilt modules of node.js that provides to enable parallel processing and multi-threading. The cluster module creates child processes (worker processes) that can run simultaneously along with other child processes sharing the same port this module leverages taking the advantage of a multi-core system and by forking child processes which can handle tasks individually. This helps the applications to handle more workload and process more requests at a time. The master process is responsible for forking the child processes using *cluster.fork()* method and this is a master and worker design.Node.js Foundation (2024a).

Each forked worker process will have its own node.js instance event loop which will process the tasks individually. The master process will distribute the tasks to the worker process and handle the incoming requests to the system. If any worker crashes or dies the master process can spawn a new worker and process the task maintaining the system stability.Node.js Foundation (2024b). Let us discuss in detail about the r-loadBalancer.js library.

r-loadBalancer.js: This library is built using cluster module and worker threads to manage CPU-intensive tasks and I/O operations efficiently. The main idea behind the design is to handle the workloads and I/O operations efficiently without blocking any process. In this section will discuss about each function written in this library and their functionality.

1. **getMemoryUsage():** This function retrieves the memory usage of the system especially the heap memory that is utilised by using the process an inbuilt module in the node.js *process.memoryUsage()* function is used to retrieve the memory usage information. This function is used to monitor the memory usage for dynamic resource allocation.

2. **runWorkerTask(workerTaskCode, workerData):** The runWorkerTask function creates a new worker thread to run CPU bound tasks. This function has 2 parameters the task code and relevant data. This function is also responsible for listening to the workers for completion which is a message from the worker. The function returns results on sucessful completion of the task and rejects when there is an error or worker fails to execute the task. The main purpose of this function is to offload the tasks to worker threads preventing main thread from getting blocked with CPU intensive tasks.

3. **detectIdleCores(threshold = 0.1):** The detectIdleCores function is responsible for detecting the cores that are idle if it's idle time exceeds the threshold time

which is 0.1 ideal for cloud platforms as the resources are configured to perform only certain tasks and it is important to utilise the resources efficiently. This function returns the count of idle cores which helps to allocate tasks based on the available cores.

4. **dynamicResourceAllocator():** The dynamicResourceAllocator function runs every 500ms and this time interval is configurable. This function performs the core logic of the library. It detects idle cores using *detectIdleCores()*. Continuously checks the task queue and check the memory availability by calling respective functions and makes a decision based on the available resources and the tasks waiting in the queue and also throttles the CPU intensive task when the memory usage is more and no enough memory is availble to perform the tasks. This function also logs the resource allocations.

5. **startLoadBalancer(app, port):** The startLoadBalancer functions initialises the cluster management and spawn worker processes by forking the number of worker threads based on available CPU cores and reserved cores for I/O tasks. In the next step dynamicResourceAllocator function is called for ongoing tasks management. This function make sure that if a worker dies a new one is spawned to maintain system availability. This is like a main function which start the application the app parameter is the node.js app where the r-loadBalancer.js library is integrated. And port is customisable and user can run app on any port.

6. **manageCpuBoundTasks(taskQueue):** The manageCpuBoundTasks function is responsible for handling CPU intensive tasks by analysing the available resources. Adds tasks to `taskQueue` if no cores or memory are available, for processing later. Resolves or rejects tasks based on whether resources were available at the time of execution.

7. **getHealthMetrics():** The getHealthMetrics returns the system metrics, including CPU load averages, uptime, total and available memory, and task queue details. All the metrics are returned as a JSON object with CPU core data, memory usage, task information fields. This function basically provides a snapshot of the system health.

All these functions collectively make r-loadBalancer.js library a powerful algorithm that can enhance node.js performance in cloud as cloud platforms provide extensive resources this library will ensure the resources are utilised efficiently by dynamic load balancing and task distribution. This can be integrated with any node.js application. In the below section we will discuss in detail on how to integrate this library in application.

Application development: As part of the research, a node.js application is designed and developed using micro-service design architecture. REST APIs are written to perform multiple operations like CPU-intensive operations and I/O operations. This application is key to evaluating the performance of the node.js application after integrating the r-loadBalancer.js library. This application is designed to perform CPU-intensive tasks like RSA-Fibonnaci calculation, normal Fibonacci calculations, I/O operations to call an external API and create 1e7 records which is also an CPU bound task and a Fibonacci calculation that runs only on a single thread. The key concept of this application design is to prove the hypothesis of enhancing the capabilities of node.js applications which can

perform CPU-intensive tasks efficiently while handling other tasks without blocking the main thread and also maintaining the event loop nature of node.js.

Express Server Setup:
The application is built using the Express framework, the app will start listening on port 3000 and handle HTTP requests.

Load Balancing and Resource Management:

- The application leverages the custom r-load-balancer.js library which is designed as part of this research and the core of this research's evaluation which dynamically manages CPU resources for efficient performance. This library can dynamically distribute task across multiple worker threads and offload the main thread.
- The *getIdleCores* function identifies the idle CPU cores based on the threshold of 10% if the CPU is idle for more than that threshold it will consider that as idle core, which are further used to allocate resources depending on the current workload.

Helper Functions:

- The **FibonacciBigInt** function is used in the application by */rsa-fibonacci* and */rsa-fibonacci-multithreaded* endpoints to perform complex calculations like RSA Fibonacci computation. This function used a big-integer library which is efficient in handling large values as the native node.js int has no capacity to hold such large values. This function calls *modExp* function which supports RSA calculations by modular exponentiation method that mostly used in performing cryptographic tasks.
- **File Chunking**: The **getFileChunks** function is used to split the large files into smaller chunks based on the available cores, which is further processed in parallel by worker threads.

API Endpoints: The application enables API endpoints to conduct the experiments using those API's

- **/rsa-fibonacci**: Computes Fibonacci sequences with modular exponentiation for encryption-like computations in a single-threaded manner.
- **/rsa-fibonacci-multithreaded:** Utilizes idle cores and distributed tasks to perform RSA-Fibonacci calculations in parallel.
- **Fibonacci Calculation Single & Multi-threaded**: This endpoint is used for performing calculation of Fibonacci numbers both in single-threaded */fibonacci-single* perform single threaded and */fibonacci* performs multi-threaded modes, useful for performance comparisons.
- **/fetch-photos**: This endpoint calls an external API using axios used to demonstrate the I/O operations while the CPU bound tasks are on going
- **/health:** This endpoint returns a JSON with all the health metrics of the system like memory available, idle cores, busy cores, memory used, uptime etc.

- **/process-large-file:** This endpoint is a POST method which is used to process large files this endpoint uses workers to process the files. It is tested using a file which is 1GB.

