High-Confidence Computing 5 (2025) 100276

Contents lists available at ScienceDirect

High-Confidence Computing

journal homepage: www.sciencedirect.com/journal/high-confidence-computing

Review article

Kubernetes application performance benchmarking on heterogeneous CPU architecture: An experimental review

Jannatun Noor, MD Badsha Faysal, MD Sheikh Amin, Bushra Tabassum, Tamim Raiyan Khan, Tanvir Rahman [*]

Computing for Sustainability and Social Good (C2SG) Research Group, Department of Computer Science and Engineering, School of Data and Sciences, United International University, Dhaka 1212, Bangladesh

ARTICLE INFO

Article history:
Received 9 January 2024
Revised 14 May 2024
Accepted 1 September 2024
Available online 18 December 2024

Keywords:
Kubernetes
K8s
K3s
Serverless computing
Container as a Service (CaaS)
Container orchestration
Docker
CPU architecture
x86
ARM

ABSTRACT

With the rapid advancement of cloud technologies, cloud services have enormously contributed to the cloud community for application development life-cycle. In this context, Kubernetes has played a pivotal role as a cloud computing tool, enabling developers to adopt efficient and automated deployment strategies. Using Kubernetes as an orchestration tool and a cloud computing system as a manager of the infrastructures, developers can boost the development and deployment process. With cloud providers such as GCP, AWS, Azure, and Oracle offering Kubernetes services, the availability of both x86 and ARM platforms has become evident. However, while x86 currently dominates the market, ARM-based solutions have seen limited adoption, with only a few individuals actively working on ARM deployments. This study explores the efficiency and cost-effectiveness of implementing Kubernetes on different CPU platforms. By comparing the performance of x86 and ARM platforms, this research seeks to ascertain whether transitioning to ARM presents a more advantageous option for Kubernetes deployments. Through a comprehensive evaluation of scalability, cost, and overall performance, this study aims to shed light on the viability of leveraging ARM on different CPUs by providing valuable insights.

1. Introduction

In recent years, the landscape of cloud computing has witnessed remarkable progress and a rapid influx of new services and technologies [1–4]. This dynamic environment constantly shapes how we interact with the web, necessitating exploring innovative approaches to meet the evolving demands of modern applications. Among the myriad of cloud technologies, Kubernetes has emerged as a leading tool for cloud computing, demonstrating its prowess in orchestrating containerized applications [5]. As developers strive to optimize their deployment strategies, Kubernetes has become a powerful enabler of horizontal scaling and efficient resource management in cloud environments.

Many services are now available on top of the cloud, such as Software as a Service (SaaS), Infrastructure as a Service (IaaS), and Function as a Service (FaaS) [6]. Most cloud providers provide automated services that reduce work from the developer's end. As Kubernetes gains more and more popularity, major cloud providers, including Google Cloud Platform (GCP), Amazon Web Services (AWS), Microsoft Azure, and Oracle Cloud Infrastructure (OCI), have taken notice of its growing importance and are now offering their own Kubernetes services [7]. Kubernetes is now available as a Product as a Service (PaaS) [8–10]. These services of PaaS provide developers with a managed platform to leverage the robust capabilities of Kubernetes for efficient container orchestration. Consequently, organizations are increasingly leveraging Kubernetes as a foundational technology to build their cloud-native architectures and enhance the scalability and reliability of their applications.

1.1. Background

Containerization, mainly through Docker, has become a popular trend in application development and deployment [11]. While it may require additional steps to create container images and Docker files for complex applications, containerization offers long-term time savings for multi-device deployments. Docker-compose simplifies the deployment process and enables easier scripting. However, manual load balancing of multiple containers and deploying them on multiple instances can be time-consuming [12]. Kubernetes automates load balancing and management of containers, known as pods, making it an efficient solution for deploying and scaling applications [11].

* Corresponding author.
E-mail address: tanviranindo@gmail.com (T. Rahman).

https://doi.org/10.1016/j.hcc.2024.100276

Kubernetes supports various container images, including Docker, facilitating seamless integration and application management. In a Kubernetes system, a master node and one or more worker nodes are necessary, with multiple master nodes ensuring high availability and failover, employing diverse CPU architectures such as x86 or ARM [13]. Besides, the x86 architecture, introduced with the 8086 in 1978, has become the dominant architecture, supporting 16-bit, 32-bit, and 64-bit systems [14]. However, the growing popularity of ARM processors in cloud computing introduces new considerations for Kubernetes, as ARM offers power efficiency and adaptability [14]. Evaluating Kubernetes performance on both x86 and ARM architectures becomes essential for determining the optimal approach for serverless computing, considering efficiency and cost-effectiveness [15].

1.2. Motivation

The dynamic environment created by the rapid advancements and growing popularity of cloud computing necessitates innovative approaches to address the evolving needs of modern serverless applications. Kubernetes has emerged as a leading tool for orchestrating containerized applications, offering horizontal scaling and efficient resource management capabilities. However, as the landscape of cloud computing expands, it becomes essential to evaluate the performance and suitability of Kubernetes on different architectures.

While x86 platforms have traditionally dominated the market, the rise of ARM processors has introduced new possibilities in cloud computing. ARM processors are known for their power efficiency and adaptability, making them increasingly popular in mobile telecommunication devices and serverless computing scenarios [16]. Therefore, understanding the performance and cost implications of deploying Kubernetes on both x86 and ARM architectures is essential for organizations seeking the most efficient and cost-effective implementation for serverless computing. This study addresses this critical question by conducting comprehensive benchmark tests and analyzing key metrics such as response times, throughput, and resource consumption. By evaluating Kubernetes' performance, scalability, and resource utilization on x86 and ARM platforms, we seek to provide valuable insights for organizations in making informed decisions regarding their cloud infrastructure. Additionally, cost analysis, including factors like pricing models and resource availability provided by cloud providers, will contribute to assessing the cost-effectiveness of each solution.

1.3. Research objectives and contributions

Our primary goal is to demonstrate to the developer community the versatility of Kubernetes analysis across various programming languages and architectures. This enables them to comprehend the price-to-performance ratio and the broader ecosystem. Based on our study, we make the following contributions in this paper:

- We discuss different cloud providers and different programming languages that provide the service of Kubernetes and understand the methods on which we test x86 and ARM platforms.
- Next, we calculate and analyze the cost for three prominent public cloud providers, namely GCP, AWS, and OCI cloud providers, and present a cost-effective platform.
- Moreover, we use Express, Flask, Gin, and Actix API to benchmark Kubernetes systems to provide real-time results and facilitate performance evaluation.
- Finally, we measure Kubernetes systems using an actual test-bed setup using several performance metrics (CPU performance, throughput, success rate, memory utilization, and I/O operations). These measurements provide a comprehensive understanding of system performance and efficiency.

1.4. Paper organization

We divide our paper into several sections to enhance its organization. Section 2 compares the different public cloud providers and explores the current scenario of serverless computing systems worldwide for Kubernetes. Following that, in Section 3, we address the classification of different CPU platforms and architectures and discuss the advantages and disadvantages of these architectures. Moreover, we discuss virtualization and containerization concepts. Section 4 contains the use cases, performance results, comparison, and analytical discussion of Kubernetes on different platforms based on the survey findings. Then in Section 5, we present our methodology, testing plan, mechanism of deploying clusters, and implementation of different API tools. Section 6 elaborates on the results of four APIs on four platforms, explicitly focusing on performance and cost. Furthermore, we discuss the possibilities and challenges of testing and deploying production-grade applications in Section 7. Finally, we state the impacts of using Kubernetes and the conclusion and prospects of this research in Section 8.

2. Public cloud providers

We present an overview of several cloud service providers to outline the configuration of our test environment and to evaluate the performance of the implemented system. We have access to multiple cloud providers for deploying and testing the experiment. In the following section, we detail five cloud providers that actively support Kubernetes deployment.

Google Cloud Platform: GCP provides serverless computing environments, Platform as a Service (PaaS), Container as a Service (CaaS), Software as a Service (SaaS), and Infrastructure as a Service (IaaS) [17,18]. The "Cloud Run" service offered by GCP allows for the creation of serverless computing systems [19]. However, before utilizing this service, the Kubernetes application needs to be containerized using Docker to package and include dependencies within a container. Subsequently, it can be deployed on either a fully managed container or a Google Kubernetes Engine (GKE) cluster on Anthos, a hybrid cloud management tool provided by GCP for managing clusters [20]. Cloud Run enables the execution of HTTP and event-based triggers, automatically scaling applications based on incoming requests. This functionality can contribute to cost reduction, and efficient resource management, and offers features such as logging and monitoring [21].

Amazon Web Services (AWS): AWS has been a significant contributor to the development of serverless computing systems, offering a comprehensive suite of services and tools. Particularly noteworthy is AWS Lambda, introduced by AWS, which has played a pivotal role in advancing the field of serverless computing [22]. AWS Lambda boasts key features such as event-driven execution, automatic scalability, pay-as-you-go cost management behavior, and seamless integration with other AWS services.

Among the essential features of AWS's serverless services are API Gateway, Step Functions, and DynamoDB. To implement serverless services for Kubernetes, AWS Fargate is utilized, serving as a compute engine for Amazon Elastic Kubernetes Service (EKS) [23]. By leveraging Fargate, there is no need for manual configuration of servers or Kubernetes pods and containers; Fargate automates the application building and deployment processes. Utilizing Fargate within EKS enables the serverless execution of

Table 1
Feature comparison of five public cloud platforms [6].

