Shradha Shrestha

CLOUD RESOURCE UTILIZATION AND ENERGY CONSUMPTION OPTIMIZATION

Cloud Based Software Engineering

2025

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Cloud Based Software Engineering

ABSTRACT

Author	Shradha Shrestha
Title	CLOUD RESOURCE UTILIZATION AND ENERGY CONSUMPTION OPTIMIZATION
Year	2025
Language	English
Pages	69
Name of Supervisor	Johan Dams

The project illustrates the design, deployment, and testing of a cloud monitoring and optimization framework for cloud resource optimization and performance to achieve enhanced energy efficiency as well as operating cost reduction. The system will be able to monitor the consumption of cloud resource usage currently in place and propose optimization recommendations against observed measurements. The overall objectives of the project were to provide timely accuracy in visibility of resource use, facilitate energy conscious decision making, and facilitate greener patterns of usage for the cloud.

The development cycle was adhered to in a disciplined manner, relying on Prometheus for metrics collection, Grafana for visualization, and Apache JMeter for load testing against varied loads. The system was deployed and tested on three environments making it flexible and robust. Extensive testing allowed the framework to be improved, resulting in a highly responsive and robust solution.

Overall, the system developed goes quite a long way towards energy-efficient cloud resource management, offering practical value to organizations that want to optimize resource use, minimize wastage of energy, and enhance sustainable cloud computing practices.

Keywords Cloud Computing, Resource Utilization, Energy Consumption, Resource Optimization

CONTENTS

FIGURES

1 INTRODUCTION

Cloud computing has become one of the major technologies that enabled digital transformation over the past few years across industries. Organizations have become increasingly dependent on cloud infrastructure to gain scalability, flexibility, and cost efficiency. The data centres available in the cloud environment gather enormous popularity for the provisioning of resources; due to the increasing demand of the data centre the energy consumption of data centres have been increased (A et al., 2024).

Since the data centers need much power to provide services, this increases CO2 emissions which raise the concerns about environmental impact and sustainability (Katal et al., 2022). Therefore, the proper utilization and optimization of the cloud resource is needed by ensuring minimum energy consumption.

The main purpose of this study is to find a way and strategies for cloud resource usage optimization while reducing energy consumption. The research aims at the development of an effective, light weight framework to gauge real time resource usage and suggest improvements towards better service management within cloud environments. With energy efficiency becoming an important factor both environmentally and economically now, the question is to operate cloud systems without compromising service performance. To support this objective, an operational framework was implemented that monitors the utilization of cloud resources and provides optimization recommendations. The framework monitors the amount of computing power, memory, and energy each service uses. When any service uses excessive resource or energy, the framework informs the user and recommends actions such as reducing or growing the size of a service based on the situation. This project helps demonstrate how cloud

services can be better managed, utilizing less energy and having less of an environmental footprint without losing performance.

1.1 Background and Motivation

Cloud computing has changed the way businesses and individuals access and make use of computational resources. Its scalability, flexibility, and efficiency in costs have made cloud computing the backbone of the modern IT infrastructure that's driving the global demand for cloud services. This fast growth has caused the boom of data centers that serve as the building blocks for cloud computing. Data centers, however, are one of the largest energy consumers and use an estimated amount of up to 12% of the electricity generated in US country in 2028 (Michel, 2025).

Basically, the energy demands of cloud data centers are contributed by several other factors that involve server operation, cooling systems, and redundancy mechanisms so as to ensure the continuous availability of a service. All these cause increased energy consumption due to inefficient resource allocation and underutilized hardware, hence leading to higher operational costs and environmental concerns (Pandey & Ahmad, 2019).

While virtualization and containerization improve resource efficiency, optimal energy management in dynamic clouds is very critical. It talks about the need felt dually adding energy effectiveness and optimizing cloud computing performance. Indeed concerned about the consumption of energy that utmost major companies have concern over encyclopedically, leading companies like Google, Amazon, ONLIVE, Giaki, and eBay (Hijji et al., 2022).

Most of the servers in the company are operating at only 10–15% capacity. Meanwhile, 30 of these commercial servers are considered 'zombie servers' they remain powered on and consume electricity, but perform no useful tasks (Katal et al., 2022).

This leads to bettered overall productivity, trustability, and vacuity of the system with considerable reduction in energy related costs of data centers from an profitable point of view. Besides the profitable benefits, there's an environmental concern. For case, it's estimated that by the time 2030, CO2 emissions will be as high as 720 million tons as cited by (Y. Liu et al., 2020).

Data centers now contribute much to global carbon emigrations, thus bringing the development of green computing. The increased operation in energy will contribute to a high rate of carbon and GHG emigration. This thesis is aimed at achieving to monitor the system or application and analyze the resource usage with recommendations which can bring energy effectiveness and high performance. Proposed results shall hence bring sustainable cloud structure, resource effectiveness, optimal energy operation .

1.2 Statement of Problem and Research Questions

The main problem or issue this study aims to solve is the efficient utilization of the cloud resource reducing the energy consumption. The research will discuss to determine the best optimization of cloud resources utilization and find out how to make a correct forecast of energy consumption according to cloud resource utilization metrics. The research formulates the following questions:

1. How does utilizing real time resource utilization metrics in cloud environments make for reasonably accurate estimates of energy consumption?
2. What are the best practices for optimization of resource allocation to minimal consumption of energy?
3. How does analytics resulting from resource usage help in making strategic decisions related to energy efficiency?

1.3 Objectives of the study

The main aim of this study is to propose and implement an integrated solution to maximizing cloud resource utilization and minimizing energy consumption in a cloud environment. The study addresses the challenges faced by organizations due to inefficient resource use, increased energy consumption, and high operational costs in cloud computing.

For this, the study introduces an end-to-end monitoring system constructed from a containerized Python application that collects and examines real time measurements for CPU, memory, and energy usage. This continuous monitoring approach helps detect inefficiencies such as over provisioning and under utilization, leading to energy wastage and unnecessary costs. Based on the data collected, the system provides actionable recommendations for scaling up or down services in order to enhance overall cloud resource efficiency.

Additionally, the research considers forward looking methodologies, such as the prospects of AI-based resource allocation and server less computing, to support sustainable cloud operations in the long term. Lastly, the thesis attempts to help shape energy-efficient, economical, and environmentally sustainable cloud computing solutions through empirical evidence on a live cloud infrastructure.

1.4 Structure of the Thesis

This thesis is composed in all of six chapters. First, the introductory chapter explains the background and the scope of research. The Chapter 2 of this thesis covers the literature review of the existing studies and research related to optimization in cloud resource utilization and energy consumption. Chapter 3 presents an overview of the related technologies and tools like Prometheus, Grafana, Python, and AWS. Chapter 4 introduces application architecture, key

technologies, and implementation details. Chapter 5 discusses the testing methodologies, testing environments and tools. Chapter 6 summarizes the paper, gives the outlook toward future work and points out areas of improvement.

2 LITERATURE REVIEW

This chapter considers the whole research and reviews the existing literature, identifies the gaps, and sets up the research. It also cites hypothetical studies to show how to integrate the findings of existing research. This road map will serve as a guide to understanding the concept of optimizing the cloud resources by breaking down the background and trends of cloud computing and the resource utilization. It also provides insight to the strategies or techniques to utilize the cloud resource.

2.1 Background of Cloud Computing and Resource Utilization

Cloud computing is one factor that has been acting as a game changer for IT infrastructure today, with its pooled computing resources scalable and on-demand, provided as servers, storage, applications, and services. Important developments in cloud computing have led to its evolution in the present form.

Emory University Professor Ramnath Chellapa in 1997 defined cloud computing as an armature in which economics determine rather than the constraint of technology. VMware developed virtual machine technology for x86 platform in 1999 that resulted in the armature of Cloud. Salesforce developed Software-as-a-Service(SaaS) that provides software through the internet (Slingerland, 2023).

Microsoft Azure and Amazon Web Services crafted in-house private cloud solution in 2010. Open-stack subsequently issued a do-it- yourself private cloud platform. In 2013, Docker began to provide lightweight, resource efficient software containers to drive building and deploying cloud applications simple and easy. Google released Kubernetes in 2014, which is the swish open source vessel orchestrating software that contains tone mending. AWS Lambda, introduced later during the same period, helped in developing server less operation by enabling more

effective operation of resources and scaling. AWS released its Machine Learning services in 2015 and, in 2016, its cloud based IoT services. This allowed real time processing of data and allowed micro services armature (Slingerland, 2023).

One of the most primary challenges in cloud computing is resource allocation efficiently. Resource allocation involves dynamic allocation of computer coffers to exertion and operations. Resource operation in a proper manner boosts performance, cuts costs, keeps service quality and icing energy efficiency. Dynamic workloads put advanced methodologies to multi cloud and edge computing scripts since traditional results are behind. Efficient allocation has an effect on operation performance, stoner experience, and functional charges (Zheng et al., 2024).

The first among resource operation-rested resource allocation approaches is to employ resources available in the present fashion with the least power operation to the minimum. In this way, one can insure that resources won't come spare. Optimal quantum of resource operation is needed for conservation of the terrain and it helps a person optimize profit (Abid et al., 2020).

A variety of green ICTs and energy reduction problems of ultramodern cloud computing systems also attracted huge attention from the exploration community. Other sweats have been made in developing energy consumption models, energy apprehensive cost development, workload change operation, and trying to gain an effective trade off between system performance and energy cost.

Junaid et al. (2020), proposed CSO and SVM methods for load balancing algorithm in cloud to reduce energy consumption. The file type will be classified in the cloud and SVM will classify the file one by one. The CSO method will be applied in distributing the files on the base of the load on VM to reduce energy and increases its performance. Simulation results

also indicate that the CSO system performs better in terms of energy effectiveness. The drawbacks of the Support Vector Machines approach is the reduced performance due to the handling of imbalance data.

Arshad et al. (2022), proposed an algorithm grounded on energy proficiency heuristics by exercising virtual machine connection to reduce lesser operation of energy consumption in the cloud data server environment. They build up a model for virtual machines relocation from one physical host to the other with an aim to lower energy consumption.

Liu et al. (2020), consider four objects for the virtual clusters to reduces the energy consumption and ameliorate the performance. The four objects are energy consumption, resource load balance, average resource utilization and availability.

Moura et al. (2021) used the internal value fuzzy logic approach to overcome the problems of resources using vagueness and inaccuracies to save energy with the lowest performance deprivation. They increased energy effectiveness by 2.3% in cloud computing simulation environments.

Buyya et al. (2023) promoted the transition from performance-oriented to energy-conscious approaches through the integration of intelligent, learning based resource management systems. Their work provides a thrust towards designing dynamic scheduling and provisioning algorithms trading off user QoS requirements with energy efficiency with the help of DVFS, thermal states, and workload partitioning across green-energy-fueled data centers. This energy aware approach seeks to minimize carbon footprints along with making the Cloud infrastructure of the future more eco friendly.

Although significant gains have been made through cloud resource monitoring and the optimization of energy consumption, there is still more that has not been fully mended. Most importantly, most of the present models make considerations for CPU utilization as a main metric

in estimating energy consumption while ignoring other vital factors such as memory usage and energy usage. Second, most research has tested these models in homogeneous environments, whereas in reality, cloud infrastructures are heterogeneous, hosting a mix of hardware with different energy profiles.