Each endpoint that performs multi-threading operations has a worker file where the task splitting and and assigning takes place based on the available cores. the worker file will have the business logic to perform the calculations. The app server starts with *start-LoadBalancer* function from *r-loadBalancer.js* library which manages the load balancing logic and resources as the requests come to the server.

7.2 Deployment

The application is desgined in such a way that it can be deployed in any cloud platform either as a containerised application or non-containerised application. As part of this research the application is deployed in AWS and GCP cloud platforms.

- **AWS EC2:** To deploy non-containerised application in AWS a EC2 instance has been used. The EC2 instance was created with Amazon linux and 4 core CPU with 8GB RAM the instance type is **t3.xlarge** the application is cloned using github and ran the application on EC2 instance by exposing the required port which can be accessable as a public endpoint.

- **AWS EKS:** The containerised applilcation is converted into a docker image that can run on any chipset. The converted image is further pushed to AWS ECR (Elastic Container Registry). This registry will have all the latest images everytime a new image is built an pushed. The steps and procedure to push the image is mentioned in detail in config manual. The AWS EKS(Elastic Kubernetes Service) cluster is created with a t3.2xlarge instance type which is a 8 core CPU with 32 GB RAM. everytime the deployment.yml file is run by the Kubernetes the latest image will be deployed to cluster and a new pod will be created with latest image. The VPC and firewall configurations are made to expose the port and enable the public ip access.

- **GCP App Engine:** The non-containerised application is deployed in GCP app engine to do this an app.yaml file is required with all the resource configurations and target port details. Apart from this simple change there are no other changes required to deploy the app in GCP App engine. Same amount of resources are selected in GCP as AWS EC2 this will help to analyse the performance of cloud platforms when this application is run and tested with different workloads. In GCP App engine 4 core CPU with 8GB RAM is configured

- **GCP Kubernetes Engine:** The application is already converted to docker image how ever the image should be pushed to GCP container registry by executing kubectl commands. Cluster has to be created with all the required system configurations and ports to be exposed and memory required. in GCP Kubernetes Engine a cluster with 8 core CPU and 32 GB RAM is created and the image is pushed and deployed to the engine through gcp ssh commands.

7.3 Outputs Produced

- **Codebase:** A micro-service-based node.js application was developed and the r-loadBalancer.js library which is the core of the implementation was developed and published to the npm registry.
- Performance Metrics: The output includes detailed key performance metrics that includes execution time, CPU and memory usage, and API response time. These metrics are collected by integrating the native cloud monitoring tools. These metrics are used to evaluate the performance of the implementation.
- Deployment: The application has been containerised and converted to a docker image using docker which can be deployed to any cloud platform using Kubernetes services. Both containerised and non-containerised applications are deployed in AWS and GCP and tested with different workloads.

7.4 Tools and Technologies Used

Programming Language:

The main objective of this research revolves around the enhancement of Node.js capabilities. The Node.js v18.x has been chosen to develop the micro-services and custom load-balancing library.

Frameworks and libraries:

- **Express.js:** Used to develop the REST APIs
- **Cluster module & Worker threads:** Used to develop the custom load balancing library (r-loadBalancer.js)

Containerisation:

- **Docker:** Docker is used to convert the application to a containerised image
- **Kubernetes:** Kubernetes services are used to deploy the containerised application

Cloud platform:

- **AWS:** Deployed non-containerised application in EC2 and containerised app in AWS EKS(Elastic Kubernetes Service)
- **GCP:** Deployed non-containerised application in GCP App engine and containerised app in GCP Kubernetes engine.

Performance Monitoring:

- **AWS Cloud watch & GCP cloud monitoring:** Used to monitor the performance of the application and collect key performance metrics and system health metrics

8 Evaluation

After conducting a series of experiment with different scenarios key metrics data are collected to analyse the performance of node.js application. The experimentation was conduction in the following order first calculating the theoretical speed up using Amdahl's law with different number of cores. Then executing the CPU bound task and a simple API call without parallelizing the application, Performing experiments on AWS EC2 with non-containerised app with 4 core CPU configuration, Experimenting with containerised app in AWS EKS with a 8 core CPU configuration and repeating the same in GCP App engine for non-containerised app and GCP Kubernetes engine for containerised app with same CPU configurations as AWS. The evaluation aims to analyse the data collect and first compare the performance of node.js after integrating parallel processing and multi-threading with a node app that runs on single thread then compare the performance between containerised and non-containerised app in AWS then compare the performance between containerised and non-containerised app in GCP and then compare the results between both the cloud platforms this evalution aim to draw the conclusions based on analysis. This experiments was conducted to perform high computational task which is RSA Fibonacci calculation by gradually increasing the N value. The other tasks like I/O operation of a simple API call to return health metrics, record gen(1M) is to generate 1 million records using worker threads, API call is trigerring an external API call using axios are also performed in parallel while the CPU intensive task was on going.

8.1 Experiment / Case Study 1: Calculating theoretical speed up using Amdahl's law:

There will be no speedup if the code is not parallelized even if there are more than one core if the code can not perform parallel processing then the speedup will always be 0. The formula to calculate theoretical speedup Amdahl (1967)

$$1/(1-P) + (P/N)$$

where P represents the amount of parallelized portion of the code, and N is the number of processing units.

After integrating parallel processing and multi-threading in the application 75% (can be written as 0.80) of the code is now parallelized. Let us calculate the speedup that can be achieved with 4 cores and 8 cores CPU respectively.

Theoretical speedup with 4 core CPU:

$$1/(1-0.80) + (0.80/4) = 1/0.4375 = 2.2857143$$

Theoretical speedup with 8 core CPU:

$$1/(1-0.75) + (0.75/8) = 1/0.34375 = 2.909091$$

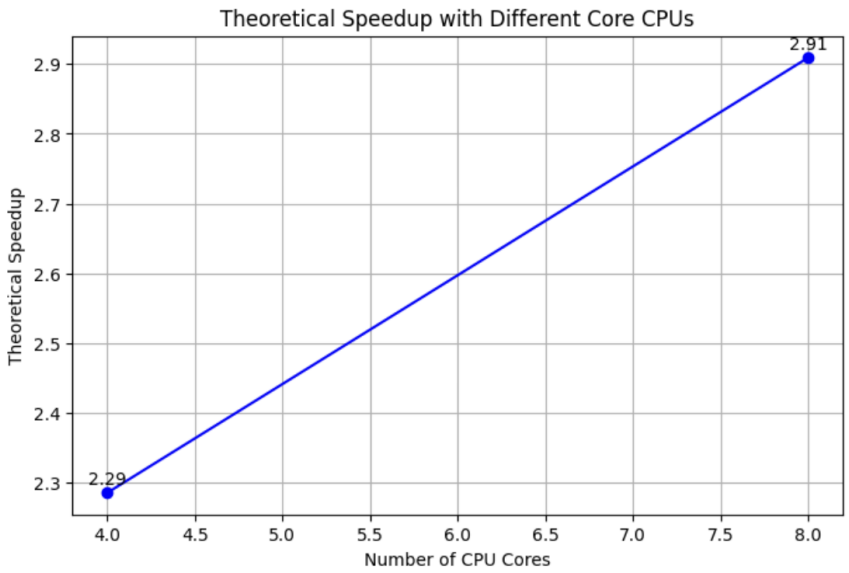

Figure 4: Theoretical speedup using Amdahl's law

As per the calculation the speedup is linear and there will be a notable speed up when using the 8 core CPU. However, in cloud platforms when we choose a 8 core or 4 core CPU we will not be able to utilize all the core as some CPU is reserved to perform system operations and this speed up estimation will vary.

8.2 Experiment / Case Study 2: Perform an experiment with only single thread execution:

This experiment is conducted in local environment with mac m1 chipset. In this experiment the code is not parallelised. The CPU intensive task which is Fibonacci calculation and a simple API call that returns system health data.
Observation 1: Time taken by /health API when called without initiating CPU intensive task was *6ms* which is below 10ms.
Observation 2: Time taken by fibonacci calculation API for n=1000000 is *12.60 seconds*
Observation 3: Time taken by /health API when called after triggering Fibonacci calculation API for n=1000000 is *10.2 seconds* while the CPU intensive task *12.60 seconds*
Based on this analysis it is clear that the CPU intensive task is blocking the main thread until the task is completed. when both the tasks are triggered at the same time the task which took only 6ms to complete took 10.2 seconds when triggered along with the CPU intensive task as the Fibonacci calculation took 12.60 seconds to complete and blocked main thread completely till the computation is completed.

8.3 Experiment / Case Study 3: Performing experiments on AWS EC2 with non-containerised app with 4 core CPU configuration:

This experiment is conducted with non-containerised app deployed in AWS EC2 with following configurations. Experiments were conducted to calculate Fibonacci for a large

number(3000000) along with other operations which runs in parallel

- **Instance type:** t3.xlarge
- **CPU cores:** 4
- **Memory:** 8Gib

Table 3: RSA AWS EC2 - 4 cores 8gb Ram — Amazon Linux (amzn2023.x86_64)

RSA - N	time (sec)	CPU	I/O (s)	Record Gen (1M)	API call
10	175ms	1 Core	111ms	2s 610ms	346ms
100	236ms	1 Core	88ms	2s 740ms	258ms
1000	263ms	1 Core	102ms	2s 580ms	271ms
10000	330ms	2 Cores	121ms	2s 670ms	371ms
100000	428ms	3 Cores	104ms	2s 720ms	289ms
1000000	10s 120ms	8 Cores	91ms	4s 910ms	260ms
2000000	1m 24s	8 Cores	56ms	5s 380ms	342ms

Table 3 is the data collection of time taken to finish task all those tasks were performed in parallel in the same system while the CPU intensive tasks were on going. It is evident that even when the large number computation line 2000 is been processing the other tasks were processed with minimum time. And as the workload increased the CPU allocation as increased.

Another experiment was conducted to calculate the normal Fibonacci of a given number along with other tasks running in parallel. In this experiment the Fibonacci calculation was done in 2 different ways. one with only single thread and other with multi-threading. Below are the results after the experiment.

Fibonacci (N)	Single-thread time (ms)	Multi-thread time (ms)	Single-thread I/O time	Multi-thread I/O time	Single-thread API Call Load (10M/4000 req/s)	Multi-thread API Call Load (10M/4000 req/s)
10	93ms	144ms	114ms	95ms	263ms	242ms
100	95ms	170ms	255ms	94ms	95ms	235ms
1000	89ms	144ms	295ms	100ms	95ms	267ms
10000	103ms	126ms	90ms	95ms	275ms	257ms
100000	269ms	228ms	100ms	64ms	282ms	197ms
1000000	7s 470ms	7s 230ms	109ms	144ms	255ms	436ms
2000000	1m 18s	1m 18s	98ms	86ms	278ms	436ms

Figure 5: Performance comparison single vs multi threaded in EC2

In the above table the metrics are recorded while performing similar operations with single and multi-threaded. The results displays a different behavior where the multi-threaded approach took longer time to perform Fibonacci calculations as the load increased the multi-threaded approach was faster than the single-threaded event loop. The above graph illustrates the response time difference between each task with a single and