Feature	GCP	AWS	OCI	IBM	Azure
Service	IaaS, PaaS, SaaS	IaaS, PaaS, SaaS	IaaS, PaaS, SaaS	IaaS, PaaS, SaaS	IaaS, PaaS, SaaS
Virtualization technique	KVM	Xen, Nitro	VirtualBox	PowerVM	Hyper-V
Container orchestration	GKE	EKS, Fargate	OKE	IKS	AKS
Serverless function	Cloud functions	Lambda function	Oracle functions	Cloud functions	Event grid
Pricing methodology	Pay-as-you-go	Usage-based	Flexible	Consumption-based	Pay-as-you-go
Server distribution	37 regions, 112 zones	33 regions, 99 zones	44 regions	9 regions, 27 zones	59 regions, 113 zones

Kubernetes workloads.

Oracle Cloud Infrastructure: Oracle Cloud Infrastructure offers a serverless computing system through its Oracle Cloud Infrastructure Container Engine (OKE), facilitating Kubernetes cluster deployment and management without complex infrastructure provisioning [24]. Known for enterprise-grade services, OCI handles underlying infrastructure, ensuring reliability and scalability [25]. Like other providers, OCI's serverless system features event-driven triggers, automatic scaling, and efficient resource utilization, dynamically adjusting to workload demands [24]. Users can deploy serverless functions alongside Kubernetes applications, enabling hybrid architectures [6] and simplifying development, management, and deployment processes.

IBM Cloud: To utilize Kubernetes within IBM Cloud's serverless computing framework, one must leverage the IBM Kubernetes Service (IKS) alongside Knative service, enabling the building, deployment, and management of serverless workloads [26]. IBM emphasizes streamlining the application development process, allowing developers to focus on coding rather than deployment infrastructure management [23]. Additionally, IBM Cloud offers core features for real-time event handling, including event-driven triggers from message queues, databases, or external APIs, aligning with capabilities provided by other cloud service
providers. Overall, IBM Cloud aims to simplify the development and deployment of serverless applications.

Microsoft Azure: Azure offers Azure Kubernetes Service (AKS) as part of its serverless computing solution, facilitating the creation of serverless instances and automating infrastructure management. Additionally, Azure Functions seamlessly integrates with AKS within a managed container orchestration service [27]. This setup enables the deployment and management of Kubernetes clusters alongside hybrid development architectures. Developers can leverage core serverless features such as event-driven triggers, data changes, timers, or HTTP requests [24]. Azure provides flexibility in scaling costs and customizing availability and performance according to specific requirements. Furthermore, Azure's interconnected ecosystem allows integration with other Azure services, empowering the development of robust and powerful applications [23].

2.1. Cloud providers comparison

In Table 1, we conduct a comparative analysis of cloud providers offering serverless computing, examining both their features and support for Kubernetes application containerization services. Across the five providers, we assess six key features, encompassing service offerings, virtualization techniques, container orchestration capabilities, serverless function support, pricing models, and server distribution. While our focus lies on Container as a Service (CaaS), it is noteworthy that each provider employs distinct virtualization mechanisms. For instance, GCP utilizes kernel-based virtual machines (KVM), while AWS leverages Xen and Nitro hypervisors, with Nitro developed in-house to optimize performance and security [22]. Similarly, OCI employs VM VirtualBox for virtualization, tailored to various hardware configurations, while IBM utilizes Power-VM with the Power processor [27]. Azure, on the other hand, adopts Hyper-V, a prevalent choice for x86-64 systems. In summary, GCP offers Google Kubernetes Engine (GKE) paired with Google Cloud Functions, AWS provides Elastic Kubernetes Service (EKS) alongside Fargate, OCI delivers Oracle Container Engine for Kubernetes (OKE) with Oracle Functions, IBM offers IBM Cloud Kubernetes Service (IKS) integrated with IBM Cloud Functions, and Azure furnishes Azure Kubernetes Service (AKS) complemented by Azure Event Grid [23].

Furthermore, clients often make their cloud provider selections based on the pricing methodology and server distribution offered. GCP adopts a pay-as-you-go approach, providing services across 37 regions and 112 zones [28]. AWS, on the other hand, operates on a usage-based model across 33 regions and 99 zones, enabling clients to pay for actual usage [28]. OCI offers flexibility in service selection and pricing across 44 regions, while IBM operates in 9 regions and 27 zones with a consumption-based charging model akin to AWS. Azure mirrors GCP's pay-as-you-go pricing methodology, available in 59 regions and 113 zones [28]. These cloud providers aim to support the need for hybrid development architectures through their serverless computing systems. Developers often face challenges in balancing application development and deployment due to disparate infrastructures. Therefore, serverless computing systems offer features that facilitate the development, management, and simultaneous operation of Kubernetes applications, providing much-needed flexibility.

3. Context and overview

Kubernetes has an upfront complexity, but it is beneficial for someone who has overcome that hurdle. Our primary goal is to find a cost-effective solution based on the different CPU architectures. Kubernetes is a complex architecture having the evolution of many underlying systems. Firstly, people started to deploy VMs to make the best use of their resources. Then it improved into containers. Once people started to deploy large-scale applications, the need for a container management tool arose. Kubernetes is the result of that demand. In this section, we introduce the reader to the topic of CPU architectures and how they are core to improving the performance and efficient resource utilization of large-scale applications administered by Kubernetes. Due to the distinct features and benefits of CPU architectures, such as x86 and ARM, it is important to consider how Kubernetes makes it affordable to run across all these different architectures.

3.1. CPU architecture

The central processing unit (CPU) powers computers and comprises various components such as registers, the Arithmetic Logic Unit (ALU), and the control unit [29]. These components work together to execute instructions and store data. RAM is the primary memory for instructions and data, and buses connect the CPU components [30]. Input and output devices enable external data interaction with computers, facilitating data input and output [31, 32].

x86 vs ARM: x86 and ARM processors are part of the ongoing RISC vs. CISC debate. x86 prioritizes backward compatibility and

Table 2
Feature comparison for x86 vs ARM architectures.

Features	x86	ARM
Performance	High	Energy efficient
Power consumption	High	Low
Instruction set	CISC	RISC
Compatibility	Widely compatible	Growing compatible
Cost	High	Low
Scalability	Good scalability	Scalable for embedded system

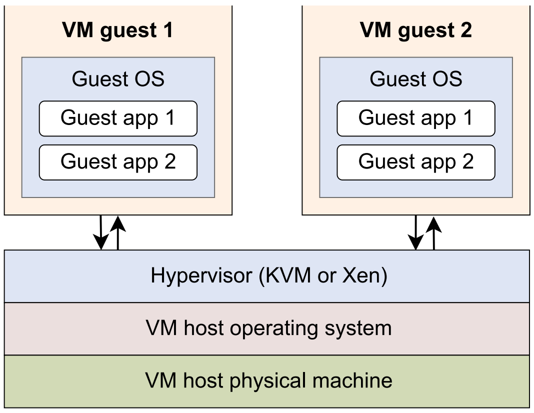

Fig. 1. Hypervisor (KVM, XEN) containerization.

has evolved from 8-bit to 16 and 32-bit instruction sets [29,30]. ARM uses the RISC architecture, offering an alternative design approach [31]. RISC enables faster execution with shorter instructions in a single clock cycle. x86 uses the CISC architecture, processing complex instructions over multiple cycles [32]. ARM consumes less power and generates less heat than x86, which focuses on performance but consumes more energy. The choice of processor depends on specific application requirements. Modern cores' power and performance depend on more than just RISC and CISC attributes, as ISAs evolve to support workload-specific semantics [15,29].

ARM Advantages: From Table 2, we can see ARM processors are highly efficient in terms of power consumption and instruction requirements, making them suitable for small-sized devices. They are particularly relevant in today's market, where there is a demand for more compact devices [33]. ARM processors have several advantages. Firstly, they are affordable to produce, making them suitable for inexpensive mobile phones and electronics. Hopkins [33] mentions that the production cost of ARM CPUs is significantly lower compared to other processors. Additionally, ARM processors are known for their speed and efficiency [33]. Besides, Jain [34] states that ARM performs a single operation at a time, benefiting from a simple instruction set, pipeline architecture, multiprocessing feature, and load-store architecture. This results in faster response times and reduced latency.

3.2. Virtualization

In the 1990s, virtualization emerged as a solution to the limitations of bare metal deployments [35]. Virtual machines (VMs) became popular, offering enhanced scalability, efficiency, and cost-effectiveness. Virtualization finds applications in various domains, including operating system virtualization, hardware-level virtualization, and server virtualization, each benefiting within its domain. As depicted in Fig. 1, the hypervisor, running on physical hardware, powers VMs by allocating resources to multiple virtual machines. These VMs, managed by the hypervisor, are termed guest systems, while the machine hosting the hypervisor is the host system. Hypervisors can be installed on the operating system or directly on hardware, as illustrated by Kernel-based Virtual Machine (KVM), integrated into the Linux kernel. Decoupling facilitates time and space-multiplexing of I/O devices, enabling the implementation of multiple logical devices using fewer physical ones.

3.3. Containerization

Containerization is a smart-to-deploy application [36]. It is like a virtual machine but without excessive overhead. In containerization, the process shares the Kernel. So there is no need to virtualize hardware and software. It has revolutionized the way we create, deploy, and test applications. Containers store bundled components of programs, including middleware and business logic (in binaries and libraries), and are self-contained and ready for deployment when running applications [37].

3.3.1. Docker

Docker is open-source, maintained by Google, and written in the Go language, which Google also maintains [38]. Each runtime is called a container. According to study [12], Docker is exceptionally efficient in deployment. Docker helps developers to build and deploy applications faster. According to study [39], Docker helps developers create cloud-native applications and isolated environments to run their code.

Docker Components: Docker encompasses vital components facilitating containerization. The Docker client–server duo orchestrates container management, accessible either locally or remotely via a RESTful API. Docker images, based on Linux OS like Debian or Alpine, form the core of containers. These images, constructed layer by layer with Dockerfile, can be tagged and stored in Docker registries such as Docker Hub, enabling image sharing. Docker containers, executed from images, serve as the execution environment for applications like web servers or databases and can be created using the "docker run" command. Docker-compose aids in managing multiple containers or those with specific configurations. Together, these elements empower efficient and scalable containerization, offering developers scalability, speed, and portability for applications. Moreover, Docker simplifies container creation, operation, and supervision, while orchestration and scaling can be achieved using tools like Kubernetes [17,36,38,39].