This research tries to bridge the gap of resource utilization energy consumption for holistic cloud resource management by applying real time resource monitoring, predictive analytics, and optimization algorithms.

2.2 Energy Consumption in Cloud Data Centers

Digital service demand is growing fast. Since 2010, the number of internet users has more than doubled and global internet traffic has expanded 20-fold. Energy use has risen with the growth in data centers and data transmission networks that underpin digitalisation.

The report from the International Energy Agency stated that according to its annual electricity report, 460TWh data center consumption in 2022 represented two percent of all electricity used globally. How much electricity consumption will increase up to 2026 depends on "the pace of deployment, range of efficiency improvements, as well as artificial intelligence and crypto currency trends", the report said but the authors predict the demand will grow to between 650TWh and 1,050TW (Global Data Center Electricity Use to Double by 2026 - IEA Report, 2024).

2.3 Resource Utilization and Allocation Strategies

Efficient resource utilization is one of the cornerstones that assure optimum performance with minimal cost and energy consumption in cloud computing. This section describes different Resource Utilization Strategies and Resource Allocation Strategies used in cloud computing environments. While resource utilization is about ensuring that resources are utilized to their fullest and distributed optimally, resource

allocation is concerned with the assignment and management of resources to meet workload demands.

Effective resource utilization ensures waste is minimized and that resources are utilized fully within the cloud infrastructure. The common strategies usually employed in cloud computing to improve resource utilization include consolidation, load balancing, and auto scaling of Virtual Machines. The three strategies will work together in the enhancement of system performance, cost efficient, energy consumption reduction, and resource allocation will be on the basis of current demand (Alsadie & Alsulami, 2024).

2.3.1 Virtual Machine (VM) Consolidation Techniques

VM consolidation is a critical strategy for resource optimization in a cloud environment. In this regard, the process of consolidating multiple virtual workloads on a few physical servers ensures the maximum utilization of resources while minimizing idle resources. Dynamic VM migration can help realize VM consolidation in large-scale hierarchical cloud data centers by ensuring that hosts distribute resources in a manner that provides a good balance of load with minimum energy consumption.

Zhang et al. (2024) introduce the VMM-HHGT VM consolidation approach in hierarchical cloud data centers. Energy consumption and resource contention are the major challenges in large-scale cloud environments where managing resources is quite complex. The proposed VMM-HHGT approach will merge Hyper-Heuristic-assisted broker techniques with Game-Theory-assisted hypervisor techniques to arrive at an optimal VM placement and migration strategy. With both energy consumption and resource contention handled effectively, it ensures higher utilization of resources by having higher deployment success rates, in general, under hierarchical cloud data centers.

2.3.2 Load balancing

The load balancer distributes the load to a large number of servers such that no one resource gets overwhelmed. It is an issue that is being resolved as the main approach that has been used for the purpose of maximizing utilizing of the resources in cloud computing such that no resource can be too bogged down with the traffic.

It consists of the distribution of a workload across numerous computing resources, such as servers, virtual machines, or containers, for greater efficiency, availability, and scalability. But this has to be planned and implemented correctly in a way that works out well without introducing additional risks. Cloud computing is able to balance its load at various levels, including the network layer, application layer, and database layer.

2.3.3 Auto scaling Mechanisms

Auto scaling provides auto scaling of resources based on system demand in terms of cost and performance. Auto scaling concept provides the users with an automatic method of adding and removing compute, memory, and networking scale of resources they have provisioned, in response to traffic spikes or specific usage patterns. Auto scaling is really a fresh cloud computing key topic area for deployments. Cloud computing provides users pay-for-what-they-use benefit from portion of the central elastic resources, such as applications available based on demand for fulfilling demands. Because auto scaling makes this cloud workload in achieving an application that expresses the desired availability and performance to keep varying services at optimum levels, workloads should utilize scaling outwards to their applications whenever needed (Kerner,2021).

Without auto scaling, the resources are statically defined and tied to one specific configuration for one specific set of resources. If, for example,

an organization has a massive analytics workload that it needs to process, it may need more compute and memory resources than were initially defined. With an auto scaling policy, compute and memory will scale automatically to process the data in a timely fashion. Some of the cloud providers supporting auto scaling features include: Amazon Web Services, Google Compute Engine, IBM Cloud, Microsoft Azure and Oracle Cloud Infrastructure (Kerner, 2021).

2.3.4 Task Scheduling

Task scheduling is choosing the best execution order for tasks to reduce latency and maximize throughput. The overall purpose it serves is concerned with resource utilization optimization, maximizing system throughput, reducing energy usage, reducing costs, and reducing processing time.

One of the principal issues is the task scheduling issue (TSP), and it makes users of the cloud computing platform experience latency in getting requests. TSP is an NP-hard problem responsible for efficient distribution of computing resources to application tasks (Hassan et al., 2022).

While first come first served (FCFS), round-robin (RR), and shortest job first (SJF), are classical scheduling algorithms, none of them solve NP-hard optimization problems; hence, in recent years, the latest optimization algorithms named meta heuristics algorithms have been used instead. These kinds of algorithms can find optimal or near-optimal solutions within reasonable time periods compared to the use of traditional scheduling algorithms. But still afflicted by falling into local minima and with a low convergence rate. To avoid this challenge, the scientists proposed a new task scheduler, known as hybrid differential evolution (HDE), as a solution to the task scheduling challenge in the cloud computing environment (Abdel-Basset et al., 2022).

Resource allocation is the task of assigning computation resources like CPU, memory and storage to workloads in a cloud data center. Good resource allocation ensures that workloads receive just enough resources to run at their best, without overwhelming the system or wasting resources.

2.3.5 Static vs. Dynamic Resource Allocation

Resource allocation strategies can be either static or dynamic. Under a static policy, resources are predecided statically according to the anticipated load, where normally huge under utilization is very often encountered for conditions other than the estimated workload. Nevertheless, it is the easiest do-it-yourself mechanism that suits routine circumstances.

The dynamic allocation approach adjusts resource distribution by real time workloads. Several techniques, like load balancing and task scheduling, help to ensure resources are optimally distributed in dynamic allocation ("Comparison of Static and Dynamic Resource Allocation Strategies for Matrix Multiplication," 2015).

2.3.6 Centralized vs. Decentralized Resource Allocation

Cloud resource provisioning can be widely categorized into decentralized and centralized types. In the centralized type, a single controller or scheduler allots resources to the entire data center, from complete system level information. Effective resource allocation and optimal decision making can be provided under this type. It is slowed down in the scalability aspect as a single point of control is the point of slowdown with a growth in the system size. Centralized systems are also more susceptible to failure compared to decentralized ones since any breakdown at the central controller will extend to the entire process of allocating resources.

Decentralized resource allocation, however, distributes decision making among different nodes so that each node can manage resources according to local conditions. This also makes it more fault tolerant because failure at one node need not be located throughout the entire system. Besides, decentralized approaches are more scalable than conventional centralized models because nodes operate independently without overloading a central controller. However, decentralized allocation may result in inefficiencies via lack of global overview and consequently lead to future imbalances in resources as well as subpar performance.

Both techniques compromise on scalability, fault tolerance, and system overhead, which must be given due attention based on the cloud infrastructure as well as operating requirements.

2.3.7 Resource Allocation with QoS (Quality of Service)

Quality of Service (QoS) provisioning of cloud resource allocation is paramount for latency-sensitive applications where predictability of performance is a primary requirement. QoS-sensing-based provisioning of resource allocation considers a number of parameters such as capacity to process, bandwidth, and response time such that workloads are at their predicted level of performance. Historical allocation techniques focus more towards utilization optimization, whereas QoS-based allocation techniques guarantee a higher priority on service guarantee for achieving reliability for mission-critical applications. For achieving efficient QoS-based resource provisioning, complex algorithms such as heuristic-based and priority aware scheduling algorithms are employed. These algorithms maintain system statistics as well and reschedule the resources in real time to prevent contentions over resources and bottlenecks (Khalid et al., 2021).

With QoS-aware scheduling algorithms, cloud providers dynamically increase/decrease resources based on workload demand to prevent performance degradation and ensure a high quality user experience. With cloud computing emerging as an ongoing paradigm, integrating QoS-aware resource management still remains critical in order to make the services both efficient, inexpensive, and dependable in multi tenant clouds.

2.3.8 Resource Allocation Based on Machine Learning

Machine learning (ML) has been employed for resource allocation and has gone a long way in making it more efficient as it permits predictive as well as adaptive strategies. Conventional allocation techniques rely on static rules or heuristic algorithms, which either lead to wastage of resources or inadequate provision. ML-based planning, however, applies historical data of workload in forecasting the swing in demand and redistributing the resources. Reinforcement and supervised learning algorithms facilitate ML models in making sense out of the past and adjusting in real time in response to demands for enhancing utilization of resources as well as lowest cost of operations. Another advantage of resource allocation through ML is that it can handle complex and dynamic cloud environments with high precision. The intelligent models can predict workload spikes, pre-book resources, and make cloud infrastructure resilient in dynamic environments.

Apart from that, ML algorithms also have auto scaling mechanisms that dynamically assign computing resources according to predicted usage patterns. The approach not only minimizes wastage of resources but also improves system responsiveness as well as availability and is a significant leap in the backdrop of contemporary cloud computing systems.

2.4 Barriers to Effective Cloud Resource Optimization and Their Impacts

The different approaches have been adopted in recent times to optimize cloud resources, and it mainly accounts for the reasons behind cloud resource optimization among most of the modern business. However, many organizations would also hesitate while adapting to the new practices on resource optimization and have one of the big challenges concerning security and compliance concerns.

Seth et al. (2024) confirmed that data security and compliance are some of the major issues coming up while enterprises deploy multi cloud systems. With multi cloud increasing these difficulties also where enterprises needed to deploy different security measures among all the cloud platforms and simultaneously ensure all such platforms must conform to standards or regulations that need to be strictly followed.

Multi cloud management is such a skill intensive task. Most organizations do not have the capability in terms of expertise and knowledge to adopt and implement resource optimization strategies. Finding someone who already has these skills can prove very difficult and as such can potentially increase training and operational costs (Baranenko, 2025).

Another challenge of effective cloud resource optimization is managing the costs involved in order not to over provision. Over provisioning will lead to unnecessary costs while under provisioning degrades performance and customer satisfaction. There are different cloud providers with multiple pricing models; these include per hour rates, per request charges, reserved instances discounts, and spot pricing. These differences make cost comparison hard, hence the budgeting complexity in multi cloud environments (Baranenko, 2025).

While performance, cost efficiency, and sustainability are pegged on

effective cloud resource optimization in the context of cloud data centers, these challenges should be addressed in trying to enhance cloud resource utilization and ensure that cloud infrastructures have efficient runs. Overcoming these requires further research in developing smarter algorithms, improved visibility tools, and cost management strategies.

3 THEORETICAL FRAMEWORK

This chapter will introduce some of the key concepts, theories, and models that form the foundation for understanding and optimizing cloud resource utilization and energy consumption. This section will provide a structured approach to monitor, analyze, and optimize cloud infrastructure performance while reducing energy usage.

3.1 Prometheus

The open source monitoring system and alerting toolkit Prometheus provides much important machinery in managing the modern cloud infrastructure. Prometheus focuses on gathering and storing such metrics in time series in a way that is successfully allowing monitoring and alerting cloud environments. Its ability for multi dimensional data collection and querying has made it exceptionally popular in dynamic, service oriented architectures. Designed for reliability, Prometheus is a standalone server that does not rely on network storage or other remote services, so it will keep running even during infrastructure outages. This is very important in diagnosing problems in cloud resource allocation and optimizing energy consumption.