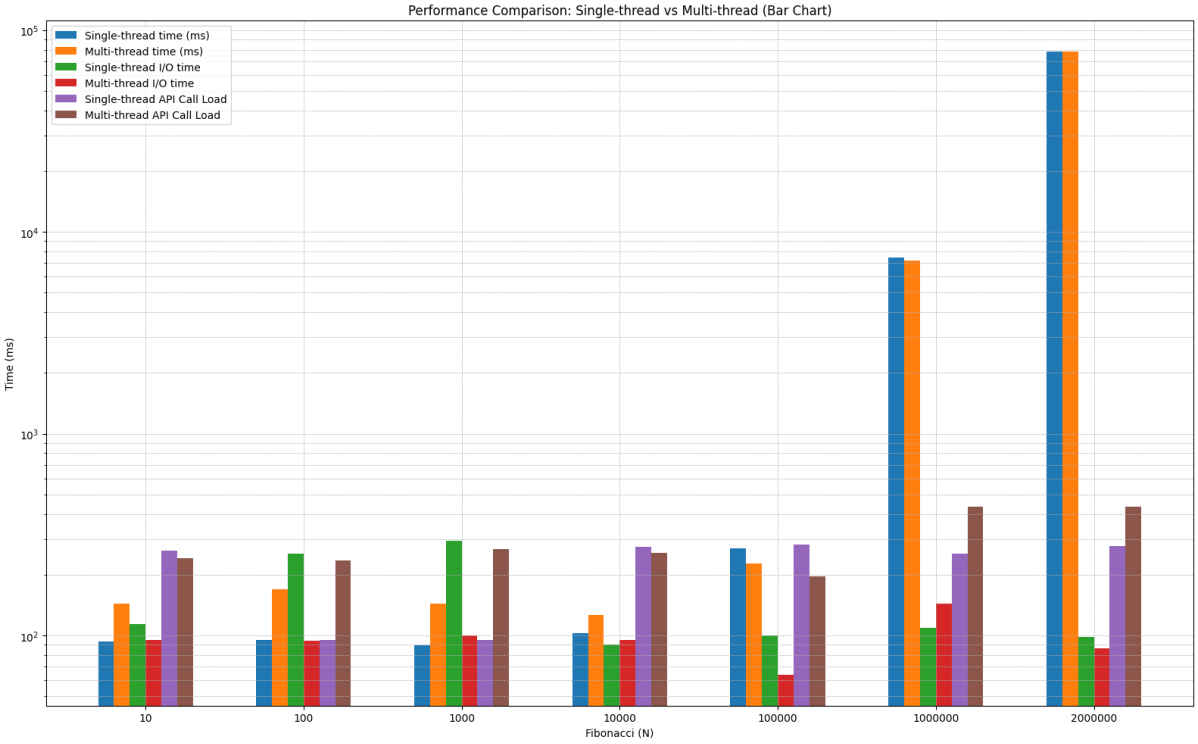

Figure 6: Performance comparison in EC2 bar graph

multi-threaded approach in AWS EC2 in non-containerised application.

The last experiment conducted in AWS EC2 is by running all the tasks at the same time with larger values in all the Fibonacci calculations. Please refer below table for the response time taken by each task.

This experiment proves that the custom load balancer logic efficiently handles tasks based on the workload and also dynamically handles resource allocations. The I/O API call took only 91ms even while the CPU-bound tasks were processing which took 5m 18s and 11m 48s respectively.

8.4 Experiment / Case Study 5: Performing experiments on AWS EKS with the containerised app with 8-core CPU configuration:

This experiment is conducted with the containerised app by building a docker image of the application and deployed to AWS EKS with the following configurations. Experiments were conducted to calculate Fibonacci for a large number(3000000) along with other operations which run in parallel. This experiment mainly focuses on analysing the performance of a containerised app.

- **Instance type:** t3.2xlarge
- **CPU cores:** 8 (only 6 cores were allowed to configure)
- **Memory:** 32Gib

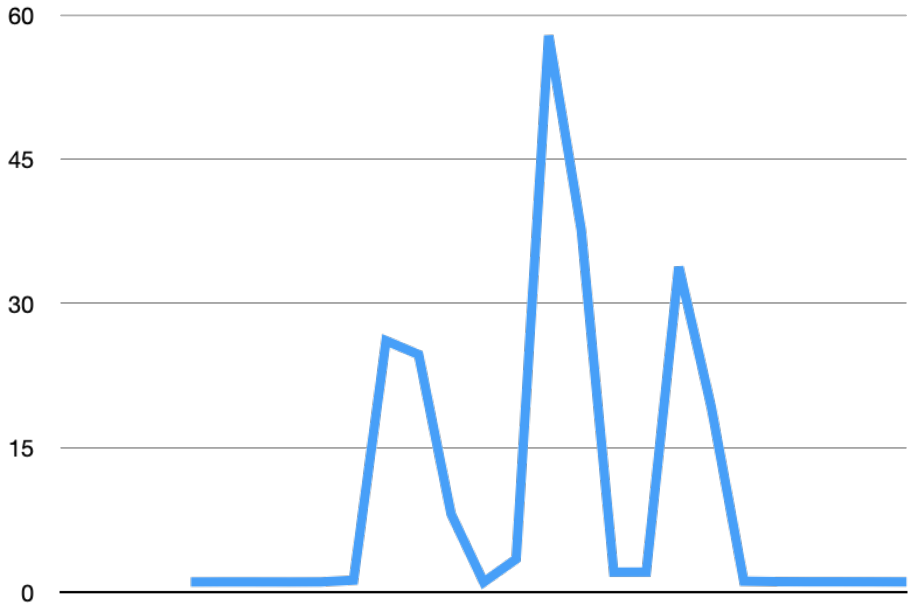

Figure 7: AWS-EC2 CPU utilization throughout the experiments

Table 5: AWS EKS - 8 cores 32GB Ram — Amazon Linux (amzn2023.x86_64)

(N)	time (sec)	I/O time	Record Gen (1M)	API Call (10M/4000 req/s)
10	462ms	91ms	3s 350ms	438ms
100	444ms	86ms	3s 290ms	445ms
1000	568ms	93ms	3s	770ms
10000	485ms	102ms	3s 140ms	419ms
100000	641ms	91ms	2s 880ms	310ms
1000000	10s 640ms	97ms	5s 110ms	333ms
2000000	2m 18s	103ms	7s 60ms	1s 340ms

In this experiment there are some noticeable difference in the responce time of 1million record generation and RSA Fibonacci calculation when compared to response time of non-containerised application. The last 2 executions has 2sec and 1min difference in RSA execution and 2sec in 1million record generation task.

Another experiment was conducted in containerised app to calculate the normal Fibonacci of a given number along with other tasks running in parallel. In this experiment the Fibonacci calculation was done in 2 different ways. one with only single thread and other with multi-threading. Below are the results after the experiment. In the above table the metrics are recorded while performing similar operations with single and multi-threaded. The results display a different behavior where the multi-threaded approach took longer time to perform Fibonacci calculations as the load increased the multi-threaded approach was faster than the single-threaded event loop. The above graph illustrates the response time difference between each task with a single and multi-threaded approach in AWS EKS in containerised applications.

Fibonacci (N)	Single-thread time (ms)	Multi-thread time (ms)	Single-thread I/O time	Multi-thread I/O time	Single-thread API Call Load (10M/4000 req/s)	Multi-thread API Call Load (10M/4000 req/s)
10	124ms	148ms	126ms	94ms	455ms	298ms
100	116ms	162ms	128ms	96ms	295ms	282ms
1000	106ms	135ms	113ms	90ms	288ms	285ms
10000	182ms	156ms	104ms	94ms	357ms	276ms
100000	286ms	212ms	104ms	92ms	283ms	290ms
1000000	8s 650ms	8s 290ms	105ms	95ms	302ms	277ms
2000000	1m 48s	1m 42s	107ms	107ms	290ms	612ms