In the context of Kubernetes, Docker serves as the default container runtime. Kubernetes interacts with Docker to schedule containers onto nodes, monitor their health and resource usage, and perform actions such as scaling, rolling updates, and load balancing. By combining Docker with Kubernetes, organizations can leverage the benefits of both technologies to build and deploy scalable and portable applications (see Fig. 2).

3.3.2. LXC

LXC, known as Linux Containers, provides lightweight virtualization, allowing multiple isolated systems to run on a single host machine at the operating system level [40]. While functioning independently, LXC is lighter than traditional virtual machines. However, compared to Docker, LXC is not as lightweight and operates more similarly to a virtual machine, complete with its networking capabilities. Docker typically utilizes bridge networking and establishes multiple virtual networks for routing [11].

Besides, Kubernetes serves as a container orchestration platform, overseeing both Docker containers and LXC. It offers a high-level abstraction layer for managing containerized applications, simplifying the deployment of complex multi-layered applications [11]. In Fig. 3, containers communicate directly with LXC and the Kernel via an arbiter interface. By managing containerization technologies like Docker and LXC, Kubernetes becomes a potent tool capable of scaling, load-balancing, and facilitating rolling updates.

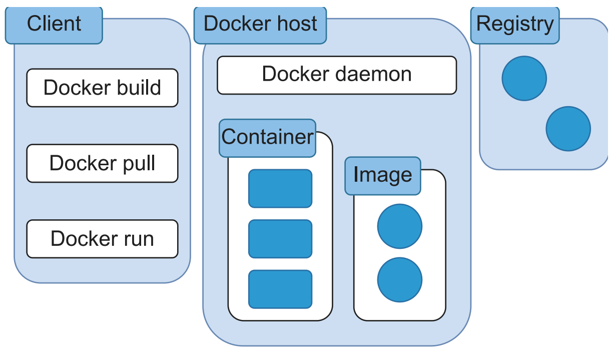

Fig. 2. Docker components.

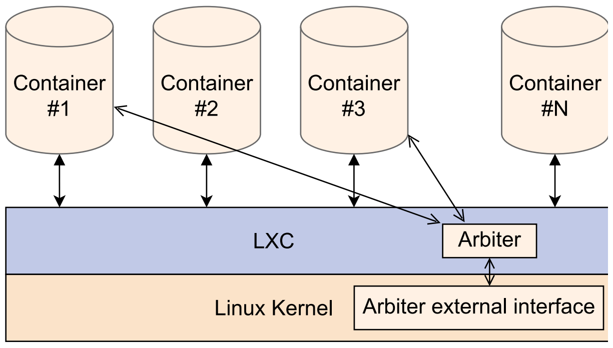

Fig. 3. LXC (Linux Container).

4. Taxonomy of Kubernetes and CPU architecture

This section provides a comprehensive overview of research studies concerning Kubernetes and CPU architecture. It covers diverse areas including performance analysis, auto scalability, various Kubernetes configurations, current CPU platforms, ARM support on major cloud providers, resource management performance, network policies, storage I/O in cloud environments, Kubernetes resource monitoring, Kubernetes deployment on Raspberry Pi, CPU power consumption, automatic throughput and critical path analysis, performance and resource management modeling, KCSS container scheduling strategy, parallelism in ARM architecture, and a comparison between x86 and ARM architectures in terms of power efficiency. These studies collectively enhance our understanding of Kubernetes' performance, scalability, and characteristics across different CPU architectures, facilitating informed decision-making and optimization for Kubernetes applications (in Table 3).

4.1. Kubernetes performance and scalability

Kubernetes Performance on Cloud Providers: According to the study by Bernstein et al. [11], Kubernetes emerges as a superior deployment system compared to most container orchestration systems. Ferreira's study [41] compared several managed Kubernetes systems, benchmarking their CPU, Memory, and I/O performance. While these benchmarks provide insights, they are synthetic, and real-world applications are typically more intricate. In practice, loads are often HTTP-based, which do not follow linear patterns. Our experiments primarily concentrated on assessing CPU performance based on HTTP performance.

Kubernetes Auto Scalability: Kubernetes excels in scalability and availability, with pods representing the smallest unit of application running instances. Pods automatically scale or restart as needed, facilitating easy adjustment of instance numbers based on demand, as highlighted in the study by Dewi et al. [42]. Additionally, Kubernetes automatically load-balances, as indicated in another study [46]. While a single-node cluster suffices for testing, production environments typically utilize multiple nodes to distribute the load equally, ensuring fair resource allocation.

4.2. Types of Kubernetes and cluster management

Different Types of Kubernetes: Different types of Kubernetes exist, each managing clusters, pods, and services uniquely, with their respective advantages and disadvantages, as outlined in the study by Telenyk et al. [14]. While creating ARM clusters is a relatively novel concept in cloud computing, it has gained popularity among the Home-lab community, with many enthusiasts setting up clusters on Raspberry Pi [47]. A study by Todorov et al. [13] describes the creation of a Raspberry Pi cluster based on K3s, a lightweight version of Kubernetes, whereas Google's vanilla Kubernetes is referred to as K8s. The Raspberry Pi cluster offers exceptional price-to-performance, although it may have limitations and lacks availability. Setting up a highly available cluster on Raspberry Pi is challenging for individuals, who must also bear the upfront hardware costs, making cloud solutions more appealing in such scenarios.

Kubernetes on Raspberry Pi: In Kubernetes, various CPU types from different architectures are commonly utilized. For example, the Vwall server runs on the x64 platform, as highlighted in the study by Goethals et al. [48], which defines the resource needs of orchestration worker nodes. Additionally, for FLEDGE resource requirements, an ARM-based architecture like the Raspberry Pi 3 can be employed [48].

4.3. Resource management and optimization in kubernetes

Resource Management Performance Characterizing: In the study by Medel et al. [43], the authors performed a performance analysis on Kubernetes using a performance model based on a Petri Net. They investigated containers' deployment and termination overheads and explored the performance of various configurations of Kubernetes pods.

Analyzing Performance and Security of Network Policies: In the study by Budigiri et al. [44], the authors assessed network policies and concluded that Kubernetes is the optimal solution for network security. They found that there is no significant performance overhead associated with Kubernetes. The study highlights the growing need for effective tenant isolation, particularly as interdependent microservices are often deployed on the same node to maximize performance.

Storage I/O in Cloud Environment: According to the study by Mercl et al. [45], cloud service providers offer various possibilities for application configuration, influencing data storage speeds. Public cloud providers like Google Cloud, Azure, and Oracle utilize different data storage solutions, resulting in varying speeds. Additionally, the study sheds light on the storage backend speeds of Microsoft Azure.

Resource Monitoring: According to the study by Mercl et al. [45], Kubernetes pods consume specific amounts of memory, CPU, and network resources. Monitoring tools like Gangila enable the tracking of power consumption, while third-party software such as PDU can also be utilized for this purpose. These tools offer valuable insights for further research. Additionally, the study suggests enhancing this information by incorporating custom-made benchmarks and collecting results accordingly.

Table 3
Summary of research papers on Kubernetes performance and scalability.

Research paper	Key characteristics	Applicability	Scalability	Cloud deployment	Hardware architecture
Bernstein et al. 2021 [11]	Assessment of Kubernetes performance on cloud providers, specifically focusing on CPU performance based on HTTP performance.	Highlight Kubernetes as a superior deployment system compared to most container orchestration systems	–	Cloud providers	ARM
Ferreira et al. 2019 [41]	Benchmarking the CPU, Memory, and I/O performance of several managed Kubernetes systems.	Understand the CPU, Memory, and I/O performance characteristics	–	Managed Kubernetes services	–
Dewi et al. 2019 [42]	Highlighting Kubernetes's scalability and availability features, with pods representing the smallest unit of application running instances.	Understand how Kubernetes can effectively handle the scaling of application instances based on demand	Pods can automatically scale or restart as needed	Cloud and on-premises deployments	–
Telenyk et al. 2021 [14]	Outline the different types of Kubernetes and their management of clusters, pods, and services.	Provides insights into Kubernetes and their unique features and evaluates their advantages and disadvantages	–	–	x86
Todorov et al. 2021 [13]	Creation of a Raspberry Pi cluster based on K3s and compares to Google's vanilla Kubernetes (referred to as K8s)	Insights into the process, advantages, and limitations of setting up a Raspberry Pi cluster based on K3s	Scalable however high availability challenges due to hardware limitation	–	Raspberry Pi and x64 platforms
Medel et al. 2018 [43]	Investigate the deployment and termination overheads of containers in Kubernetes	Insights into the deployment and termination overheads associated with containers in Kubernetes, which can help in optimizing resource allocation	–	Cloud providers	–
Budigiri et al. 2021 [44]	Highlight the absence of significant performance overhead associated with Kubernetes	Guides in making informed decisions about adopting Kubernetes for their containerized applications	–	Cloud and on-premises deployments	x86
Mercl et al. 2019 [45]	Explore the possibilities for application configuration offered by cloud service providers and their impact on data storage speeds	Individuals or organizations interested in understanding the impact of cloud service providers' configuration options on data storage speeds	–	Cloud providers	ARM

The – indicates information that was not addressed.

4.4. Advanced topics in Kubernetes and CPU architecture

CPU Power Consumption: In Kubernetes, power consumption can vary across different CPUs, as indicated by a study [49]. This variance depends on factors such as CPU architecture, processor clock speed, memory capacity, storage type, and workload characteristics. Tuning Kubernetes configurations can optimize performance while minimizing power consumption to meet specific performance requirements.