Prometheus was developed at SoundCloud in 2012 to address limitations in existing monitoring tools like StatsD and Graphite. It introduced a multi-dimensional data model, operational simplicity, scalable data collection, and a powerful query language. Inspired by Google's Borgmon, Prometheus was open-source from the start and gained adoption by companies like Boxever and Docker (Wikipedia, 2025).

As of 2013, it had been used on production monitoring for SoundCloud and in 2015 had the first official public release. In 2016, it had been the second project to be accepted by Cloud Native Computing Foundation (CNCF) after Kubernetes and had gained traction with the major companies of DigitalOcean, Ericsson, and Google (Wikipedia, 2025).

Prometheus 1.0 was released in July 2016, followed by version 2.0 in November 2017. In August 2018, it graduated from the CNCF, solidifying its role as a key monitoring tool in cloud native environments. (Wikipedia, 2025.)

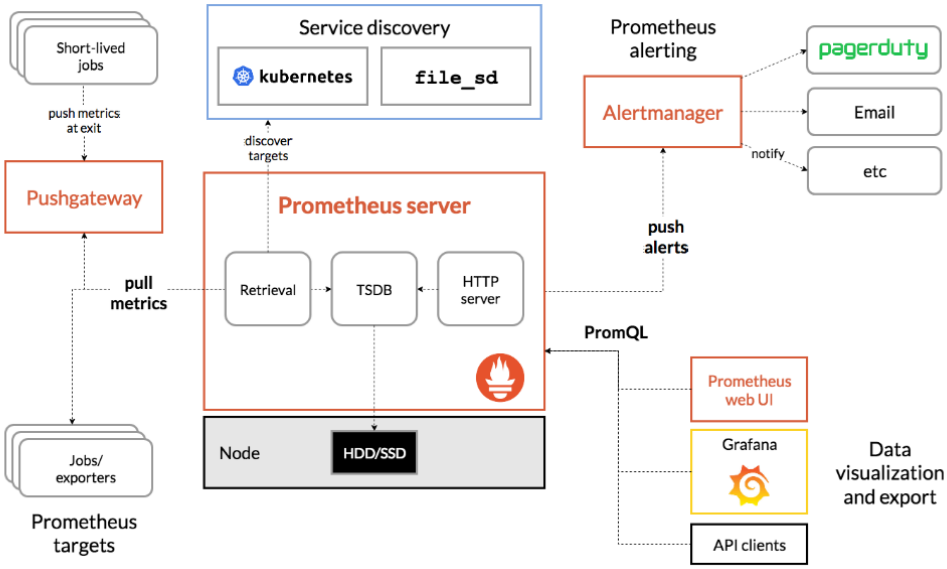

Figure 1. Architecture of Prometheus and some of its ecosystem components (Prometheus, n.d.)

3.1.1 PromQL

Prometheus provides a functional query language called PromQL (Prometheus Query Language) that lets the user select and aggregate time series data in real time (Prometheus, n.d.)

PromQL (Prometheus Query Language) allows you to query and analyze the metrics collected by Prometheus over time. We can use PromQL to retrieve resource usage data such as CPU, memory, and disk usage for your system. These queries can help you understand how resources are being utilized and detect any inefficiencies or spikes in usage. PromQL provides a lot of flexibility to filter, aggregate, and manipulate metrics, allowing you to perform in depth analysis of system resource usage.

Example of the PromQL query:

rate(process_cpu_seconds_total[1m])

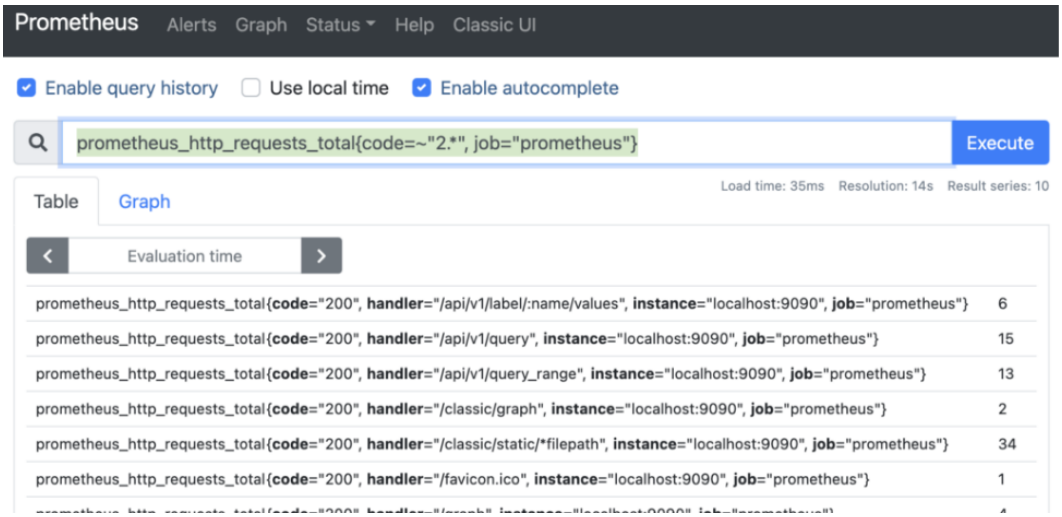

Figure 2. PromQL examples, showing labels for jobs and codes in regex, in the Prometheus UI (Reback, 2021)

3.1.2 Prometheus Alerts

The alerting system is critical to the health and efficiency of the cloud based monitoring configuration. The alert system should alert responsible stakeholders whenever pre-specified levels of resource usage, energy consumption, or system performance are violated. This ensures that any developing issues are identified and fixed early enough to avoid resource inefficiency or service downtime.

The alerting system is combined with Prometheus, which collects and stores metrics from the Python app and other services, and Alertmanager, which is a component of the Prometheus ecosystem that is designed specifically for alert management.

Prometheus is the core monitoring system of the system and collects system performance, resource utilization, and power consumption data. These are then stored and queryable, giving the foundation upon which the alerting system decides them on a continuous basis.

Prometheus alert rules are configured to look for specific thresholds, i.e., CPU, memory, or energy consumption. Once the threshold is reached, Prometheus triggers an alert. Such alerts are then routed, grouped, and notified by Alertmanager to the respective channels.

Example Prometheus Alert Rule:

The following is an example of a Prometheus alerting rule defined in the alert.rules.yml configuration file to alert on excessive CPU usage:

```yaml
groups:

- name: real_time_alerts

  rules:

  # High CPU Usage Alert for Any Docker Container

  - alert: HighCpuUsage

    expr: rate(container_cpu_usage_seconds_total{container_name=~".+"}[5m]) > 0.5

    for: 2m

    labels:

      severity: critical

      source: "docker"

    annotations:

      description: "High CPU usage detected for container {{ $labels.container_name }}."

summary: "CPU usage exceeded 50% for container {{ $labels.container_name }}."
```

Figure 3. Prometheus Alert Rules

3.1.4 Benefits of the Alert System

The alert system proposed in this project has numerous major advantages that help in improving the efficiency and reliability of the cloud system. The most important one of them is proactive monitoring, where the decline in performance, inefficiency in the utilization of

resources, or unusual activity will be detected and addressed immediately. This will prevent costly downtime or system failure and hence guarantee availability and stability in services. Besides, the system also supports customization, and alert thresholds can be configured according to some infrastructure needs. This capability makes sure that only meaningful issues are given priority without producing any false positives, which leads to real and effective alerts.

Also, the system is equipped with an alerting system based on seamless integration with Prometheus, Alertmanager, and Grafana for a unified monitoring environment. They are coordinated to obtain, process, and display alert information for efficient monitoring and system anomaly feedback by the admins. The other significant strength is in scalability since the system is quite scaleable to monitor additional services or infrastructure components as the project evolves. This ensures consistent monitoring and alerting even with the introduction of new workloads or cloud capacity. Through this warning system, the project enhances the persistent health and performance of the cloud infrastructure, hence contributing to efficient use of cloud resources and energy consumption.

3.1 Grafana

Grafana is an open source graphing and monitoring tool that talks to Prometheus to provide real time depiction of the usage of cloud resources and system activity. Prometheus has a knack for scraping and holding time series readings, whereas Grafana gets the information and displays it within interactive dashboards in which users can monitor trends, examine performance, and readily spot anomalies.

Grafana, first released in 2014 by Torkel Ödegaard, was conceived as a project at Orbitz and was designed to work first with time-series databases like InfluxDB, OpenTSDB, and Prometheus. Later, relational

databases like MySQL, PostgreSQL, and Microsoft SQL Server were added. Grafana Labs, the company that created Grafana, received considerable funding, including $24 million in Series A (2019), $50 million in Series B (2020), and $220 million in Series C (2021). Grafana Labs also bought several companies, including Kausal in 2018, k6 and Amixr in 2021, and Asserts.ai in 2023 (Wikipedia contributors, 2025a).

Grafana is able to generate charts, graphs, and web notifications upon integration with supported data sources. It is capable of CPU, memory, and network usage monitoring by cloud providers, Docker container metrics, and monitoring AWS resource utilization. Grafana, as an addition to providing a simplified way of comprehending data, is a central component of the monitoring setup for optimizing observability and enabling cloud resource optimization.

3.2 Docker

Docker is software used to automate application deployment into light weight containers such that applications run seamlessly in various environments isolated from one another (Wikipedia contributors, 2025). The function of Docker is to help achieve efficient deployment, monitoring and scaling of the system installed with the aim of maximizing the utilization of cloud resources as well as minimizing power consumption.

3.2.1 Docker Architecture

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of running, creating, and deploying your Docker containers. The Docker client and daemon can share a host or you can remote mount a Docker client to a remote Docker daemon. The Docker client and daemon use a REST API to communicate with each other, either via a UNIX socket or network interface. Another client for Docker is Docker Compose, through which

you can interact with applications that consist of a set of containers (Docker Documentation,n.d.).

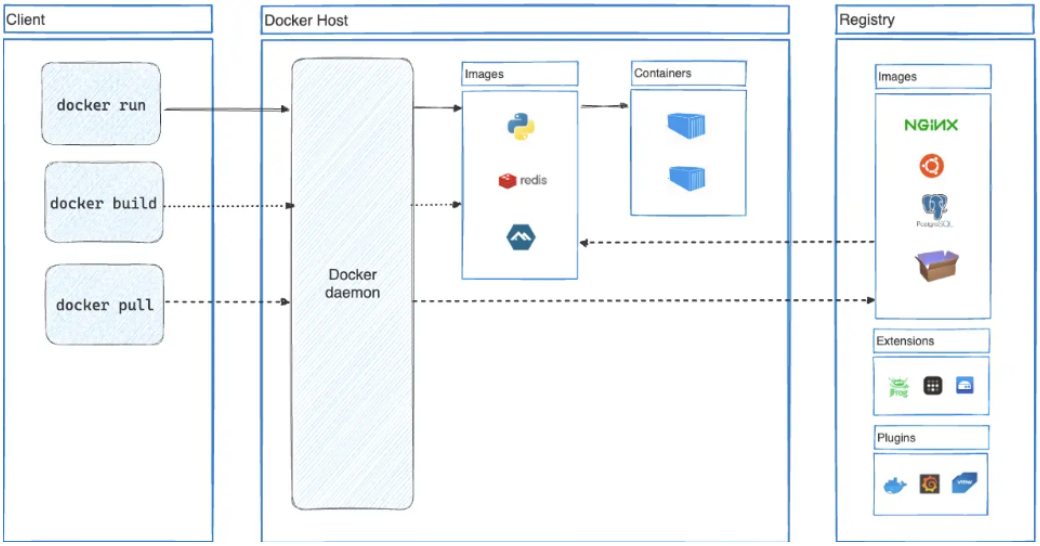

Figure 4. Docker Architecture (Docker, n.d.)