Figure 8: Performance comparison single vs multi-threaded in EKS

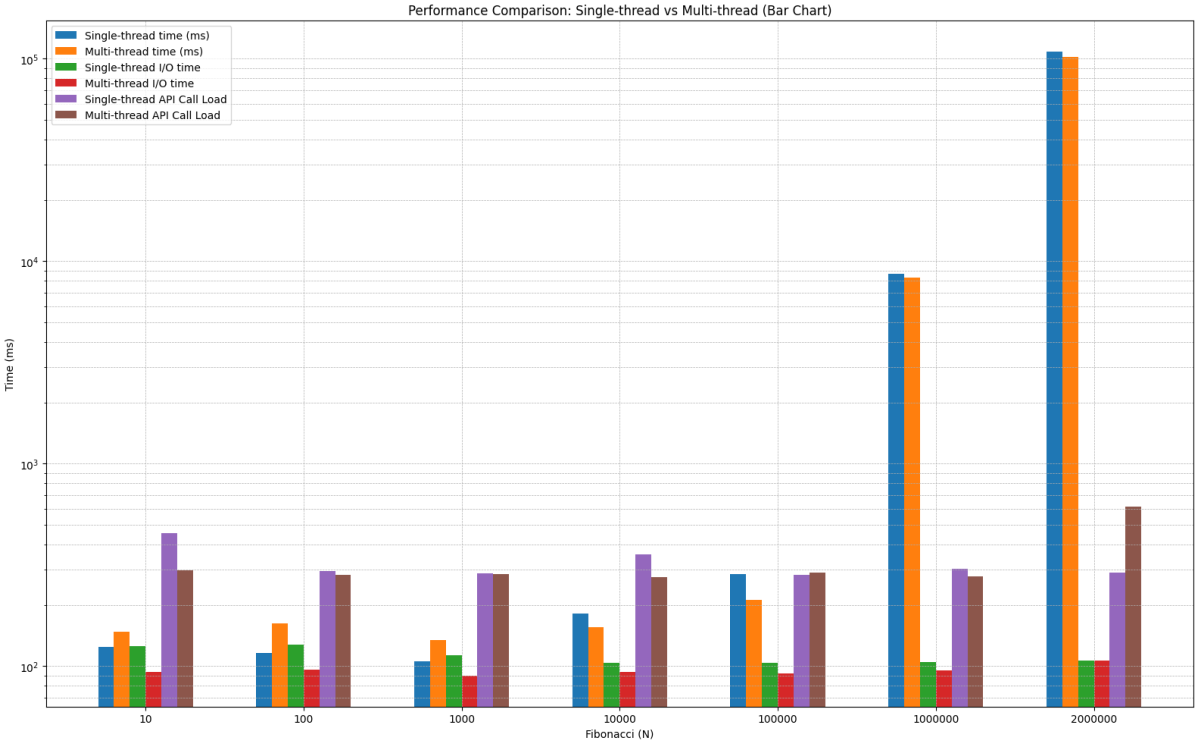

Figure 9: Performance comparison in EKS bar graph

The last experiment conducted in AWS EKS is by running all the tasks at the same time with larger values in all the Fibonacci calculations. Please refer below table for the response time taken by each task.

Table 6: AWS EKS - 8 cores 32 GB RAM — all tasks metrics

CPU	Fibanocci-(N)-2M	rsa(N)- 3M	I/O time	Rec gen (1M)	API Call
6	2m 12s	10m 6s	110ms	7s 320ms	347ms

This experiment proves that the custom load balancer logic efficiently handles tasks based on the workload and also dynamically handles resource allocations. The I/O API call took only 110ms which is more compared to the EC2 environment even while the CPU-bound tasks were processing which took 2m 18s and 10m 6s which is a huge difference compared with EC2 due to additional cores.

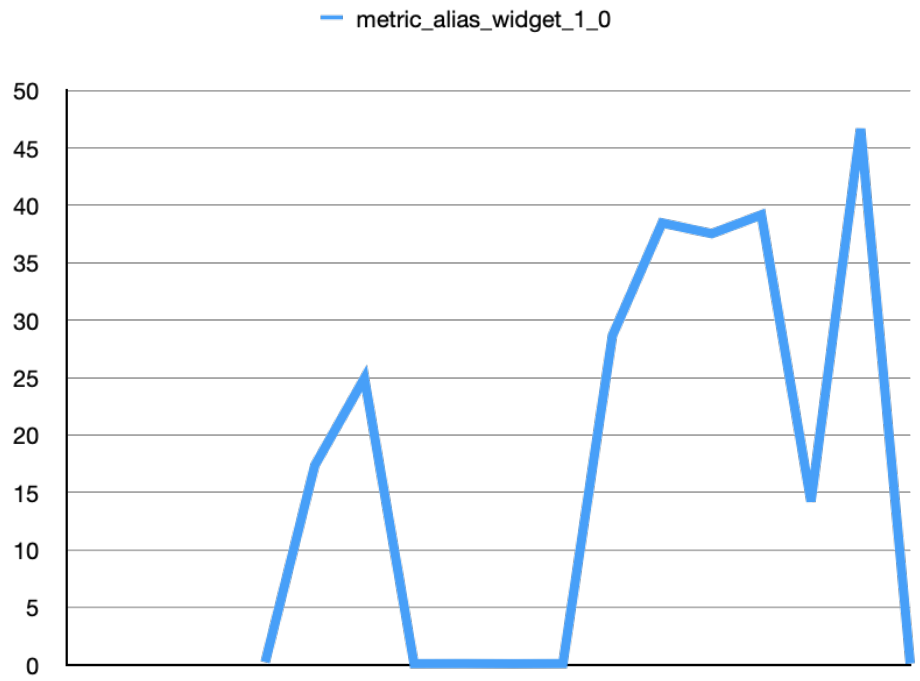

Figure 10: AWS EKS CPU utilisation through out the experiments

8.5 Experiment / Case Study 4: Performing experiments on the GCP App engine with the non-containerised app with 4-core CPU configuration:

This experiment is conducted with a non-containerised app deployed in the GCP App engine with the following configurations. Experiments were conducted to calculate Fibonacci for a large number(3000000) along with other operations which run in parallel

- **Instance type:** flex
- **CPU cores:** 4
- **Memory:** 8Gib

Table 7: GCP App engine - 4 cores 8 GB RAM

N	time (sec)	I/O (s)	Record Gen (1M)	API Call (10M/4000 req)
10	80ms	38ms	5s 510ms	270ms
100	250ms	23ms	5s 160ms	331ms
1000	90ms	32ms	3s 40ms	109ms
10000	100ms	27ms	3s 200ms	88ms
100000	230ms	31ms	4s 760ms	369ms
1000000	12s 280ms	28ms	6s 680ms	85ms
2000000	2m 17s 710ms	32ms	6s 230ms	275ms