Automated Analysis of x86 and ARM Assembly Kernel Throughput: According to the study by Laukemann et al. [50], they showcased the feasibility of automatically extracting, processing, and analyzing critical paths of assembly loop kernels using their cross-platform program OSACA. They found that OSACA's conclusions are accurate, and in some cases, even more precise and flexible compared to predictions made by similar tools like IACA and LLVMMCA.

Performance and Resource Management Modeling: The study by Medel et al. [51] discusses a deployment utilizing Kubernetes, accompanied by a performance model. It highlights that the overhead linked with virtual machines (VMs) remains a challenge for applications requiring frequent instance startups and shutdowns. The authors employed a benchmark-based strategy to provide a more precise description of the behavior of a Kubernetes system utilizing Docker containers.

Kubernetes Container Scheduling Strategy (KCSS): A recent study introduced a novel Kubernetes Container Scheduling Strategy named KCSS [10]. The primary aim of KCSS is to schedule user-submitted containers with optimal efficiency. Notably, the KCSS algorithm excels in selecting the most appropriate node by striking a hybrid balance between user requirements and the current conditions of the cloud infrastructure for each submitted container.

Parallelism and ARM Instruction Set Architecture: The study [52] highlighted the significance of the ARMv6K Instruction Set Architecture (ISA) as a pivotal multiprocessor-aware instruction set. Specifically, the architecture and its implementation in the ARM11 MPCore were noted for their potential to provide low power to high-performance devices, owing to their foundation in low-power design principles. These innovative designs have the capacity to fundamentally transform the way technology is utilized.

Comparison of x86 and ARM Power Efficiency: In the study by Aroca et al. [53], they introduced a cost-effective testing method to assess the power efficiency of computers. They extensively evaluated and compared numerous ARM and x86 devices using this test setup for typical server and computing tasks such as web serving, database management, and floating-point computations. The comparison encompassed temperature, CPU usage, and power consumption of each device under varying load conditions.

Additionally, they presented Watt metrics and latency figures for serving HTTP or SQL requests, which serve as valuable indicators of system service quality. In the following section, we outline our methodology.

5. Methodology

We construct a stress-testing model aimed at assessing the performance of diverse architectures within cloud systems. Given the intricacies of stress testing, meticulous attention to multiple factors is essential. Our aim is to evaluate how each architecture performs under identical load conditions and ascertain its cost-effectiveness, thereby identifying the most favorable platform. To attain this objective, our methodology concentrates on several key facets, including:

(1) **Deployment Complexity:** We consider the complexity of deploying pods in clusters based on different architectures, which involves examining the ease of setting up and managing the infrastructure, including the necessary configurations, networking, and resource allocation for each architecture.
(2) **Benchmark Tool Selection:** We carefully evaluate and select benchmark tools for our experiments. The chosen benchmark tools are needed to provide accurate and comprehensive performance metrics, allowing us to assess various aspects of the system, such as CPU utilization, memory usage, and network throughput. The tool plays a crucial role in obtaining meaningful and reliable performance data.
(3) **Stress Testing:** We conduct stress tests on the clusters built with different architectures. These stress tests involve subjecting the systems to high workloads and intense resource demands to evaluate their performance under challenging conditions. We measure key performance indicators, such as response times, throughput, and scalability, to gain insights into how each architecture handles the stress and performs under heavy loads.
(4) **Cost Consideration:** In addition to performance, we consider the cost aspect. We compare the price-to-performance ratio of the different architectures to determine their cost-effectiveness. This analysis considers upfront costs, ongoing expenses, and each architecture's overall value. We aim to identify the architecture that best balances performance and cost (see Fig. 4).

5.1. Testing plan

Kubernetes is commonly utilized in microservices and web applications, particularly those relying on the HTTP protocol. A well-known microservice example is a REST API, which leverages HTTP methods like GET, POST, DELETE, PATCH, and PUT for different server operations [42]. In our experiments, we deploy clusters on different cloud providers that operate on distinct CPU architectures. We develop identical applications using popular frameworks and conduct comprehensive benchmarks across these deployments. These tests encompass various scenarios, allowing us to evaluate application performance across diverse platforms. To further assess the system's capabilities, we employ a synthetic benchmark called Sysbench, which enables us to measure the I/O operations of our nodes. This benchmark provides valuable insights into the system's read and write speeds, aiding in evaluating overall system performance.

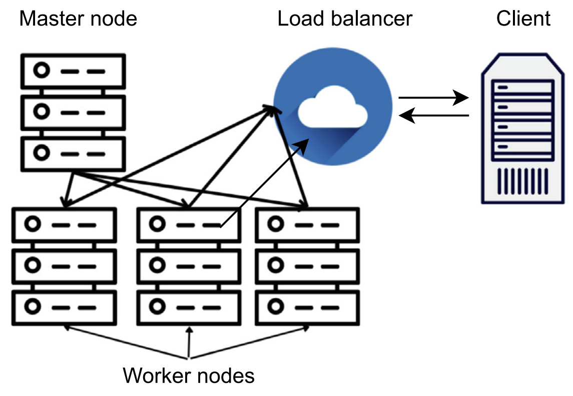

Fig. 4. Testing model.

Listing 1 Express API

```
const express = require("express");
const app = express();
app.use(express.json());
app.use("/:value", (req, res) => {
    let num = req.params.value;
    let num1 = 0;
    let num2 = 1;
    let sum = 0;
    for (let i = 0; i < num; i++) {
        sum = num1 + num2;
        num1 = num2;
        num2 = sum;
    }
    console.log(num2);
    res.status(200).json({
        status: "success",
        data: num2
    });
});
app.listen(5000);
```

5.2. Deploy clusters

We test on four platforms: Intel, AMD, Ampere, and Graviton. Intel and AMD platforms are based on x86 architecture, while Ampere and Graviton platforms run on ARM architecture. We create Kubernetes clusters on these platforms using public cloud providers. Specifically, we create GCP, AWS, and Oracle Cloud clusters for testing. We maintain uniformity across all clusters to ensure consistency in our testing environment. Each cluster has three nodes with two cores and 8 GB of RAM. Additionally, each cluster has its virtual cloud network. We deploy the clusters in different fault domain zones to enhance availability, minimizing latency while ensuring high availability. For cluster creation on the Google Cloud Platform and Oracle Cloud Infrastructure, we utilize the web GUI provided by the respective platforms. However, for AWS, we utilize the AWS-CLI tool to access our cloud resources and the EKSCTL tool to create the Kubernetes cluster using a YAML configuration file.

x86 Intel Cluster: We create an Intel x86 cluster on the Google Cloud Platform. The nodes run on Intel® Xeon® Gold 6314U

Table 4
We conduct tests on diverse cloud platforms utilizing varying CPU architectures, while ensuring uniformity in nodes, cores, RAM, and geographical regions across all trials.

Cloud provider	InstanceType	Number of nodes	CPU	Cores	RAM	Kubernetes version	Data-center location	CPU architecture
GCP	N2	3	Intel® Xeon® Gold 6314U	2	8	1.22.11-gke.400	asia-southeast1-a	x86
GCP	T2D	3	AMD Epyc 7B13	2	8	1.22.11-gke.400	asia-southeast1-a	x86
OracleCloud	VM.Standard.A1.Flex	3	Ampere A1	2	8	1.24.1	ap-singapore-1	ARM
AWS	m6 g.large	3	Neoverse N1 CPU	2	8	1.22	ap-southeast-1	ARM

processors. These nodes are GCP N2-type based instances. We create the cluster using Kubernetes version “1.22.11-gke.400”. The region of the data center is “asia-southeast1-a”.

x86 AMD Cluster: We create an AMD x86 cluster on the Google Cloud Platform. The Nodes are GCP T2D-based instances. The Kubernetes version and data center region are the same as the Intel x86 cluster. The nodes run on an AMD EPYC 7B13 processor.

ARM Ampere Cluster: We create an Ampere cluster on the Oracle Cloud. We create the cluster on Kubernetes version 1.24.1; also, nodes are based on Oracle. It is hosted on Oracle Cloud “ap-singapore-1” zone. The nodes are based on VM.Standard.A1.Flex.

ARM Graviton Cluster: We create a Graviton cluster on AWS. The Kubernetes version of the cluster is 1.22. The nodes are based on AWS m6 g.large instances. The cluster location is in the “ap-southeast-1” AWS zone. These nodes run on Neoverse N1 CPU (see Table 4).

5.3. HTTP tools

We develop our HTTP applications explicitly for benchmarking clusters, focusing on stressing the nodes’ CPU and evaluating the performance of the HTTP protocol [54]. To ensure a targeted assessment of CPU performance, we design an application that calculates the Fibonacci series [54]. The application receives the series’ last position through a GET request and responds with the result in JSON format. To maintain the integrity of CPU performance testing, we deliberately exclude external APIs, databases, and block/object storage to avoid introducing variables that could affect the results. The application complexity follows O(n), where n represents the Fibonacci series position.

To make our testing CPU-intensive and generate more realistic experimentation results, we send multiple concurrent requests to the Fibonacci application using Vegeta and our custom scripts [55]. Each request triggered a Fibonacci calculation, simulating real-world scenarios where the application handles numerous simultaneous requests. Besides, we gradually increase the number of concurrent requests over time. We start with a low load (100 requests per VM instance) and gradually scale up (2000 requests per VM instance) to simulate increasing demand for the application. Moreover, we test the Fibonacci application with a variety of input values. This includes testing with small and large Fibonacci numbers to assess how the application handles different input sizes. Then we take the average of the results for the comparison. We generate synthetic loads during our experiment to simulate higher input scenarios and evaluate system performance under such conditions. It is important to note that deploying tests across different data centers may be limited due to variations in CPU architectures among providers. This constraint influences our choice of deployment locations and affects the comparability of results.