3.2.2 Docker Components

The three major building blocks of the Docker Software as a Service offering are Software, Objects, and Registries. All of them have an important role to play in containerizing applications and maximizing cloud resource utilization.

Software Components

The fundamental software building blocks of Docker are the Docker Daemon and the Docker Client. Both are responsible for the orchestration and running of containerized applications.

The Docker daemon (dockerd) is a long-running background process that manages Docker containers and performs operations related to the container. It accepts incoming requests through the Docker Engine API

and is responsible for operations like building, running, and monitoring of containers. The daemon is the primary process in Docker container management system that oversees the correct lifecycle of the container (Wikipedia,2025).

The Docker Client (docker) provides a command-line interface (CLI) by which users can interact with the Docker daemon. Users provide commands via the client to manage containers, such as starting, stopping, or removing containers. The client sends requests to the Docker daemon to execute operations, offering an abstraction between the user and the process of container orchestration (Wikipedia,2025).

Objects in Docker

Docker objects are the objects that store and specify the containerized applications. They are the center of Docker activities and consist of containers, images, and services.

A Docker container is an executable image of Docker, which is an immutable definition that is applied to create containers. Containers package applications and their dependencies into a secluded environment that may be executed predictably across different systems. They could be started, released, stopped, relocated, or deleted through Docker CLI or API. Furthermore, containers can be joined to networks, mounted on volumes, and even dedicated in hopes of building a new image out of their current state (Docker Documentation, n.d.).

Containers also have the ability to share the kernel of the host operating system, thus dispensing with a large amount of the overhead in hosting multiple app instances. In other words, system resources are not used so extravagantly as is the case with virtual machines, since containers make less usage of resources than virtual machines.

In resource optimization in the cloud, this translates to that each container is only allocated the resources it needs, without over-provisioning and inefficient energy usage. Docker provides a variety of resource utilization benefits, which are within the optimization range of this project. Containers, unlike traditional virtual machines, share the host operating system's kernel and hence decrease per instance execution overhead of the applications to a great extent. This facilitates improved utilization of system resources because containers use less than virtual machines.

Docker images are read only templates that are utilized to build containers. They hold the file system and configuration needed to execute an application. Images can be built from scratch or based on other images, making it easy to reuse common components of an application. Docker images are versioned and immutable, offering a predictable and reproducible way of deploying applications across different environments (Wikipedia,2025).

Docker services offer the flexibility of scaling and orchestration of containers across multiple Docker daemons. In a multi node environment, a service can be used to identify a collection of containers that is treated as a single entity. Scaling is offered by Docker Swarm, which is an orchestration and clustering layer that assists in distributing containers across different machines. Services enable containerized applications to scale up or down with ease depending on the workload demands (Wikipedia,2025).

Docker registries

A Docker registry refers to a central location for storing and distributing Docker images. Registries enable sharing of container images between developers, allowing seamless deployment and management of applications. Docker supports public and private registries, which cater to different use cases.

The global default public registry, Docker Hub, is a global repository where users can pull an enormous number of existing images. Docker Hub allows for the sharing of application images to the Docker community, such that users can pull, push, and share containerized application (Wikipedia,2025).

Docker also supports the creation of private registries, which allow companies to store their proprietary images in a secure manner. Private registries provide image distribution control to help assure that internal or sensitive applications are only seen by authorized users. (Wikipedia,2025).

Docker registries can also be used to give event based alerts, which allow automated processes to be triggered whenever images are pushed or modified. The feature proves useful in continuous integration and continuous deployment (CI/CD) pipelines, where rapid and automated updates of container applications are paramount (Wikipedia,2025).

Dockerfile (example)

```
ARG CODE_VERSION=latest
FROM ubuntu:${CODE_VERSION}
COPY ./examplefile.txt /examplefile.txt
ENV MY_ENV_VARIABLE="example_value"
RUN apt-get update

# Mount a directory from the Docker volume
# Note: This is usually specified in the 'docker run' command.
VOLUME ["/myvolume"]

# Expose a port (22 for SSH)
EXPOSE 22
```

Figure 5. An example of a Dockerfile (Wikipedia,2025)

3.2.3 Benefits of using Docker

As a containerization platform, Docker enables development of portable, lightweight, and isolated environments, therefore, the ideal fit to execute the various components of my system like the Python application to be used for monitoring, Prometheus for the collection of metrics, and Grafana for visualization.

Docker provides several advantages in the realm of resource usage, which is one of the optimization goals of this project. Containers are distinct from the conventional virtual machines since they share the host operating system's kernel, which reduces a great deal of overhead that comes with running multiple copies of applications. This results in improved system resource utilization since containers consume less resource compared to virtual machines. In addition, Docker offers resource isolation such that each component of the system, whether the Python application, Prometheus, or Grafana, is run in its own isolated environment.

Docker also boasts numerous advantages regarding resource utilization, which are at the center of objectives for the optimization of this project. Effective deallocation and allocation of resources guarantee cloud infrastructure is utilized only when needed, with no resource wastage during low demand scenarios. Docker's lighthearted nature makes it possible to scale quickly with no overhead that is otherwise inherent in virtual machine based implementations, directly contributing towards minimizing the use of energy in cloud setups.

3.3 Nginx

NGINX, otherwise referred to as "engine-x," is a free web server software used for reverse proxy, load balancing, and caching. NGINX went on to become one of the highly used products in the internet realm

of web hosting and application delivery. It was originally created by Igor Sysoev in 2004. It went on to become an absolute necessity in the technology stack of many sites and online businesses because of its striking performance, extreme scalability, and vast feature list (Alesh, 2023).

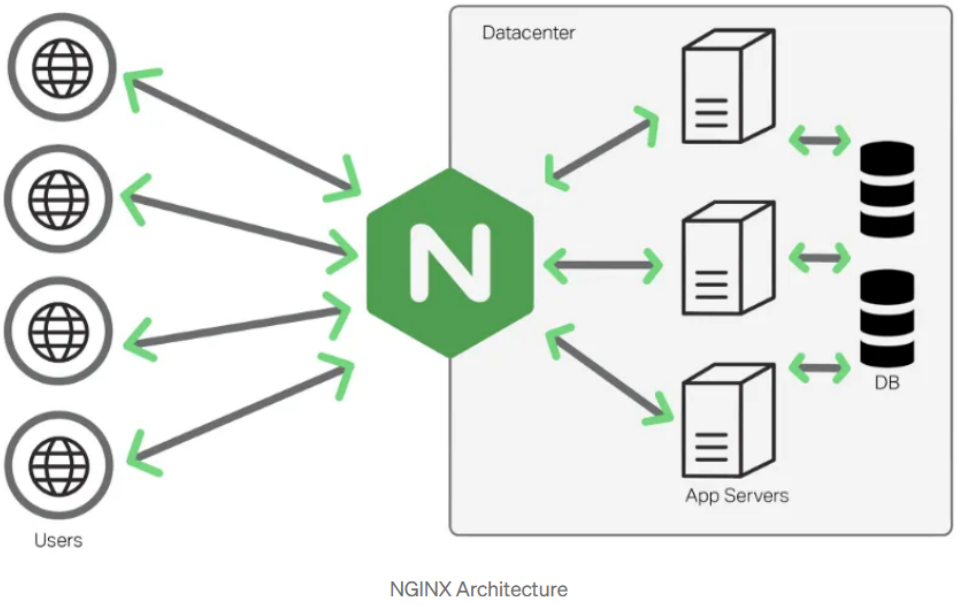

Figure 6. NGINX Architecture (Alesh, 2023).

Nginx is utilized in this project as a high performance reverse proxy and load balancer to improve traffic handling and system efficiency. Nginx, in this configuration, acts as an entry point for incoming requests and forwards them to the respective backend services, such as the Python app, Prometheus, and Grafana. This configuration not only improves response time but also enhances security by controlling access to internal services.

One of the significant strengths of Nginx is its load balancing, where traffic is distributed across several instances of the application. This helps to ensure that no service becomes overwhelmed with traffic, hence ensuring high availability and stability in the system. Nginx also provides SSL termination, which secures data transmissions by encrypting HTTP traffic. The inclusion of Nginx in this project simplifies the orchestration

of services and makes it scalable. Nginx helps optimize cloud resource usage and energy consumption through optimal traffic management and resource usage optimization.

4 MONITORING TOOL DEVELOPMENT

This chapter outlines the cloud resource monitoring and utilization system architecture deployed in this project. The system includes a number of components like Docker, Prometheus, Grafana, Nginx, and AWS that enable effective monitoring of resources, alerting, and optimization. The chapter elaborates in detail each of the components and offers information regarding implementation details, issues encountered, and solutions implemented during the development stage.

The project is designed as a containerized cloud deployable monitoring tool. This tool enables the user to analyze the metrics such as CPU usage, memory consumption, network bandwidth, and energy used by the app or services deployed in the docker. This system is integrated with the alert system which will triggered in case of high usages of CPU or memory. The user can also fix the high usage of the application by scaling up or scaling down via this tool.

4.1 User Interface Design

The user interface (UI) of the cloud monitoring solution is designed for an end to end and interactive dashboard to graph usage of cloud resources. The UI supports real time monitoring of performance metrics such as CPU, memory consumption, and utilization of energy at the service level. It also supports intelligent suggestions on how to optimize resources such as scaling up or down as per metrics observed.

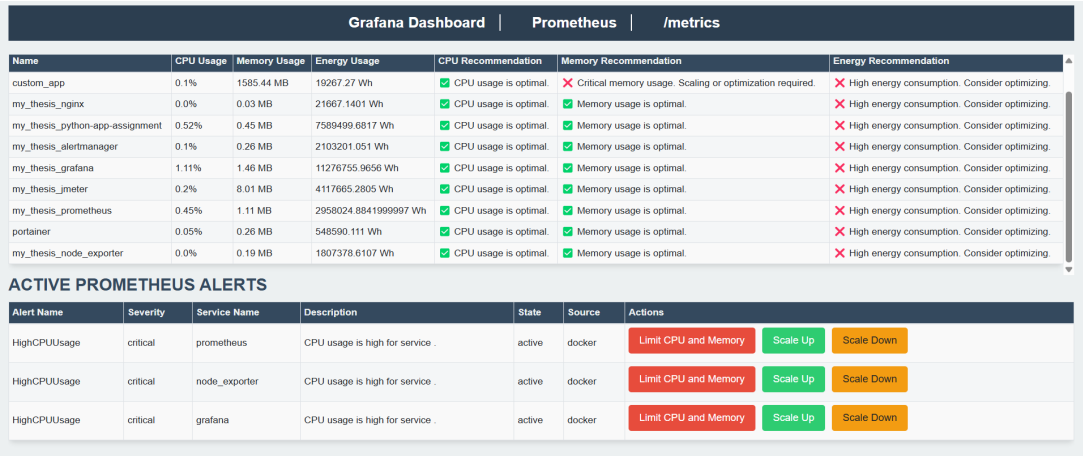

Figure 7. Main Frontend page

This interface is established on a responsive and modular design for optimal usability across devices and screen sizes. It is made in HTML, Bootstrap, and JavaScript, which provides the dynamism to present data and incorporate interactive elements. The core dashboard has various sections:

Service Monitoring Table displays real time information of CPU, memory, and energy usage for every running service. The table also provides recommendations based on the resource usage to optimize the use of resources.