Table **??** is the data collection of time taken to finish a task all those tasks were performed in parallel in the same system of a non-containerised app in the GCP app engine while the CPU-intensive tasks were ongoing. It is evident that even when a large

number of computations like 2000000 have been processed the other tasks were processed with minimum time. And as the workload increased the CPU allocation increased. Another experiment was conducted to calculate the normal Fibonacci of a given number along with other tasks running in parallel. In this experiment the Fibonacci calculation was done in 2 different ways. one with only single thread and other with multi-threading. Below are the results after the experiment. In the above table the metrics are recorded

Fibonacci (N)	Single-thread time (ms)	Multi-thread time (ms)	Single-thread I/O time	Multi-thread I/O time	Single-thread API Call Load (10M/4000 req/s)	Multi-thread API Call Load (10M/4000 req/s)
10	24ms	87ms	28ms	37ms	90ms	71ms
100	29ms	80ms	28ms	38ms	96ms	92ms
1000	33ms	83ms	30ms	38ms	117ms	79ms
10000	37ms	83ms	39ms	60ms	75ms	113ms
100000	223ms	253ms	50ms	87ms	100ms	136ms
1000000	12s 140ms	10s 660ms	38ms	34ms	88ms	87ms
2000000	2m 18s	2m 24s	75ms	30ms	144ms	90ms

Figure 11: Performance comparison single vs multi-threaded in GCP app engine

while performing similar operations with single and multi-threaded. The results displays a different behavior where the multi-threaded approach took longer time to perform Fibonacci calculations as the load increased the multi-threaded approach was faster than the single-threaded event loop. The above graph illustrates the response time difference

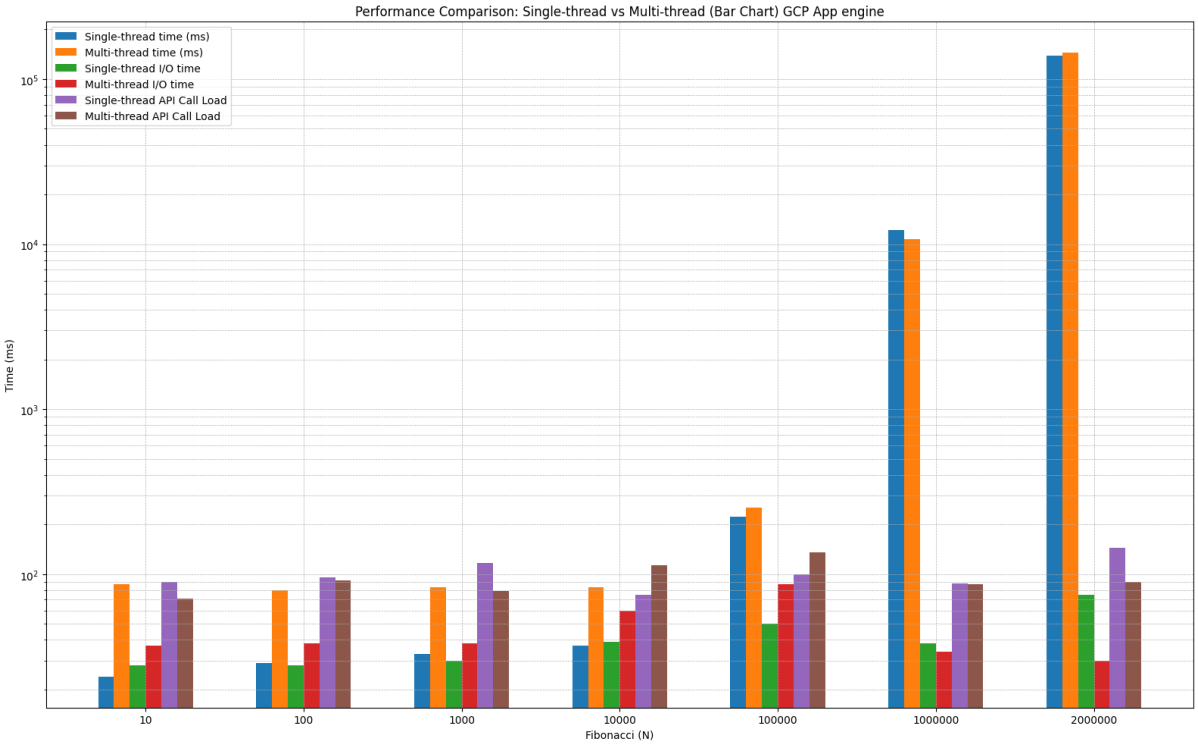

Figure 12: Performance Comparison: Single-thread vs Multi-thread (Bar Chart) GCP App engine

between each task with a single and multi-threaded approach in the GCP app engine in

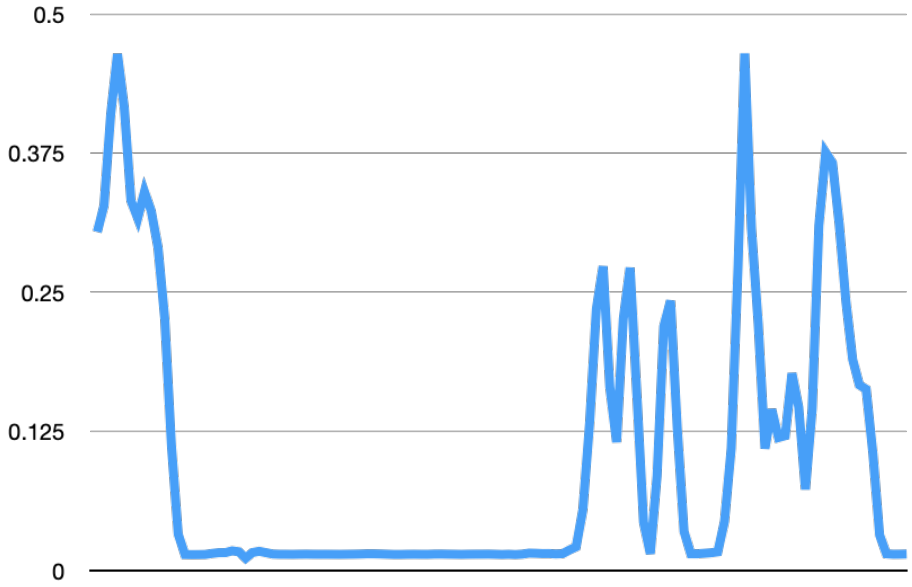

Figure 13: GCP app engine CPU usage through out the experiment

non-containerised applications.
The last experiment conducted in GCP app engine is by running all the tasks at the same time with larger values in all the Fibonacci calculations. Please refer below table for the response time taken by each task.