Creating REST API: We use the REST API, which provides the best scenario for implementing Kubernetes. The last index of the Fibonacci series is delivered to the application through a GET request, with the value included in the URL [54]. The application calculates the value and sends it as an HTTP response with a status code 200 and the value in the body in JSON format. We create the same application using four powerful frameworks: Express, Flask, Gin, and Actix. These frameworks are implemented using Node.js, Python, Golang, and Rust, respectively [24]. Currently, in cloud-native development, these languages are widely popular. Node.js and Python are mature languages, while Golang and Rust have gained significant popularity for their performance and memory management, despite being relatively new in the programming world.

Listing 2 CPU Information Collection API

```
const os = require('os');
const express = require('express');
const app = express();

app.use(express.json());
app.use('/', (req, res) => {
    try {
        res.status(200).json({
            cpu: os.cpus()[0].model,
            thread: os.cpus().length,

        });
    } catch (e) {
        res.status(404).json({
            status: "failed",
        });
    }
});

app.listen(5000);
```

5.4. CPU information collection API

From the dashboard of any primary cloud provider, one can typically view the CPU’s family or generation. However, to obtain specific details about the CPU model our system utilizes, we develop an Express application. This application retrieves the system’s CPU information and sends it within the body of a JSON response.

Containerization: To run our application on a Kubernetes cluster, we convert our application into a Docker image. This process presents our first challenge when running our application on different architectures. Although the code for running on different architectures remains the same, the runtime environment for x86 architecture will not work on ARM architecture due to their different instructions. For instance, Node.js for x86 and Node.js for ARM are not similar binary files. For creating an HTTP Docker image, we use specific Docker images as a base and build upon them. Therefore, for x86 systems, we use Docker images intended for x86 systems, while for ARM systems, we use Docker images intended for ARM systems. We create a Dockerfile to generate the image to simplify the building process. We make the x86 image on an x86 system by running the Dockerfile. Similarly, we execute the Dockerfile on a Raspberry Pi-4 to create the ARM image. The images are tagged differently to facilitate easy identification.

Uploading to Image Registry: In our experimental setup, we use Docker Hub as our image registry for ensuring image accessibility across different data centers [56]. We create a Docker Hub account, obtain login credentials, and authenticate our CLI session to access the platform. With a successful login, we can upload experiment images to Docker Hub, assigning custom tags for easy retrieval. By leveraging Docker Hub, we ensure the availability and accessibility of our images across data centers, facilitating deployment and utilization within our Kubernetes clusters for practical experimentation and performance comparison between architectures.

5.5. Sysbench

We utilize the popular benchmarking tool Sysbench [31] to evaluate storage performance accurately. We establish a standardized environment for conducting storage-related tests by deploying a single pod containing the Sysbench application onto the target nodes. This approach enables us to isolate and focus solely on the storage capabilities of the underlying infrastructure, eliminating the influence of network congestion or contention with concurrent applications.

With Sysbench, we execute tests and gather performance metrics such as input/output (I/O) throughput, cost per month ($), price to performance, and success rate (%). Here, throughput is the operation count of input and output operations done in a system; cost per month is the cloud bill that is generated every month based on usage; price to performance is costing of the service that we use and the performance comparison with each other; lastly, the success rate is how many times the application operations have been successfully executed in the system [57]. These metrics provide valuable insights into the storage performance characteristics of the nodes, allowing for informed comparisons and evaluations.

Listing 3 Dockerfile

```
FROM node:latest
WORKDIR /app
COPY . .
RUN npm install
EXPOSE 5000
ENTRYPOINT ["node", "index.js"]
```

6. Experimental evaluation

In this section, we discuss the experimental evaluation, including the experimental settings, connecting with the cluster, deploying pods and services, and stress testing the system.

6.1. Experimental settings

We establish specific experimental settings and procedures to comprehensively evaluate and analyze the performance differences between the two architectures.

6.1.1. Connecting with the cluster

To access the resources of the Kubernetes cluster, we utilize the widely-used tool called “kubectl”, specifically version 1.24.0. This tool provides essential functionalities to interact with the cluster, allowing us to view information about the cluster’s nodes, pods, namespaces, and services. Additionally, we inspect logs and create interactive shells within the pods to gain deeper insights into their behavior. We rely on a configuration file called “Kube-Config” to establish a connection with the cluster, which contains the necessary authentication details. Depending on the cloud provider, we employ specific command-line tools to retrieve the Kube-Config file. For GCP clusters, we utilize the Google Cloud CLI tool, while for the AWS cluster, we rely on the AWS-CLI tool. Similarly, we use the OCI tool to access the OCP cluster, ensuring seamless connectivity to the Kubernetes cluster.

6.1.2. Deploying pods and services on the clusters

Once connected to the cluster, we deploy our application using a YAML file. We deploy 15 pods for our experiment, allocating five to each available node. This distribution ensures a balanced workload across the cluster. Kubernetes employs internal networking mechanisms to facilitate communication within the cluster. To access the HTTP application, we utilize an external load balancer that acts as a traffic distributor, evenly distributing incoming requests among the nodes. All deployments, including pods and services, are executed within the default namespace. We deploy x86 images specifically on x86 clusters, while ARM images are deployed on ARM clusters, adhering to the architecture requirements. Additionally, we deploy an application responsible for collecting CPU information specified in a YAML file [4]. To enable external access to this application, we associate it with an external load balancer that efficiently routes incoming requests.

Listing 4 Service and Deploy API

```
apiVersion: v1
kind: Service
metadata:
  name: node-test
spec:
  type: LoadBalancer
  selector:
    app: node-test
  ports:
    - port: 31111
      targetPort: 5000
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-test
spec:
  replicas: 15
  selector:
    matchLabels:
      app: node-test
  template:
    metadata:
      labels:
        app: node-test
    spec:
      containers:
        - name: k8s-nginx
          image: tanvir/node-test
          ports:
            - containerPort: 5000
```

6.1.3. Stress testing the system

We utilize a client simulator to comprehensively evaluate the system’s performance and analyze the differences between the two architectures. This simulator is designed to generate massive requests, putting the system under high-stress conditions. By simulating a heavy workload, we aim to assess the system’s performance and capabilities in demanding scenarios, enabling us

Table 5
Testing client machine configuration.

Threads	RAM	CPU	Architecture	Core	Kernel	OS
12	32 GB	Intel 5820K	x86	6	5.15	Linux Mint 21

to draw meaningful conclusions and observations. The client simulator is designed to make API calls to our application, creating a realistic and intensive load on the system. It allows us to thoroughly evaluate the system's response and performance under challenging conditions. We gather valuable insights and assess its capabilities by subjecting the system to such a workload.

Setting up Benchmarking Tool: We utilize an open-source tool called Vegeta to conduct stress testing on our cluster. We set up the Vegeta on multiple VM instances on a Linux machine running Linux Mint 21. The machine has an Intel 5820K processor with six cores and 12 threads. It has 32 GB of RAM and a 256 GB SSD. Additionally, it is connected to the cloud data centers via a 1 GBPS up-link, ensuring efficient communication during the testing process. In our stress-testing scenario, multiple VMs are instantiated within the client, each configured with Vegeta to simulate user load. Moreover, we create our custom scripts using multi-threading to simulate user load. We initiate four VMs in the single client machine based on the node capacity, resource requirements, and network overhead.

This setup allows for the parallel execution of stress tests, distributing the load across the instances to accurately assess the target system's performance under heavy traffic. Through coordinating the initiation of test scenarios across all instances simultaneously, realistic stress conditions are replicated, enabling thorough analysis of system behavior. This approach ensures comprehensive stress testing coverage and facilitates rapid feedback for performance optimization efforts (see Table 5).

Benchmarking Parameters: We perform stress tests on each cluster using Vegeta, a Golang-based tool. We conduct the tests for 30 s for each application running on the cluster. For Express, Gin, and Actix applications, we send 8000 requests per second for 30 s. However, we observed that sending more than 500 requests per second for Flask resulted in exponential degradation of results. Hence, for Flask, we limit the requests to 500 per second. To measure I/O performance, we deploy a Sysbench container in the cluster. We use a single pod for the benchmark, with the test parameters set to default. We set the Prime Number limit to 2000 and utilize a single thread for running the benchmark.

Collecting CPU Information: By utilizing the CPU information collecting API, we execute a comprehensive analysis to retrieve crucial details about the CPU. It includes gathering data on the CPU model, clock speed, and the number of cores present. Leveraging this information, we aim to gain insights into the underlying hardware specifications, facilitating further analysis and optimization of system performance.

6.1.4. Consideration of network overhead

Performing stress testing on Kubernetes clusters deployed in a public cloud can potentially have several network impacts. Stress testing involves generating a high volume of requests to simulate heavy loads on the system. This results in increased network traffic between the client generating the requests and the Kubernetes cluster, potentially saturating network links and impacting overall network performance. Moreover, the increased network traffic generated during the stress testing can introduce latency, network congestion, and interference with other workloads, causing delays in communication between components within the Kubernetes cluster and between the cluster and external services.

To overcome this network-related overhead, we review and optimize our stress testing configuration to minimize unnecessary network traffic. This includes adjusting parameters such as request rates, payload sizes, and concurrency levels to generate a realistic load without overburdening the network. We deploy the stress testing infrastructure in an isolated environment to minimize interference with other workloads sharing the same network resources. We use dedicated Kubernetes namespaces, clusters, and virtual networks to segment testing traffic from other traffic.

We set up the Kubernetes cluster in the same availability zone. The stress tester (client machine) is in Bangladesh and all the testing cluster's data centers are in Singapore. Hence, the latency may vary from 1–5 ms from one data center to another. The test is not bandwidth-dependent. Moreover, we Schedule stress tests during off-peak hours to minimize the impact on other users and workloads sharing the same network infrastructure. By conducting tests during periods of lower network activity, we try to reduce the risk of congestion and interference.