Prometheus Alerts Section: This section is integrated with Prometheus Alert Manager and displays all running alerts and displays information of severity, service, and alert description.

Action Panel: Offers the capability to scale up or down services directly or apply resource limits to ensure optimal performance.

Navigation Menu: Provides direct links to Prometheus, Grafana, Custom Grafana dashboard, and the /metrics endpoint so users can view in depth analytics and system metrics.

The UI takes advantage of the asynchronous data fetching technique to qualify that the monitoring data updates dynamically without refreshing

the page. Furthermore, the use of Prometheus automates the handling of alerts, thus enabling users to get proactive alerting of system performance issues.

By providing an easily understandable layout with clear visual hierarchy, clear navigation and controls, the system enhances usability and productivity for cloud administrators to the best possible level. Providing real time suggestions and automated alerts significantly improves decision making for optimizing the utilization of cloud resources.

4.2 System Architecture

The design of the optimization and monitoring system has been made cloud native, scalable, and modular. The system is based on a containerized micro services architecture that is distributed and deployed with Docker Swarm to enable orchestration. The system design provides high availability, horizontal scalability, and automated deployment process, especially on cloud platforms such as Amazon Web Services (AWS).

At the heart of the system lies a set of containerized services executed as a Docker Stack. Services include a custom built Python monitoring tool, Prometheus, Grafana, and Node Exporter. All of these services are executed as Docker containers and orchestrated with Docker Swarm, which makes it simple to distribute and scale services across nodes.

The Python application is the core logic for collecting, processing, and presenting real time resource utilization statistics. It aggregates CPU utilization, memory utilization, and estimated energy utilization for all the running services. It also provides smart recommendations for scaling decisions based on predefined thresholds and heuristics. The data is presented through a particular HTML dashboard that shows service wise statistics, active Prometheus alerts, and includes actionable controls to scale services up or down.

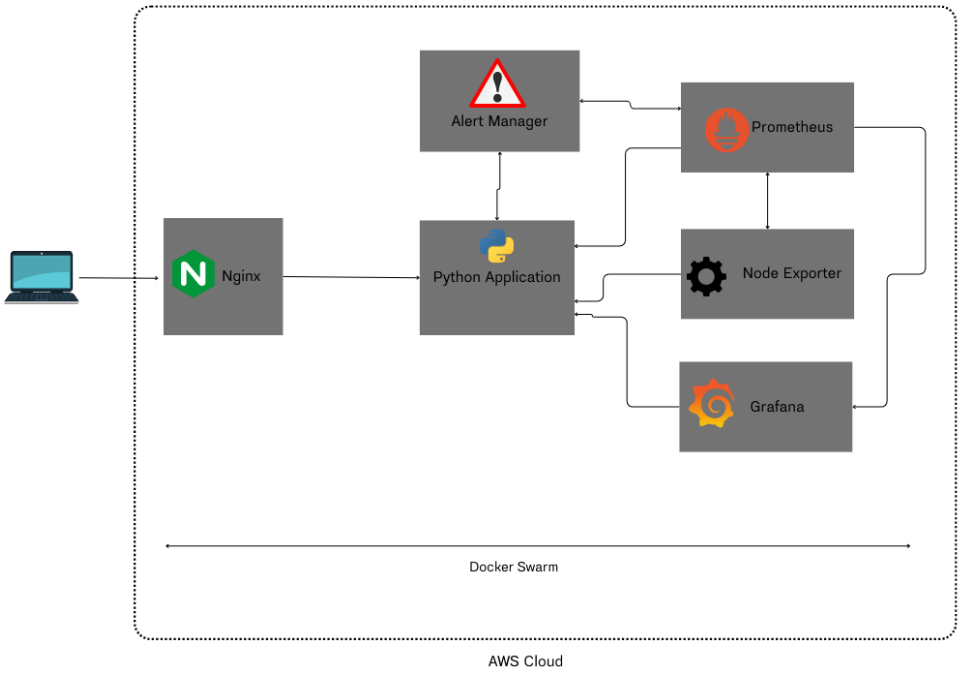

Figure 8. High level System Architecture

4.2.1 Containerization of System Components

For having a reproducible and consistent environment, the key components of the system are containerized using Docker. The Python application that monitors cloud resources is containerized so it runs reliably on various environments. Encapsulating the application along with its dependencies in a container prevents potential conflict with the host operating system and offers portability and deployment in various environments in an easier way.

Both Prometheus to collect cloud resource metrics and Grafana to visualize the metrics are containerized too. Docker makes it easy to deploy these services in such a manner that they become independent and supplied with all the dependencies available to execute properly. This container mechanism makes it very easy to deploy and manage the tools inside the cloud infrastructure and minimize configuration burden as well as enhance scalability.

4.2.2 Prometheus Metric Collection from Docker Services

Prometheus is the pillar of this project in the role of the primary tool to use in metric gathering and cloud resource usage measurement storage. Here, Prometheus exists as a Docker service in a Docker Swarm configuration, which promotes scalability and managing services. Being deployed as services, it makes scraping other Docker services deployed on the cloud environment for metrics uncomplicated.
All the principal operations, such as the bespoke Python monitoring application and supporting cloud based features, are run as Docker services that each publish a metrics endpoint which can be scraped by Prometheus. The service oriented architecture allows Prometheus to scrape real time performance data across the distributed system with very little configuration.

The Python service monitoring integration with Prometheus is pull based. The Python service executed using the Prometheus Python client library has an exposed HTTP endpoint which provides metrics in the Prometheus compatible format. Prometheus scrapes the endpoint periodically to capture real time metrics like CPU and memory consumption as well as estimates of energy consumed. Integration allows real time monitoring of system performance and optimized cloud resource usage.

Upon measurement collection, Prometheus enables efficient querying with PromQL (Prometheus Query Language).These measurements can then be rendered via Grafana dashboards for end-to-end monitoring and examination. Prometheus also collects infrastructure level metrics at the application's own application metrics, through exporters such as Node Exporter, and which are run as a Docker service on each node in the Swarm cluster. These exporters provide fine grained resource metrics such as CPU, memory, disk, and network use important in determining service health and in providing input to optimization.

With Docker Swarm assisting in managing Prometheus and all the support pieces as scalable services, the system ensures resilience, simple upgrades, and dynamic deployment across cloud platforms like AWS. Such a design enables real-time vision and data-driven decision-making to achieve maximum utilization of resources and energy efficiency in cloud native applications.

4.2.3 Grafana for Data Visualization

After Prometheus collects the necessary metrics, they are routed to Grafana for visualization. Docker simplifies deployment and scaling of Grafana in the cloud, enabling real time visualization of cloud resource usage and energy usage as a metric. Containerization of Grafana renders the tool portable and deployable into multiple environments without config problems.

The dashboards generated by Grafana make it possible to view patterns of resource utilization and energy usage, providing the insight required for maximizing cloud resources wherever they can be improved. The simplicity of deploying and scaling Grafana within a containerized environment ensures that visualization tools remain efficient and responsive despite the increasing volume of data being monitored.

4.2.4 Automated and Scalable Monitoring Setup

In this project, Docker Compose is used to deploy multi-service monitoring in orchestration and automation fashion. Docker Compose simplifies management of dependent services like Python monitoring application, Prometheus, and Grafana through defining them in a single file. The single file specifies the building, configuring, and network setup of each service, simplifying the setup and replicating process within development, staging, and production environments.

Using Docker Compose ensures that the entire monitoring environment can be initiated through the execution of one command. That one command launches all services ensuring Prometheus is able to scrape the metrics of the Python application, and Grafana can graph them within a connected, shared network.

Composé setup is also well suited for continuous deployment, scaling, and reuse. For instance, replicas can be scaled in or out using plain Docker commands and it is easy to be flexible when load testing or optimizing in production.

The automated setup provided by Docker Compose guarantees all the components of the system will be deployed to the system in a uniform fashion, reducing instances of misconfiguration and making easier system management on the whole.

4.3 Implementation

The implementaion phase of the envisioned monitoring system entails installation and deployment of a series of services whose combined contribution guarantees real-time monitoring, efficient data collection, and data visualization of cloud resource utilization. The most basic building blocks of the system are the Python application applied in monitoring, Prometheus for metrics collection, and Grafana for data visualization. All the components are containerized through the application of Docker in an effort to create a uniform, reproducible, and scalable environment. Here, the installation and integration of all the parts are discussed in detail, which include the use of Docker, Prometheus, Grafana, and the hosting of the system on AWS.

4.3.1 Setup and Configuration of Docker Services

The central tool to coordinate the services in this arrangement is Docker Compose, which is a multi container Docker application definition and

runtime utility. The docker-compose.yml file contains the secret to how all the services and their integration are configured. The following is a quick rundown of the key components and configurations in the Docker Compose arrangement.

The docker-compose.yml contains several services therein, such as Prometheus for monitoring, Grafana for visualization, and custom applications for metric analysis and collection. The following is the description of services mentioned in the configuration:

```
services:
 prometheus:
   image: prom/prometheus
   deploy:
     replicas: 1
     restart_policy:
       condition: any
   user: root
   ports:
     - "9090:9090"
   networks:
     - monitoring
   volumes:
     - docker_prometheus_data:/var/lib/prometheus
     - ./prometheus.yml:/etc/prometheus/prometheus.yml
     - ./alert.rules.yml:/etc/prometheus/alert.rules.yml
   command:
     - '--config.file=/etc/prometheus/prometheus.yml'
```

```
    - '--storage.tsdb.path=/var/lib/prometheus'
    - '--storage.tsdb.retention.time=150d'  # 5 months (adjust as needed)
    - '--storage.tsdb.retention.size=10GB'  # Optional: Limit disk usage
grafana:
  image: grafana/grafana
  deploy:
    replicas: 1
    restart_policy:
      condition: any
  ports:
    - "3000:3000"
  environment:
    - GF_SECURITY_ADMIN_USER=admin
    - GF_SECURITY_ADMIN_PASSWORD=admin
    - GF_SERVER_HTTP_PORT=3000
    - GF_SECURITY_ALLOW_EMBEDDING=true
  volumes:
    - grafana_data:/var/lib/grafana
    - ./nginx-logs:/var/log/nginx  # Mount logs directory
  networks:
   - monitoring
```

Figure 9. YAML Snippet for Docker Compose

The Python application in this project is containerized to ensure it operates consistently across various environments. The **Dockerfile** plays a key role in packaging the application along with its dependencies, ensuring that it runs inside a Docker container with minimal configuration and no dependency conflicts. Below is the Dockerfile used to build the Python application container:

```
# Use Python 3.10 as the parent image

FROM python:3.10.12-slim

# Set the working directory in the container

WORKDIR /app

# Copy requirements first for caching dependencies

COPY requirements.txt .

# Install required dependencies

RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application files

COPY . .

# Expose port 8000 for the Flask app and port 5000 for Prometheus metrics

EXPOSE 8000 5000

# Define environment variable for Flask app

ENV FLASK_APP=app.py

ENV PORT=8000

# Run the Flask app using Waitress

CMD ["waitress-serve", "--host=0.0.0.0", "--port=8000", "app:app"]
```

Figure 10. DockerFile Sample

4.3.2 Python for Metrics and Recommendations

To monitor the consumption of resources by containerized services and applications, a modified Python based recommendation engine and monitoring platform was employed. The application is responsible for querying metrics from Prometheus, analyzing them, and generating context-specific recommendations for resource balance.