Table 8: GCP App engine - 4 cores 8 GB RAM — all tasks metrics

CPU	Fibanocci-(N)-2M	rsa(N)- 3M	I/O time	Rec gen (1M)	API Call
4	2m 36s	8m 44s 900ms	29ms	3s 880ms	92ms

8.6 Experiment / Case Study 6: Performing experiments on GCP Kubernetes engine with the containerised app with 8-core CPU configuration:

This experiment is conducted with the containerised app by building a docker image of the application and deployed to GCP Kubernetes engine with the following configurations. Experiments were conducted to calculate Fibonacci for a large number(3000000) along with other operations which run in parallel. This experiment mainly focuses on analysing the performance of a containerised app.

- **Instance type:** t3.2xlarge
- **CPU cores:** 8 (only 6 cores were allowed to configure)
- **Memory:** 32Gib

In this experiment, there are some noticeable differences in the response time of 1 million record generation and RSA Fibonacci calculation when compared to the response time of non-containerised applications. The last 2 executions 1000000 were a 3-second

GCP Kubernetes engine - 8 cores 32GB Ram

(N)	time (sec)	I/O time	Record Gen (1M)	API Call (10M/4000 req/s)
10	310ms	53ms	2s 240ms	115ms
100	190ms	63ms	2s 380ms	121ms
1000	210ms	43ms	2s 590ms	151ms
10000	240ms	53ms	2s 160ms	172ms
100000	420ms	75ms	2s 260ms	145ms
1000000	9s 250ms	28ms	4s 620ms	238ms
2000000	2m 24s 440ms	39ms	7s 240ms	233ms

faster than the app engine and 2000000 was 4 seconds slower than the app engine difference in RSA execution and 2sec in 1 million record generation task the last execution was slower than the app engine while generating 1M records.

Another experiment was conducted in a containerised app to calculate the normal Fibonacci of a given number along with other tasks running in parallel. In this experiment, the Fibonacci calculation was done in 2 different ways. one with only single thread and the other with multi-threading in GCP Kubernetes cluster. Below are the results of the experiment.

Fibonacci (N)	Single-thread time (ms)	Multi-thread time (ms)	Single-thread I/O time	Multi-thread I/O time	Single-thread API Call Load (10M/4000 req/s)	Multi-thread API Call Load (10M/4000 req/s)
10	53ms	89ms	28ms	48ms	152ms	114ms
100	61ms	91ms	25ms	51ms	144ms	108ms
1000	58ms	79ms	48ms	65ms	174ms	132ms
10000	61ms	94ms	93ms	33ms	170ms	122ms
100000	222ms	170ms	22ms	46ms	195ms	104ms
1000000	5s 810ms	5s 240ms	23ms	27ms	141ms	154ms
2000000	1m 54s 75ms	1m 7s	39ms	25ms	333ms	135ms

Figure 14: Performance comparison single vs multi-threaded in GCP Kubernetes engine

In the above table the metrics are recorded while performing similar operations with single and multi-threaded. The results display a different behaviour where the multi-threaded approach took a longer time to perform Fibonacci calculations as the load increased the multi-threaded approach was faster than the single-threaded event loop in GCP Kubernetes cluster.

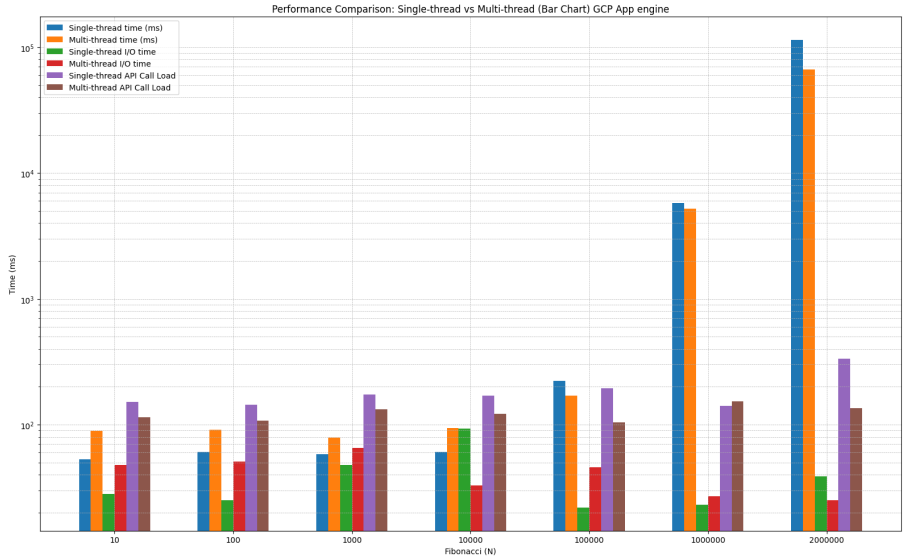

Figure 15: Performance Comparison: Single-thread vs Multi-thread (Bar Chart) GCP App engine

The last experiment conducted in GCP Kubernetes engine is by running all the tasks at the same time with larger values in all the Fibonacci calculations. Please refer below table for the response time taken by each task.

Table 9: GCP Kubernetes engine - 8 cores 32 GB RAM — all tasks metrics

CPU	**Fibanocci-(N)-2M**	**rsa(N)- 3M**	**I/O time**	**Rec gen (1M)**	**API Call**
7	2m	11m 52s 440ms	49ms	5s 580ms	142ms

This experiment, shows that the GCP Kubernetes engine performed better in normal Fibanocci calculation. However, when computing RSA-Fibonacci calculation the app engine demonstrated better performance which is 2 min less than the Kubernetes engine in real time this will make a huge difference.

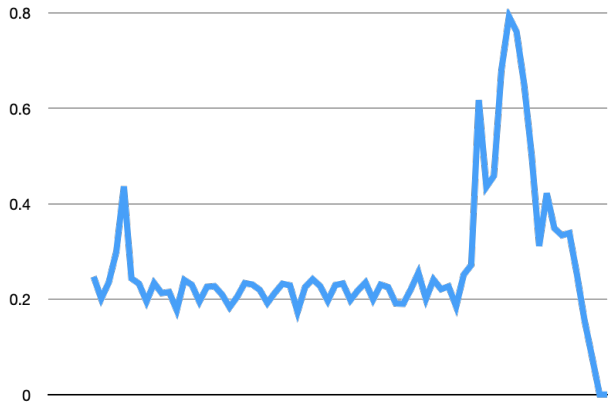

Figure 16: GCP Kubernetes engine CPU usage throughout the experiments

8.7 Discussion

Based on the analysis from the results of the experiments it is evident that the non-containerised applications demonstrated good performance compared to containerised applications. There was a significant difference in the performance while handling the CPU-intensive tasks, and I/O operations. The containerised application was a bit slower compared to non-containerised apps due to the added layers provided to ease the deployment and scaling configurations. In non-containerised environments, the user has less control over the configurations compared to containerised environments. The noticeable performance gap was identified when CPU-intensive tasks were performed where substantial amounts of CPU and memory is required. The non-containerised app performed better than the containerised environment. There's always a tradeoff to be made according to business requirements.