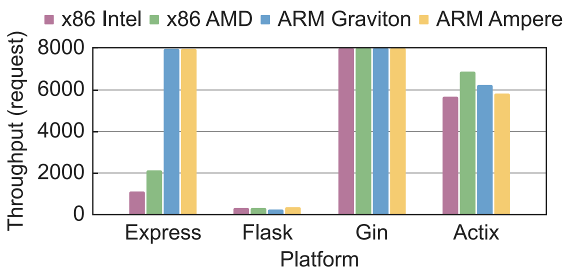

Fig. 5. API throughput using four different API.

6.2. Experimental results

After running the benchmark, we gather data on various metrics, including throughput, success rate (chances of not failing the program on the test-bed), and price-to-performance ratio. The results of the API benchmarking are presented in Table 6, while the I/O performance results are displayed in Table 7. These tables provide a comprehensive overview of the benchmarking outcomes, allowing for further analysis and comparison of the different clusters and architectures.

6.2.1. Performance results

In terms of throughput and success rate, Express API exhibits the highest performance on Ampere, followed by Graviton, Intel, and AMD. Ampere also excels in price-to-performance, significantly outpacing Graviton, while x86 platforms demonstrate comparatively lower price-to-performance ratios. Fig. 5 illustrates that Express achieves higher throughput on ARM Ampere than on ARM Graviton, with x86 AMD recording the lowest throughput at 1092.93 requests. Similarly, Fig. 6 highlights ARM Ampere's superior success rate of 100 percent, closely trailed by ARM Graviton at 99.96 percent, while x86 AMD registers the lowest success rate at 48.95 percent.

Flask demonstrates the highest throughput on Ampere, followed by Intel, AMD, and Graviton. Similarly, the success rate is highest on Ampere, with Intel, AMD, and Graviton trailing behind. Ampere also boasts the highest price-to-performance ratio, with Intel's ratio being less than half of Ampere's. AMD follows closely behind Intel, while Graviton lags with the poorest price-to-performance ratio. Fig. 5 depicts Flask's highest throughput on ARM Ampere and lowest on ARM Graviton, with Intel and AMD ranking second and third, respectively. Additionally, Flask's success rate is highest on Ampere, followed by Intel and AMD, with Graviton exhibiting the lowest performance rate at 72.24 percent.

Table 6
API benchmark result for different CPU platforms.

Platform	Metric	Express	Flask	Gin	Actix
x86 Intel	Throughput (requests / second)	2128.84	309.2	7981.29	5647.71
	Cost per month ($)	296.1	–	–	–
	Price to performance	7.189598109	1.04424181	26.95471125	19.07365755
	Success rate (%)	77.25	92.63	100	99.92
x86 AMD	Throughput (requests / second)	1092.93	305.36	7982.81	6855.55
	Cost per month ($)	314.47	–	–	–
	Price to performance	3.475466658	0.971030623	25.38496518	21.80033072
	Success rate (%)	48.95	91.6	100	99.92
ARM ampere	Throughput (requests / second)	7958.88	361.18	7989.75	5797.38
	Cost per month ($)	120.4	–	–	–
	Price to performance	66.10365449	2.999833887	66.36004983	48.15099668
	Success rate (%)	100	99.97	100	99.96
ARM graviton	Throughput (requests / second)	7956.84	243.25	7980.15	6226.32
	Cost per month ($)	243.42	–	–	–
	Price to performance	32.68770027	0.9993016186	32.78346069	25.57850629
	Success rate (%)	99.96	72.24	99.93	99.82

The - indicates the results were not obtained.

Table 7
I/O performance results based on request generation.

Platform	Architecture	Read (s)	Write (s)
GCP Intel	x86	2137.06	1424.71
GCP AMD	x86	2086.93	1391
Oracle Cloud	ARM	937.51	624.94
AWS	ARM	1807.59	1205.06

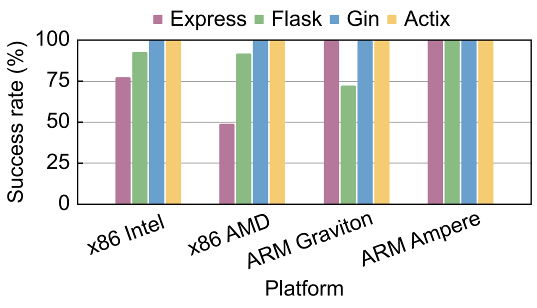

Fig. 6. API success rate (%) using four different API.

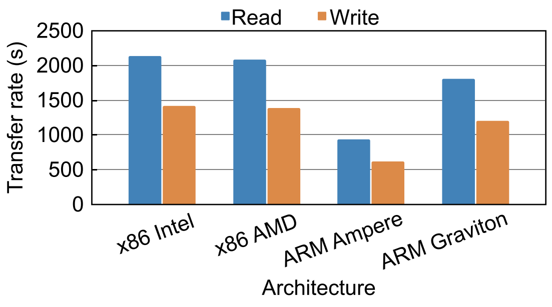

Fig. 7. I/O performance comparison.

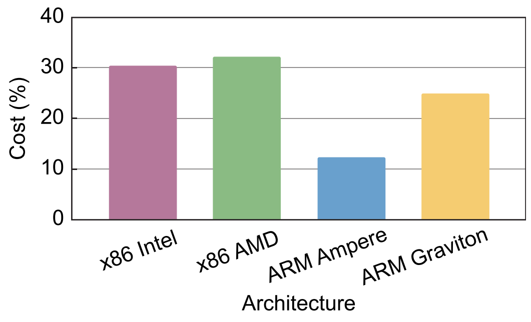

Fig. 8. Cost differences among different platforms.

The throughput of Gin and Actix remains consistent across all platforms. For ARM, Gin's price-to-performance matches that of Express, but it increases for x86 platforms. The success rate for all platforms is nearly 100 percent. Fig. 5 illustrates that Ampere boasts the highest request rate (7989.75), followed by AMD, Intel, and Graviton. In Fig. 6, Intel, AMD, and Ampere achieve a 100 percent success rate, while Graviton achieves a 99.93 percent rate. Concerning Actix, AMD achieves the highest throughput, followed by Graviton, Ampere, and Intel, respectively, as shown in Fig. 6. The success rate is nearly 100 percent for all platforms. Regarding price-to-performance, Ampere leads, followed by Graviton, AMD, and Intel. Fig. 5 displays the highest request rate on AMD for Actix throughput, with Graviton in second position and Ampere and Intel in third and fourth place, respectively. In Fig. 6, Ampere, Intel, and AMD achieve a 99.96 percent success rate, while Intel, AMD, and Graviton follow in second, third, and fourth place, respectively.

6.2.2. *I/O results*

For I/O operations, the GCP platform demonstrates better performance, with Intel and AMD taking the lead, while Graviton lags behind. Ampere's performance is the lowest in this scenario. Contrary to this finding, in Fig. 7, Ampere exhibits the lowest performance, with approximately 937 items per second for reads and 624 items per second for writes. Intel achieves the highest reading performance at approximately 2137 items per second, with writing at approximately 1424 items per second. Additionally, AMD and Graviton rank second and third, respectively, in terms of performance.

6.3. *Experimental findings*

The key findings of our study include stress testing performance, cost-effectiveness analysis, and comparison of cloud providers and CPU architectures.

- ARM consistently outperforms x86 architectures, with Express showcasing superior performance.
- Graviton and Ampere exhibit comparable performance, with Ampere standing out for its superior cost-effectiveness. Ampere was consistently the most cost-effective option, outperforming Graviton, Intel, and AMD.

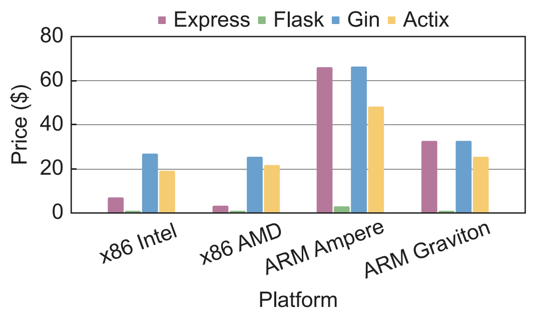

Fig. 9. Price to performance ratio.

- Fig. 8 illustrates Ampere as the most efficient platform, costing only $120.4, while AMD has the highest cost at approximately $314.47.
- Ampere surpasses Graviton, Intel, and AMD in terms of price-to-performance ratio, with Ampere achieving the highest performance ratio in Flask.
- Similar trends are observed in Gin and Actix, where Ampere consistently demonstrates the highest performance, followed by Graviton and AMD, while Intel trails behind.
- The GCP platform demonstrates better performance for I/O operations, with Intel and AMD leading the pack, while Graviton lags behind.
- Contrary to the initial finding, Ampere exhibits the lowest performance for I/O operations, with significantly lower read and write speeds compared to Intel and AMD.

These findings underscore the superior performance of ARM Ampere compared to other architectures, as detailed in Fig. 9.

7. Discussion and open challenges

In our study, we present real-world results obtained from evaluating various Kubernetes services offered by different cloud providers across diverse architectures. We conducted tests on x86 platforms using both Intel and AMD, as well as ARM platforms utilizing Ampere and Graviton. Notably, RISC-based architectures like IBM Power and Oracle SPARC are viable for Kubernetes clusters due to their optimized computing instruction sets, as indicated by previous studies [58,59]. However, setting up performance tests on these architectures may require complex configurations. Moreover, enterprise-level ARM-based architectures, such as Fujitsu ARM, Cavium ThunderX, and Qualcomm Centriq, offer promising options for performance monitoring. Our evaluation involved frameworks such as Express, Gin, Flask, and Actix, where we developed APIs and subjected the system to stress testing using a client simulator, leading to our conclusive findings.