The dynamic metrics including CPU, memory, and power consumption for both custom applications as well as Docker managed applications are fetched automatically through the use of the Python script. The application begins to iterate over an already defined list of application names. It extracts, for each application, from Prometheus on custom metrics cpu_usage, memory_usage, and energy_used.

The consumption of each application is compared to predetermined values. Based on the result, the script generates a level of recommendation for one of three levels: optimal, warning, or critical. For example, CPU usage below 70% is considered optimal, between 70% and 85% is a warning, and over 85% is flagged as critical, necessitating immediate intervention. The same logic applies to memory and energy consumption thresholds. Also, the app utilizes Docker's Python SDK to dynamically fetch all running services.

The resultant evaluations and recommendations are incorporated into a structured list of messages, then presented on the web interface of the system (as illustrated in Figure 7. Main Frontend Page). Users are able to take corrective action directly through buttons integrated in the interface, for example, scaling services up or down. This application enhances observability through correlating quantitative resource measures with qualitative decision-making support. The platform is a feedback mechanism that provides users with real time information and actionable advice, equivalent to eco-friendly and effective resource utilization patterns in cloud setups.

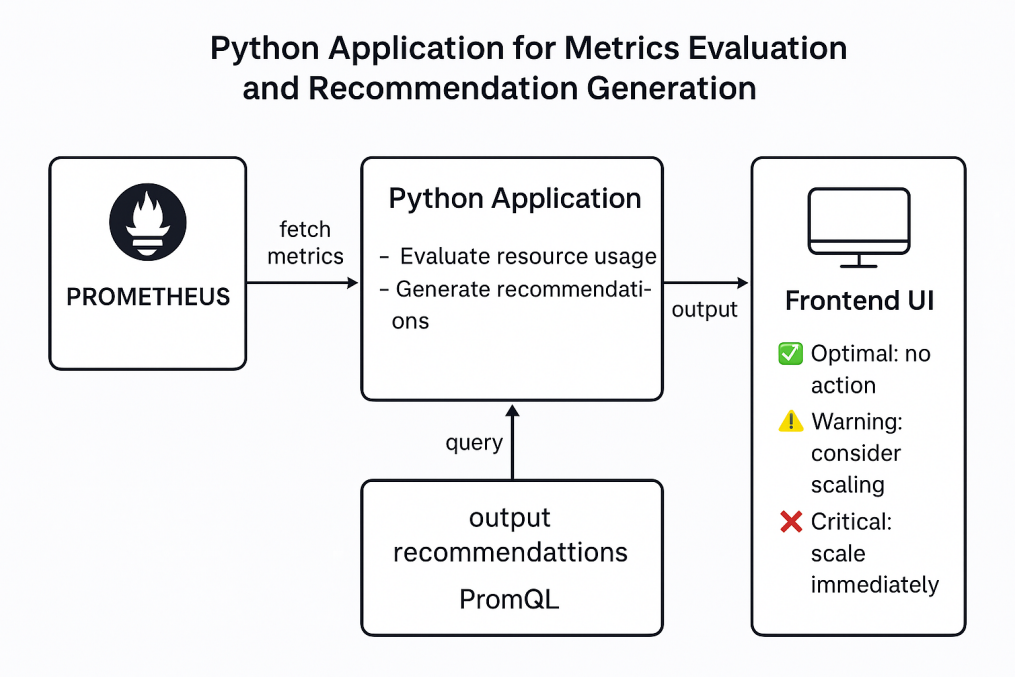

Figure 11. Python app integrated with Prometheus for generating recommendations

4.3.3 Metric Collection with Prometheus

As mentioned above section, Prometheus is integrated in python app and configured to scrape measurements from various services like the node_exporter service to obtain infrastructure-level measurements (CPU, memory, network, energy and disk) and the custom python-app-assignment service. Scraping occurs in PromQL so that it can be visualized and used to make decisions during optimization.

Prometheus configuration (prometheus.yml) is structured as follows to define the scraping targets:

```
scrape_configs:

  - job_name: 'custom_app'

  static_configs:

    - targets: ['python-app-assignment:8000']
```

Figure 12. Configuration of Prometheus

This configuration allows Prometheus to scrape the respective performance information of each service in the cloud infrastructure. Prometheus scrape the Python application for metrics at a predefined interval.

```python
@app.route('/metrics', methods=['GET'])

def metrics():

    """Expose the /metrics endpoint to Prometheus for scraping. Collects both custom app
 and Docker container metrics."""

    try:

        custom_metrics_data = custom_app_metrics.get_metrics()

        docker_metrics_data = docker_metrics.get_metrics()

        # Combine both custom app and Docker container metrics

        combined_metrics = custom_metrics_data + docker_metrics_data

        return Response(combined_metrics, content_type='text/plain')

    except Exception as e:

        logging.error(f"Error fetching metrics: {e}")

        return Response(f"Error fetching metrics: {str(e)}", status=500, c
        ontent_type='text/plain')
```

Figure 13. Implementation of collection of the metrics

Here, Prometheus will scrape the metrics from the Python app at a constant interval. The Python app has a /metrics endpoint that Prometheus consumes to collect real time performance metrics such as CPU, memory, network, disk and energy estimations. To facilitate this integration, the prometheus_client Python library is utilized to expose application metrics.

In addition to its role as the primary monitoring and metric scraping system, Prometheus also has a built in web based interface to facilitate data exploration, queryability at query time, and easy time series data visualization. This interface is used to complement the more advanced dashboards that are handled by Grafana by providing direct access to the native Prometheus query language (PromQL) and to the raw metric data.

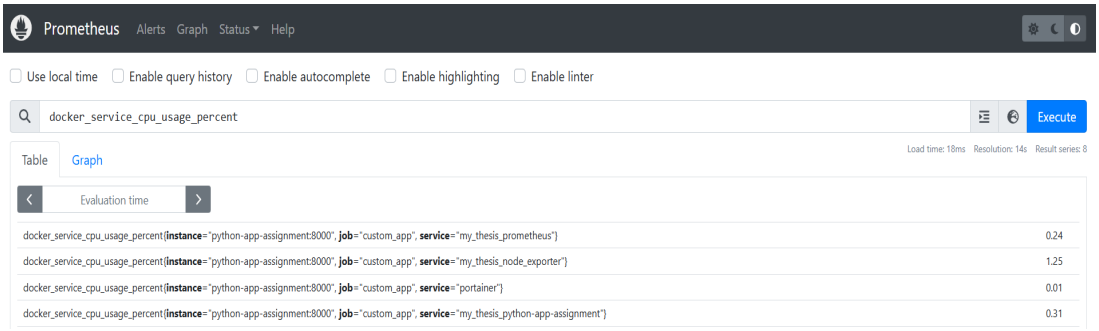

Figure 14. Interface of Prometheus displaying the metric data using PromQL

The Prometheus service is accessible via separate port on the host or cloud instance. The user interface offers system administrators and developers the functionality to run real-time queries, graph or table metrics, and debug metric exposure endpoints. It also offers utilities to navigate active targets, recording rules, alerting rules, and system state, including memory usage and uptime of the Prometheus service itself.

4.3.4 Visualization with Grafana

Once Prometheus collects the cloud resource metrics, visualization of the metrics is done with Grafana. Grafana collects data from Prometheus and enables the generation of dashboards that provide real time details of resource usage and energy consumption across the cloud infrastructure. By analyzing the the grafana dashboard, we can see whether the application usage is high or low between the time frame.

For example, Grafana queries were set to graph CPU usage:

```
docker_service_memory_usage_mb{service="my_thesis_python-app-assignment"}
```

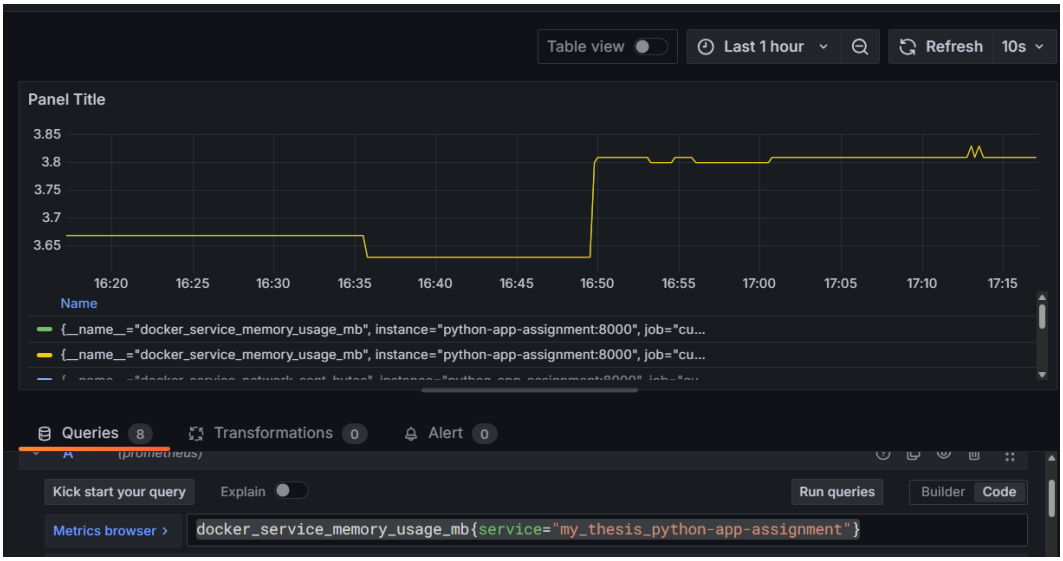

Figure 15. Interface of Grafana dashboard displaying the memory usage of the app by running the PromQL query

The integration between Grafana and Prometheus is straightforward. With Grafana running as a Docker container, the Prometheus data source is configured via the Grafana interface. In this thesis, the custom dashboards are subsequently created to chart CPU utilization, memory usage, network sent and receive, disk read and write, and energy consumption of the application. This facilitates real time monitoring and allows informed decisions on resource optimization. Grafana provides extremely strong querying using the application of PromQL (Prometheus Query Language) and allows one to extract important information from the aggregated metrics.

4.3.5 Prometheus Alerts

To enable active response to out of behavior system states, the entire system of Prometheus rules for alerting was created. Live performance anomalies like CPU high, memory high, and service or container downtime need rules to find and respond to these. Rules employ PromQL expressions in setting thresholds and when conditions under which alerts need to be triggered. The rules themselves are defined in a YAML configuration file that is referenced from the Prometheus config. The

rules themselves are organized by source and type of service being queried. Docker containers and Python custom apps, primarily. The alert conditions are queried at regular intervals, and notifications are sent via Alertmanager, which can, in turn, deliver messages via email, Slack, or other mechanisms.