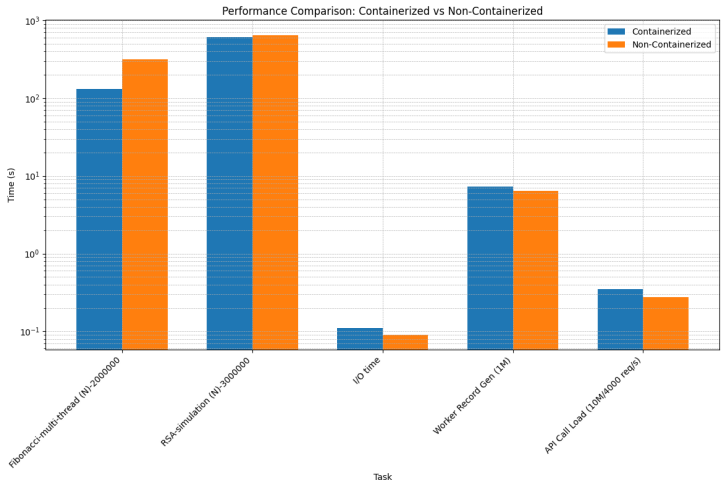

Figure 17: GCP Containerised vs non-containerised performance

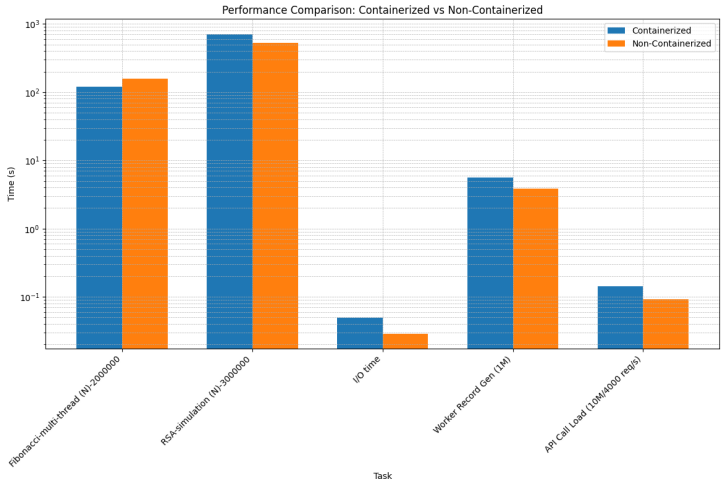

Figure 18: AWS Containerised vs non-containerised performance

9 Conclusion and Future Work

In conclusion, the non-containerised app performed well in both the cloud platforms. AWS showed slightly better performance compared to GCP due to its underlying infrastructure. While the containerised application in GCP showed a significant performance almost similar to AWS.

This research can further be carried out to enhance the custom load-balancing logic and also improve the code parallelisation to achieve more speedup. Further analysis can be performed to compare the performance in Azure cloud platform as well and cost analysis with energy efficiency analysis can also be performed to understand the further optimisations of the algorithm.

References

A. Maatouki, J. Meyer, M. S. and Streit, A. (2015). A horizontally-scalable multiprocessing platform based on node.js, *IEEE Trustcom/BigDataSE/ISPA*, Helsinki, Finland, pp. 100–107.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large-scale computing capabilities, *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ACM, pp. 483–485.
URL: *https://doi.org/10.1145/1465482.1465560*

et al., O. C. N. (2023). Comparison of node.js and spring boot in web development, *15th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, Bucharest, Romania, pp. 1–7.

et al., S. B. (2022). Supervisory event loop-based autoscaling of node.js deployments, *International Conference on High Performance Big Data and Intelligent Systems (HDIS)*, Tianjin, China, pp. 1–7.

I. Chaniotis, K.-I. K. and Tselikas, N. (2015). Is node.js a viable option for building modern web applications? a performance evaluation study, *Computing* **97**(10): 1023–1044.

J.W.Sunarto, A.Quincy, H.-T. and Widianto (2023). A systematic review of webassembly vs javascript performance comparison, *International Conference on Information Management and Technology (ICIMTech)*, Malang, Indonesia, pp. 241–246.

K. Lei, Y. M. and Tan, Z. (2014). Performance comparison and evaluation of web development technologies in php, python, and node.js, *IEEE 17th International Conference on Computational Science and Engineering*, Chengdu, China, pp. 661–668.

M. Patrou, J. M. Baird, K. B. K. and Dawson, M. (2020). Software evaluation methodology of node.js parallelism under variabilities in scalable systems, *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering (CASCON '20)*, IBM Corp., USA, pp. 103–112.

M. Patrou, K. B. K. and Dawson, M. (2019). Scaling parallelism under cpu - intensive loads in node.js, *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Pavia, Italy, pp. 205–210.

M. Patrou, K. B. Kent, J. S. and Dawson, M. (2022). Optimizing energy efficiency of node.js applications with cpu dvfs awareness, *IEEE 13th International Green and Sustainable Computing Conference (IGSC)*, Pittsburgh, PA, USA, pp. 1–8.

Node.js Foundation (2024a). Node.js documentation - cluster module. Accessed: 2024, `https://nodejs.org/api/cluster.html`.

Node.js Foundation (2024b). Node.js documentation - worker$_t hreads. Accessed$: 2024, .

Pratama, P. A. E. and Raharja, M. S. (2023). Node.js performance benchmarking and analysis at virtualbox, docker, and podman environment using node-bench method, *JOIV: International Journal on Informatics Visualization*, Vol. 7.

Tanadechopon, T. and Kasemsontitum, B. (2023). Performance evaluation of programming languages as api services for cloud environments: A comparative study of php, python, node.js and golang, *2023 7th International Conference on Information Technology (InCIT)*, pp. 293–297.

Tilkov, S. and Vinoski, S. (2010). Node.js: Using javascript to build high-performance network programs, *IEEE Internet Computing* **14**(6): 80–83.

Zhu, J., Patros, P., Kent, K. B. and Dawson, M. (2018). Node.js scalability investigation in the cloud, *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, IBM Corp., USA, p. 201–212.