7.1. Backend performance variations analysis

The variations in performance observed during stress testing of the Fibonacci number calculation application across different platforms are attributed to a combination of several backend factors. Below, we provide key reasons for the observed performance patterns:

7.1.1. ARM vs. x86 performance

ARM-based architectures (e.g., Ampere, AWS Graviton) consistently outperformed x86-based architectures (e.g., Intel, AMD) in several test scenarios. This discrepancy can be attributed to several architectural differences:

- **Instruction Set Design**: ARM processors, particularly in server-grade designs like Graviton, use a Reduced Instruction Set Computing (RISC) architecture. RISC architectures execute fewer and simpler instructions per cycle, leading to reduced instruction-level overhead compared to the Complex Instruction Set Computing (CISC) design in x86 processors. This makes ARM architectures better suited for highly parallelizable tasks, like Fibonacci number calculations, where computational load can be distributed across multiple cores with less complexity.
- **Efficiency of Express on ARM**: Express frameworks perform better on ARM likely due to the efficient handling of asynchronous operations. ARM processors, with their ability to handle multi-threaded applications efficiently, benefit from the non-blocking, event-driven nature of Node.js-based applications like Express. ARM's power-efficient multi-core design aligns well with the parallelism in these web applications, leading to better throughput and lower latency.
- **Core Density and Scalability**: ARM-based systems typically offer a higher number of cores with better energy efficiency. This allows for better scaling under multi-threaded workloads, like Fibonacci computations, which benefit from high levels of concurrency. In contrast, x86 cores may be more powerful individually but have higher energy consumption and scaling inefficiencies at higher thread counts.

7.1.2. GCP vs. AWS and I/O operations performance

The GCP platform demonstrated superior performance for I/O-intensive operations, particularly on Intel and AMD platforms, while Graviton lagged behind. Several factors explain this:

- **I/O Scheduler Differences**: x86 platforms, especially Intel and AMD architectures, utilize advanced I/O schedulers optimized for handling heavy disk and network I/O loads. These schedulers are tuned for high throughput and low latency, with features like more aggressive prefetching, read-ahead, and better queue management. In contrast, ARM-based platforms, such as Graviton, may use simpler I/O scheduling mechanisms due to a focus on compute-heavy operations over I/O-bound ones.
- **Different I/O Patterns**: The workloads tested on GCP may have involved complex I/O patterns (e.g., frequent disk reads/writes or network operations) that favor x86 architectures. Intel and AMD processors benefit from optimized I/O subsystems, higher memory bandwidth, and lower I/O latency, making them more suitable for I/O-heavy tasks. ARM-based processors may not yet have the same level of optimization in I/O handling due to their traditionally compute-focused design.

7.2. Theoretical comparative study

Table 8 provides an overview of our approach, which incorporates the use of synthetic and custom-built benchmark tools for evaluating Kubernetes performance. Additionally, our experimentation takes advantage of a high-capacity cluster within a public cloud environment, facilitating rigorous testing across both x86 and ARM architectures. In contrast, prior studies often lacked the utilization of such benchmarking tools, relied on private cloud services, and neglected to assess performance across multiple architectures. This meticulous testing methodology enhances the credibility and reliability of our results, offering a more comprehensive understanding of Kubernetes performance under diverse conditions. Ultimately, our findings are poised to offer valuable insights to the developer community, enabling them to make informed decisions and optimize their Kubernetes deployments in real-world scenarios.

Table 8
Comparison of different literature.

Study	Benchmark		Cloud		Highly available cluster	Platform	
	Synthetic	Custom built	Public	Private		x86	ARM
Our study	✓	✓	✓	–	✓	✓	✓
Blieninger et al. 2022 [60]	✓	✓	✓	–	–	–	✓
Telenyk et al. 2021 [14]	✓	–	✓	–	✓	✓	–
Todorov et al. 2021 [13]	–	–	–	✓	–	–	✓
Ferreira et al. 2019 [41]	✓	–	✓	–	✓	✓	–
Muddinagiri et al. 2019 [61]	–	–	–	✓	–	✓	–
Pahl et al. 2016 [62]	–	✓	–	✓	✓	✓	–

The ✓ indicates that the approach is coincident.
The – indicates that the approach is not coincident.

Table 9
Web framework performance on cloud platforms [63–65].

Metrics	Cloud platforms	Express.js (Node.js)	Flask (Python)	Gin (Golang)	Actix (Rust)
Execution speed	GCP	9 ms	7 ms	9 ms	9 ms
	AWS	7 ms	9 ms	7 ms	9 ms
	Oracle	5 ms	5 ms	5 ms	5 ms
	Azure	7 ms	7 ms	7 ms	5 ms
	IBM	9 ms	9 ms	9 ms	9 ms
Memory consumption	GCP	7 MB	3 MB	3 MB	3 MB
	AWS	3 MB	7 MB	3 MB	5 MB
	Oracle	3 MB	3 MB	3 MB	3 MB
	Azure	3 MB	3 MB	3 MB	5 MB
	IBM	5 MB	5 MB	5 MB	5 MB
CPU utilization	GCP	70%	80%	90%	95%
	AWS	75%	85%	80%	90%
	Oracle	60%	70%	65%	75%
	Azure	80%	90%	85%	75%
	IBM	75%	85%	80%	90%
Latency	GCP	10 ms	15 ms	20 ms	25 ms
	AWS	15 ms	20 ms	18 ms	22 ms
	Oracle	25 ms	30 ms	28 ms	35 ms
	Azure	12 ms	18 ms	16 ms	20 ms
	IBM	20 ms	25 ms	22 ms	28 ms
Throughput	GCP	1000 req/s	800 req/s	900 req/s	1000 req/s
	AWS	800 req/s	900 req/s	700 req/s	1000 req/s
	Oracle	600 req/s	700 req/s	500 req/s	800 req/s
	Azure	900 req/s	1000 req/s	800 req/s	900 req/s
	IBM	1000 req/s	1100 req/s	1000 req/s	1200 req/s
Scalability	GCP	High	Medium	High	High
	AWS	Medium	High	Medium	High
	Oracle	Low	Low	Low	Low
	Azure	Medium	Medium	Medium	Low
	IBM	High	High	High	High
Concurrency support	GCP	High	Medium	High	High
	AWS	Medium	High	Medium	High
	Oracle	Low	Low	Low	Low
	Azure	Medium	Medium	Medium	Low
	IBM	High	High	High	High

7.3. Impact of programming language on Kubernetes performance

The programming language chosen for cloud-based applications, including those running on Kubernetes, can significantly impact their performance and overall effectiveness [66]. In our study, we evaluated the performance of a Kubernetes application using four different web frameworks: Express.js (Node.js), Flask (Django), Gin (Golang), and Actix (Rust). While Kubernetes primarily manages containerized applications, the programming language choice can affect resource utilization, scalability, latency, and throughput, all of which are critical for application performance on Kubernetes [64]. Commonly considered language performance metrics include response time, throughput, latency, memory consumption, CPU utilization, and scalability under various load conditions. Insights from existing literature and surveys on language performance in cloud environments provide valuable guidance for understanding how different languages perform on Kubernetes.

In Table 9, we present the performance of web frameworks on different cloud platforms based on existing studies [63–65]. Express.js (Node.js), Flask (Python), Gin (Golang), and Actix (Rust) demonstrate similar execution speeds, ranging from 5 ms to 9 ms across platforms. Flask consistently shows the lowest memory consumption, ranging from 3 MB to 7 MB. Actix exhibits higher CPU utilization (90% to 95%) compared to Flask (60% to 85%). Express.js and Flask generally have lower latency values compared to Gin and Actix, indicating faster response times. Express.js, Gin, and Actix demonstrate high throughput (800 req/s to 1200 req/s), while Flask exhibits slightly lower throughput (500 req/s to 1000 req/s). Actix, Express.js, and IBM show high scalability, while Flask and Gin exhibit medium scalability. Actix, Express.js, and IBM also demonstrate high concurrency support, whereas Flask and Gin exhibit medium support. Oracle generally shows lower scalability and concurrency support for all frameworks.

In a nutshell, Express.js, Flask, Gin, and Actix perform well in terms of execution speed, with Flask showing the lowest memory consumption. Actix has higher CPU utilization, while Express.js and Flask generally have lower latency. Express.js, Gin, and Actix exhibit high throughput, scalability, and concurrency support. By considering these factors and examining relevant literature, we can make an informed decision about the programming language that best aligns with requirements and optimizes the performance of applications running on Kubernetes or other cloud platforms.

7.4. Challenges and limitations

Our study is subject to certain limitations primarily due to the exclusive focus on enterprise-level and costly architectures. Local and cloud-based architectures were tested based on resource availability and requirements, leading to some constraints. Specifically, the inability to examine CPUs beyond x86 Intel, x86 AMD, ARM Ampere, and ARM Graviton due to limited cloud service resources poses a limitation. Moreover, the evaluation was confined to cloud service providers such as GCP, AWS, and Oracle, neglecting other providers that may offer different virtualization technologies, storage speeds, and instances. Consequently, the performance of identical applications may vary across different cloud providers utilizing similar hardware infrastructures.

7.5. Future work

In forthcoming research endeavors, we intend to enhance our benchmarking efforts by conducting more diverse assessments to pinpoint the specific strengths and weaknesses of each architecture. Our focus remains on performance testing across three prominent cloud platforms: Google Cloud Platform, Oracle Cloud Infrastructure, and Amazon Web Services. Additionally, we acknowledge the importance of considering other platforms such as Digital Ocean (DO), Alibaba Cloud, Microsoft Azure, and IBM Blue-mix, which will broaden the scope of our investigations. Furthermore, we aim to incorporate multi-machine setups in our future work to monitor multi-client performance and conduct more comprehensive analyses.

8. Conclusion

Our study was motivated by the need to assess the performance and cost implications of ARM CPUs compared to x86 architectures in Kubernetes deployments. Given the increasing interest in ARM processors and their potential benefits for cloud computing, we conducted extensive benchmarking tests to evaluate their suitability as a cost-effective solution. Our results consistently demonstrated that ARM CPUs outperformed x86 architectures across various scenarios, owing to their advanced instruction sets, lower cost, and greater scalability. Despite the widespread use of x86 in the industry, the affordability and energy efficiency of ARM processors position them as promising alternatives for organizations seeking efficient Kubernetes infrastructure.