Critical alerts were set up for the following:

High CPU Usage: Fired when CPU usage has exceeded 50% for Docker containers or 70% for custom applications for more than two minutes.

High Memory Usage: Triggered when memory usage has exceeded 500MB for Docker containers or 1GB for custom applications.

Container or Application Down: Triggered when a monitored service has been unavailable or not reporting metrics for five minutes continuously.

Service Specific Alerts: Additional alerts are configured for services like my_thesis_prometheus if their CPU usage exceeds 80%.

An example excerpt of the alert rules configuration is as below:

```yaml
groups:

- name: real_time_alerts

  rules:



  # High CPU Usage Alert for Any Docker Container

  - alert: HighCpuUsage

    expr: rate(container_cpu_usage_seconds_total{container_name=~".+"}[5m]) > 0.5

    for: 2m

    labels:

      severity: critical

      source: "docker"

    annotations:

      description: "High CPU usage detected for container {{ $labels.container_name }}."

      summary: "CPU usage exceeded 50% for container {{ $labels.container_name }}."
```

Figure 16. Alert Rules Example

In addition to the backend notifications sent through Alertmanager, the live alerts are shown directly on the top level front end interface of the monitoring tool. As seen from Figure 7 (Main Frontend Page), the alerts are shown in a tabular, well organized format with the service name, alert category, resource usage values (CPU, memory), and status.This integration enables system developers and operators to see crucial information at a glance without the need to switch to standalone dashboards. Each alert row is succeeded by interactive action buttons that allow users to respond to the issue immediately for example, by triggering scaling operations (up/down), restarting a container, or accepting the alert. This seamless integration of detection and resolution not only improves response time but also allows for more self managing and user driven management of cloud resources. The frontend is easy to use, such that even technical stakeholders can monitor and intervene as required with minimal support.

4.3.4 Deployment on AWS

Amazon Web Services (AWS) facilitates strong containerized application deployment through the implementation of Docker services. Strong integration between AWS and Docker offers ease of deployment and execution of services in the cloud infrastructure. Execution of services through Docker services as opposed to distinct containers enhances orchestration, automatic load balancing, service discovery, and fault tolerance, thus better managing cloud resources. Usage of Docker services is part of the vision for high utilization of cloud resources and lowering the energy spent. The application components can be scaled, duplicated, and dispatched dynamically to AWS resources using service deployment as opposed to containers.

The monitoring system developed in this project comprising of a Python application, Prometheus, and Grafana was containerized and executed as Docker services. Python monitoring application Docker images, Prometheus and Grafana were developed and pushed to Docker Hub in this project, so that reusable and centralized access could be made available for containerized components.

To deploy, docker-compose.yml was uploaded securely along with all the configurations of the services and their dependencies onto an AWS Elastic Compute Cloud (EC2) server. AWS server Docker Compose installed the complete monitoring stack using docker compose up -d. This way, numerous dependent services can be orchestrated using a single configuration file to offer management simplicity and deployment consistency. Through executing services using Docker Compose on AWS, the environment gets to utilize resources effectively and achieve isolation. The strategy is also compatible with energy efficiency targets because the containers are easy to manage tightly, get started quickly, and are scaled on demand, cutting unused resource use. The strategy provides an end to end reproducible, uniform, and maintainable

deployment process with adherence to best practices in today's cloud native application development.

5 TESTING AND ANALYSIS

This chapter discusses the testing process utilized to justify the reliability, efficiency, and precision of the engineered cloud resource monitoring and optimization framework. The system was comprehensively tested in terms of performance, response under load, as well as accuracy of monitoring data using automated test tools and performance simulation. The goal of this testing phase was not only to ensure the system worked as intended but also to ensure that it was able to handle realistic workloads while maintaining proper reporting of resource utilization and responsiveness.

Testing is a critical aspect of developing reliable cloud based systems, particularly when the system is intended to guide users to efficient use of resources and energy conscious decision making. The tests were developed to measure responsiveness of the system, resource usage, and overall performance while keeping energy efficiency in mind all the time.

This chapter further explains the testing process, the different environments used, the tools used, and how the feedback was used to improve the system. By the end of this chapter, we will illustrate how the testing activity contributed towards the development of a healthy, energy efficient, and scalable cloud resource optimization solution.

5.2 Testing Aproach

The current sub section represents the governmental procedure utilized for certification of performance, reliability, and precision of cloud resource optimizing and monitoring system used in the given research work. The most imperative reason to go through testing procedures was measurement of the amount of system's performance for resource optimizing, its tracking, as well as limiting its energy intake. Performance testing using load simulation was conducted to identify

system response to the continuously changing levels of stress in an effort to make the framework respond suitably to various scenarios of real usage by the cloud.

The testing process was designed to fulfill a number of significant objectives. First, it was to ensure that the system behaved as intended under different conditions and scenarios. This included ensuring that the system functioned correctly both in normal and stressed modes. The second objective was to ensure the correctness and precision of resource utilization statistics generated by the system, which is imperative to resource optimization and energy conservation efforts.

Second, the test process was intended to evaluate the extent to which the system would perform when subjected to heavy loads characterized by high volumes of user demands and cloud resource loads. Finally, the test process tested the real time response of the system at various loads of user requests to verify that the system was capable of maintaining stability and responsiveness under stress in operations. The testing procedure entailed an iterative process with continuous testing loops with possible issue fixing incorporated to mold the framework.

5.2 Test Environments

In order to conduct full scale testing and validation, the infrastructure for optimizing and monitoring cloud computing resources was tested and validated in three test environments selected to mimic different operating environments and deployment choices. These environments were used in system testing of the flexibility, scalability, and responsiveness of the system in real environments.

Local deployment was applied as an enclosed and isolated context for initial system debugging and testing. The environment permitted quick iteration and extensive observation of system entities without the system's complexity through large scale deployments. The Docker deployment environment mimicked a transportable and extensible version of the system through wrapping it around Docker containers.

This environment simulated cloud deployment scenarios to enable tests to measure how the system interacted with multiple services and scaled in a containerized context. Finally, the AWS cloud environment provided the closest to real world testing scenario, in which the system was put to test with big scale cloud resource management. The environment enabled testing the optimization strategies of the system in a production level cloud environment to verify that it is capable of efficiently handling cloud based operations.

Testing the system across these three disparate environments ensured that it was flexible enough to accommodate a variety of deployment scenarios and could handle different scales of operations. Each environment was necessary for verifying the performance of the system, from local, controlled environments to very large cloud based deployments.

5.3 Tool Used For Testing

The testing phase was driven by a set of specialist software to enable the simulation of cloud usage patterns, system performance analysis, and resource utilization. The tools played a critical role in the creation of realistic traffic, evaluation of the scalability of a system, as well as the verification of energy optimization targets.

Apache JMeter was employed as the primary system load test tool. Apache JMeter had the feature of submitting concurrent requests for users concurrently and simultaneously, which was great to see what would happen when various loads are provided to the system. According to JMeter, the response rate of the system, resource utilization, and how the system reacts when given the system in loaded conditions were fully tested, primarily for load testing. These tests were highly instrumental during tests for the responsiveness and scalability of the framework.

Additionally, Prometheus and Grafana that had been integrated into the system, also provided important data for evaluation and for providing

the metrics captured in the testing stage, which maximizes the use of the resources and minimizes energy consumption. Data visualization of resource usage by the tool ensured that energy usage and distribution of resources were being maximized as per the goal of the system.

Together, these tools facilitated a comprehensive and intensive testing process. They provided the required information and data to test the system behavior under different conditions of loading as well as avoid energy consumption with optimal resource utilization.

5.4 Summary of Testing and Evaluation

The sequence of test experiments conducted throughout the development cycle of the cloud resource monitoring and optimization framework was instrumental in establishing the effectiveness, reliability, and performance of the system. With the implementation of a multi environment testing approach encompassing local environments, Dockerized environments, and a cloud based AWS platform, the framework was rigorously tested across a broad spectrum of operational scenarios. This facilitated the identification and resolution of environment specific performance issues, thereby ensuring the system's resilience and flexibility across deployment environments.

Apache JMeter deployment provided vital perspective on the system's performance with varying and concurrent user loads. The simulations helped in performance benchmarking and found the system to be stable, responsive, and scalable at higher level operations. This also helps to analyze increase in the usage of the metrics in case of the heavy user loads. Besides that, with the integration of Prometheus and Grafana, real time monitoring and visualization of crucial performance metrics such as CPU, memory, and power consumption were possible, thereby reaffirming the design goals of the framework being energy aware.

Overall, the experimentation activity showed that the proposed framework functions well in real and dynamic cloud settings, with

accurate monitoring data and correct optimization suggestions. The results confirm the feasibility of the framework as an energy aware and scalable solution for cloud resource management.

6 CONCLUSION AND FUTURE WORK

This chapter recapitulates the results of the research work conducted in performance evaluation of the cloud resource monitoring and optimization framework developed during this thesis work. It highlights the key findings, argues on the weaknesses and strengths of the proposed system, and suggests possible directions for future improvement. By discussing the research deliverables and objectives, this chapter gives an insight into the effectiveness of the framework and how it can help achieve energy aware cloud computing. It also gives the key issues for future research, including areas that can be enhanced in the performance of the system, its scalability, and flexibility to accommodate emerging technologies.

6.1 Summary of Findings

This chapter summarizes the findings of the study with a focus on how the cloud resource monitoring and optimization framework achieved the overall objectives set at the beginning of the thesis. The overall goal of the research was to develop a system that is capable of optimizing cloud resource use with the intention of minimizing energy consumption. Through the use of monitoring techniques and optimization algorithms, the system was able to identify cloud resource inefficiencies and provide effective recommendations against unnecessary energy usage.

The strategy used in the thesis included the development of a multi-environment test approach (local, Docker, and AWS environment) to verify the system's performance under different deployment scenarios. The performance testing carried out using Apache JMeter and the real time monitoring with Prometheus and Grafana demonstrated the system's ability to process varying traffic volumes and provide accurate resource utilization statistics. The system was found to be effective for resource utilization monitoring, with visualizations that facilitated energy-efficient decision making.

Among the major outcomes was the ability of the framework to provide real time data on energy consumption patterns of cloud services, which could be used by users to maximize resource utilization without affecting system performance. Regular incorporation of feedback during testing was a key element that contributed to refining the design of the framework and making the tool reliable and user friendly.

6.2 Limitations

While the framework met the important objectives, several limitations were achieved during development and testing. The limitations influenced the performance and coverage of the system, and they provide valuable information for the future development.

- Metrics Coverage: The model cared more about live resource monitoring (CPU, memory, and power usage) but not necessarily beyond that to more analytics or full metric coverage. For instance, it was not all performance metrics such as disk I/O or network bandwidth that could also affect cloud resource optimization choices.

- Predictive Capability: The system at present lacks the ability to predict expected future utilization of resources based on past experience. A prediction based model using machine learning would significantly enhance the system by predicting peak loads and pre-allocating resources to avoid issues beforehand.

- Tool Dependence: The framework relied on some tools and platforms, for example, Prometheus for monitoring and Grafana for visualization. Though these are great tools, they have dependencies that limit flexibility. For example, the performance of the system is inextricably bound up with Prometheus and Grafana's configuration and performance, and introduction of other tools or a change of platform would involve radical changes to the structure.

- Scalability: Although the system performed adequately in environments it was tested, there was some processing limitation on processing extremely gigantic-scale cloud instances with hundreds of thousands of services. The architecture would have to be maximized in the aspect of enormous scales in order to handle colossal datasets without compromising speed.

- Energy Consumption Data: While the system provided estimates of energy consumption, these estimates were derived from data that would or would not have been present. In cloud systems where the energy data are not directly monitored by all the cloud service providers, estimates had to be done, which may introduce slight discrepancies in the actual consumption reported.

6.3 Future Work

There are several paths of future work that can continue to develop on the current framework and overcome some of the following limitations. Such advancements can continue to enhance the ability of the system to maximize cloud resource usage and support more complex decision making.

Integrating Machine Learning to Predictive Resource Optimization: The most promising potential area of activity in the future is integrating machine learning algorithms towards predictive resource optimization. From the past data and usage patterns, machine learning algorithms can forecast upcoming resource requirements, and the system can dynamically configure resource allocation and scaling ahead of bottlenecks or performance decline occurring. This would lead to even more optimal cloud resource utilization and even greater energy wastage reduction.

Enabling Multi cloud and Hybrid Deployments: Although the framework has been applied to single cloud environments (AWS), the effort would be done to enable deployments in multi cloud and hybrid clouds.

Organizations would more and more deploy in multiple clouds to avoid vendor lock in, as well as to optimize for cloud infrastructure. It would enable the framework to be more versatile and beneficial in a wider audience of organizations in order to enable multiple platforms of clouds simultaneously.

Include Historical Trend Analytics: The application may be supplemented with analytics functionality in order to keep track of and compare historical resource and energy use trends. It would offer greater insight into the long term trend and enable wiser decisions with regard to deployment of cloud resources. Historical trends can also be utilized for capacity planning and benchmarking in such a way that businesses are able to utilize cloud more effectively over the long run.

Enhancing the UI/UX of the Dashboard: The UI/UX of the dashboard for tracking can be enhanced in a manner that it will become more convenient and user friendly to use. Although the dashboard does show visualizations of usage of cloud resources, the regions of layout, interactivity, and availability of information require enhancement. For example, with more filtering power, more configurable views, and drilling to individual data points, would provide even more control to the end user by the dashboard.

Integration with Serverless Architectures: Integration of the framework with serverless platforms is another area of growth because serverless platforms are becoming more popular in the cloud. Serverless architectures are particularly challenging to resource optimize because they are dynamic. Incorporating support for serverless platforms such as AWS Lambda or Azure Functions would bring the utility of the framework into present day cloud computing environments.

Energy Efficiency in Containerized and Orchestrated Systems: As the number of containerized applications and usage of orchestrations platforms like Kubernetes increases, optimizing energy consumption, particularly in containerized and orchestrated systems, could be one

direction for future work. That would involve further enhancing the framework to optimize resource utilization at the container level and facilitate efficient execution of cloud native applications.

6.4 Conclusion

Overall, this thesis suggested the development of a cloud resource monitoring and optimization framework to improve energy efficiency in cloud systems. The system could accomplish its primary goals through real time monitoring of resource usage and providing insights into potential energy savings. In spite of some flaws, such as the lack of predictability and reliance on some tools, the system was worth it in promoting the general movement towards cost effective and sustainable cloud computing.

The work points to growing significance of energy conscious decision making in the field of cloud computing, where the growing demand for services is prone to convert into higher energy use and environmental impact. By presenting a simple solution to implement in cloud resource optimization, this system can potentially help organizations reduce their carbon footprint while reducing operational costs simultaneously.
Follow up research in server less optimization, multi cloud enablement, and machine learning can further potentially enhance the framework's reach and make it more appropriate and scalable for actual cloud environments. Lastly, continuous innovation in power efficient cloud computing solutions will become vital to catalyzing sustainability within the rapidly expanding cloud services industry.

REFERENCES

A, P., Sangeetha, S., & P, S. (2024). Performance optimization and energy minimization of cloud data center using optimal switching and load distribution model. *Sustainable Computing Informatics and Systems*, *43*, 101013. https://doi.org/10.1016/j.suscom.2024.101013

Abdel-Basset, M., Mohamed, R., Elkhalik, W. A., Sharawi, M., & Sallam, K. M. (2022). Task scheduling approach in cloud computing environment using hybrid differential evolution. *Mathematics*, *10*(21), 4049. https://doi.org/10.3390/math10214049

Abid, A., Manzoor, M. F., Farooq, M., Farooq, U., & Hussain, M. (2020). Challenges and issues of resource allocation techniques in cloud computing. *KSII Transactions on Internet and Information Systems*, *14*(7). https://doi.org/10.3837/tiis.2020.07.005

Alesh, S. (2023, October 3). What is Nginx? - Sami Alesh - Medium. *Medium*. https://medium.com/@sami.alesh/what-is-nginx-7db76b2e79f8

Alsadie, D., & Alsulami, M. (2024). Efficient Resource management in cloud environments: A modified feeding birds algorithm for VM consolidation. *Mathematics*, *12*(12), 1845. https://doi.org/10.3390/math12121845

Arshad, U., Aleem, M., Srivastava, G., & Lin, J. C. (2022). Utilizing power consumption and SLA violations using dynamic VM consolidation in cloud data centers. *Renewable and Sustainable Energy Reviews*, *167*, 112782. https://doi.org/10.1016/j.rser.2022.112782

Baranenko, I. (2025, February 13). *Multi-cloud cost optimization: Benefits and challenges*. Spendbase. https://www.spendbase.co/blog/multi-cloud-cost-optimization/

Buyya, R., Ilager, S., & Arroba, P. (2023). Energy-efficiency and sustainability in new generation cloud computing: A vision and directions for integrated management of data centre resources and workloads. *Software Practice and Experience*, *54*(1), 24–38. https://doi.org/10.1002/spe.3248

Comparison of static and dynamic resource allocation strategies for matrix multiplication. (2015). In *26th IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. https://inria.hal.science/hal-01163936v2

Global data center electricity use to double by 2026 - IEA report. (2024, January 26). DCD. https://www.datacenterdynamics.com/en/news/global-data-center-electricity-use-to-double-by-2026-report/

Hassan, K. M., Abdo, A., & Yakoub, A. (2022). Enhancement of health care services based on cloud computing in IOT environment using hybrid swarm Intelligence. *IEEE Access*, *10*, 105877–105886. https://doi.org/10.1109/access.2022.3211512

Hijji, M., Ahmad, B., Alam, G., Alwakeel, A., Alwakeel, M., Alharbi, L. A., Aljarf, A., & Khan, M. U. (2022). Cloud servers: resource optimization using different energy saving techniques. *Sensors*, *22*(21), 8384. https://doi.org/10.3390/s22218384

Junaid, M., Sohail, A., Rais, R. N. B., Ahmed, A., Khalid, O., Khan, I. A., Hussain, S. S., & Ejaz, N. (2020). Modeling an optimized approach for load balancing in cloud. *IEEE Access*, *8*, 173208–173226. https://doi.org/10.1109/access.2020.3024113

Katal, A., Dahiya, S., & Choudhury, T. (2022). Energy efficiency in cloud computing data centers: a survey on software technologies.

Cluster Computing, *26*(3), 1845–1875. https://doi.org/10.1007/s10586-022-03713-0

Kerner, S. M. (2021, March 1). *autoscaling*. Search Cloud Computing. https://www.techtarget.com/searchcloudcomputing/definition/autoscaling

Khalid, A., Ain, Q. U., Qasim, A., & Aziz, Z. (2021). QOS based optimal resource allocation and workload balancing for FOG enabled IoT. *Open Computer Science*, *11*(1), 262–274. https://doi.org/10.1515/comp-2020-0162

Liu, X., Cheng, B., & Wang, S. (2020). Availability-Aware and Energy-Efficient virtual cluster allocation based on Multi-Objective optimization in cloud datacenters. *IEEE Transactions on Network and Service Management*, *17*(2), 972–985. https://doi.org/10.1109/tnsm.2020.2975580

Liu, Y., Wei, X., Xiao, J., Liu, Z., Xu, Y., & Tian, Y. (2020). Energy consumption and emission mitigation prediction based on data center traffic and PUE for global data centers. *Global Energy Interconnection*, *3*(3), 272–282. https://doi.org/10.1016/j.gloei.2020.07.008

Michel, M. (2025, January 7). *Data centres could use 12% of US electricity within 4 years*. CSO Futures. https://www.csofutures.com/news/data-centres-could-consume-12-of-us-electricity-within-4-years/

Moura, B. M., Schneider, G. B., Yamin, A. C., Santos, H., Reiser, R. H., & Bedregal, B. (2021). Interval-valued Fuzzy Logic approach for overloaded hosts in consolidation of virtual machines in cloud computing. *Fuzzy Sets and Systems*, *446*, 144–166. https://doi.org/10.1016/j.fss.2021.03.001

Mousavi, S., Mosavi, A., Varkonyi-Koczy, A. R., Fazekas, G., Faculty of Informatics, University of Debrecen, Institute of Automation, Kando Kalman Faculty of Electrical Engineering, Obuda University, Norwegian University of Science and Technology, Department of Computer Science, Institute of Structural Mechanics, Bauhaus Universität-Weimar, & Department of Mathematics and Informatics, J. Selye University. (n.d.). Dynamic resource allocation in cloud computing. In *Acta Polytechnica Hungarica* (pp. 1–3).

Pandey, A. K., & Ahmad, S. (2019). Energy Optimization in Cloud Computing A review. *International Journal of Computer Sciences and Engineering*, *7*(2), 249–256. https://doi.org/10.26438/ijcse/v7i2.249256

Prometheus. (n.d.). *Overview | Prometheus*. https://prometheus.io/docs/introduction/overview/

Reback, G. (2021, February 16). An intro to PromQL: Basic Concepts & Examples. *Logz.io*. https://logz.io/blog/promql-examples-introduction/

Seth, D., Nerella, H., Najana, M., & Tabbassum, A. (2024). Navigating the Multi-Cloud Maze: Benefits, Challenges, and Future Trends. *Navigating the Multi-Cloud Maze: Benefits, Challenges, and Future Trends*. https://doi.org/10.21428/e90189c8.8c704fe4

Slingerland, C. (2024, April 24). *The simple guide to the history of the cloud*. CloudZero. https://www.cloudzero.com/blog/history-of-the-cloud/

"What is Docker?" (2024, September 10). Docker Documentation. https://docs.docker.com/get-started/docker-overview/

Wikipedia contributors. (2025a, February 4). *Grafana*. Wikipedia. https://en.wikipedia.org/wiki/Grafana

Wikipedia contributors. (2025b, March 10). *Docker (software)*. Wikipedia. https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=1279771096

Zhang, J., Yu, H., Fan, G., Li, Z., Xu, J., & Li, J. (2024). Handling hierarchy in cloud data centers: A Hyper-Heuristic approach for resource contention and energy-aware Virtual Machine management. *Expert Systems With Applications*, *249*, 123528. https://doi.org/10.1016/j.eswa.2024.123528

Zheng, H., Xu, K., Zhang, M., Tan, H., & Li, H. (2024). Efficient resource allocation in cloud computing environments using AI-driven predictive analytics. *Applied and Computational Engineering*, *82*(1), 6–12. https://doi.org/10.54254/2755-2721/82/2024glg0055