CRediT authorship contribution statement

Jannatun Noor: Project administration, Supervision, Writing – review & editing. **MD Badsha Faysal:** Conceptualization, Methodology, Project administration. **MD Sheikh Amin:** Investigation, Resources, Validation. **Bushra Tabassum:** Data curation, Investigation, Visualization. **Tamim Raiyan Khan:** Investigation, Writing – original draft. **Tanvir Rahman:** Conceptualization, Investigation, Methodology, Resources, Validation, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix. Kubernetes

Kubernetes, an influential open-source technology within the serverless domain, orchestrates containers effectively. It comprises various components collaborating to manage containerized applications, with a Kubernetes cluster serving as its cornerstone [17,67]. These clusters can vary from simple setups for testing to complex configurations with multiple nodes, consisting of master and worker nodes. The master nodes execute critical Kubernetes programs like the API server, facilitating cluster communication, while the controller manager oversees maintenance tasks, and the scheduler optimizes container allocation based on resource availability and workload needs [41,67]. Worker nodes host application containers, coordinated by master nodes, while Etcd, a key–value storage system, maintains the cluster's real-time state [67]. Inter-component communication occurs via a virtual network, fostering seamless node interaction. As the de facto industry standard for container orchestration, Kubernetes significantly influences the adoption of serverless architecture [8].

A.1. Control plane components

The control plane components of Kubernetes oversee resource management and cluster health, ensuring seamless functionality [8] (in Fig. 10). They monitor cluster events, promptly addressing issues like failed pod execution. While any machine in the cluster can host these components, best practices advise against deploying user containers on the same machines. In the subsequent sections, we delve into the distinct roles and functionalities of Kubernetes' control plane components.

Kube-API Server: The Kube-API server serves as the central communication hub in the Kubernetes cluster, orchestrating interactions between nodes and controllers [17]. Responsible for managing connections and facilitating information flow, it plays a pivotal role in scaling the cluster by coordinating with newly added nodes. Additionally, the API server issues commands to nodes, monitors their performance, and efficiently distributes I/O loads across the cluster.

etcd: etcd functions as the database within the Kubernetes cluster, housing crucial configuration data such as authentication keys and facilitating coordination among distributed systems [68]. Offering features like simple key–value storage, data consistency across multiple cluster nodes, real-time functionality, atomic transactions, and role-based access control, etcd serves as the primary data repository for distributed systems like Kubernetes.

Kube-Scheduler: Kube-Scheduler oversees the regular scheduling tasks within the Kubernetes cluster, ensuring the smooth execution of tasks on a routine basis [69]. With its plug-and-play architecture, Kube-Scheduler can accommodate various scheduling algorithms. While it comes with a default algorithm, which is relatively straightforward, it is also customizable, allowing for the management of increasingly complex scheduling scenarios with ease.

Kube-Controller Manager: The Kube-Controller Manager is responsible for overseeing various control loops within the Kubernetes cluster, ensuring the continuous operation of essential functionalities [17]. These control loops are comprised of different controllers, each serving a specific purpose. Firstly, the Node Controllers are tasked with identifying node outages and initiating appropriate actions to maintain cluster stability. Secondly, the Job Controller monitors Job objects representing sporadic tasks and orchestrates the creation of Pods to execute these tasks efficiently. Additionally, the Endpoints Controller facilitates the integration of Services and Pods by updating objects within the Endpoints object. Lastly, the Service Account and Token Controllers play a crucial role in generating API access tokens and

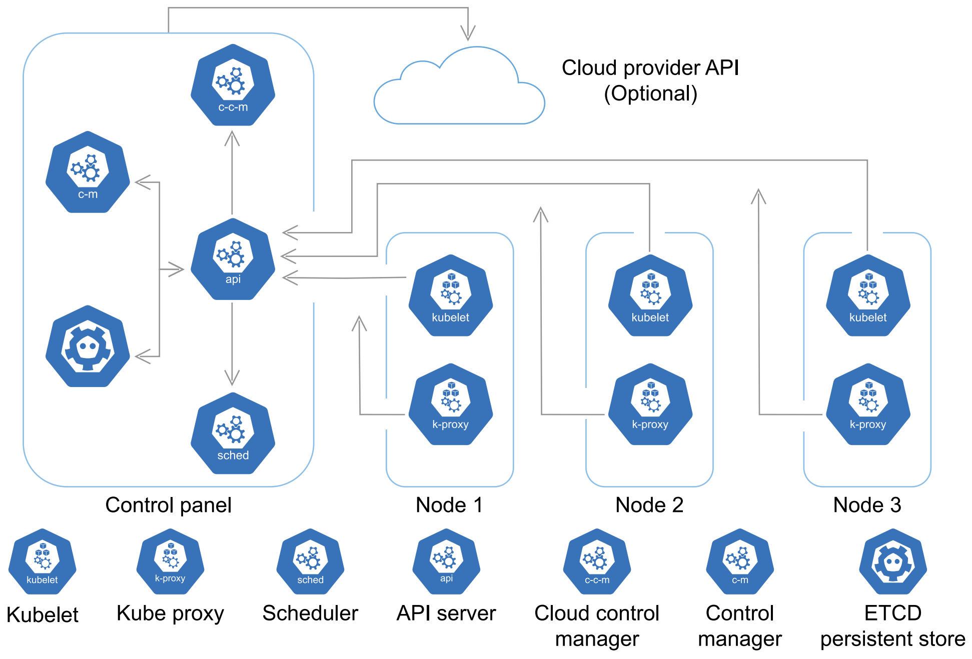

Fig. 10. Kubernetes cluster.

default accounts for new namespaces, enhancing security and access management within the cluster.

Cloud-Controller Manager: Kubernetes is predominantly deployed on cloud environments, where cloud providers integrate their applications into the cluster to manage its operations and offer services on demand [8]. However, these implementations vary across different cloud providers. The Node Controller is responsible for monitoring and managing the status of nodes within the cluster, taking appropriate actions in response to node failures or terminations. The Route Controller controls internal routing, ensuring efficient traffic direction for network communication within the cluster. Lastly, the Service Controller oversees Kubernetes services, ensuring their proper configuration and maintenance to enable seamless connectivity between various components of the cluster.

A.2. Node components

Each node in a Kubernetes cluster hosts services that communicate with the master node, receiving tasks and monitoring pod resource usage [69]. If issues arise, they report back to the master node for resolution.

Kube-proxy: Kubernetes networking is intricate, involving communication among various services, making accessing applications on default ports challenging. Kube-Proxy simplifies this by configuring firewall protocols and network rules [23].

Container Run-time: Kubernetes supports container runtime other than Docker. Some of them contain CRI-O. We can choose which container runtime we want for the application [23].

Services: If we want to access the application running on Kubernetes, some essential services are Node Port, Load-balancer, Cluster IP, and Ingress [8]. To use an FQDN on an application, we use Ingress.

References

[1] J.M. Parra-Ullauri, H. Madhukumar, A.-C. Nicolaescu, X. Zhang, A. Bravalheri, R. Hussain, X. Vasilakos, R. Nejabati, D. Simeonidou, kubeFlower: A privacy-preserving framework for Kubernetes-based federated learning in cloud–edge environments, Future Gener. Comput. Syst. 157 (2024) 558–572, http://dx.doi.org/10.1016/j.future.2024.03.041.

[2] J. Noor, S.I. Salim, A.A.A. Islam, Strategizing secured image storing and efficient image retrieval through a new cloud framework, J. Netw. Comput. Appl. 192 (2021) 103167, http://dx.doi.org/10.1016/j.jnca.2021.103167.

[3] J. Noor, M.N.H. Shanto, J.J. Mondal, M.G. Hossain, S. Chellappan, A.B.M.A. Al Islam, Orchestrating image retrieval and storage over a cloud system, IEEE Trans. Cloud Comput. 11 (2) (2023) 1794–1806, http://dx.doi.org/10.1109/tcc.2022.3162790, URL http://dx.doi.org/10.1109/TCC.2022.3162790.

[4] J. Noor, H.I. Akbar, R.A. Sujon, A.A. Al Islam, Secure processing-aware media storage (SPMS), in: 2017 IEEE 36th International Performance Computing and Communications Conference, IPCCC, IEEE, 2017, http://dx.doi.org/10.1109/pccc.2017.8280457.

[5] H. Liang, Z. Zhang, C. Hu, Y. Gong, D. Cheng, A survey on spatio-temporal big data analytics ecosystem: Resource management, processing platform, and applications, IEEE Trans. Big Data 10 (2) (2024) 174–193, http://dx.doi.org/10.1109/tbdata.2023.3342619.

[6] S. Nasrin, T.F. Sahryer, A.B.M.A.A. Islam, J. Noor, Feature and performance based comparative study on serverless frameworks, in: 2021 24th International Conference on Computer and Information Technology (ICCIT), IEEE, 2021, pp. 1–6, http://dx.doi.org/10.1109/iccit54785.2021.9689779.

[7] Pavithra, Google cloud interview questions, 2022, URL https://laraonlinetraining.com/google-cloud-interview-questions/.

[8] C. Carrión, Kubernetes as a standard container orchestrator - A bibliometric analysis, J. Grid Comput. 20 (4) (2022) 42, http://dx.doi.org/10.1007/s10723-022-09629-8.

[9] Pods | Official Documentation of Kubernetes, 2023, URL https://kubernetes.io/docs/concepts/workloads/pods.

[10] T. Menouer, KCSS: Kubernetes container scheduling strategy, J. Supercomput. 77 (5) (2020) 4267–4293, http://dx.doi.org/10.1007/s11227-020-03427-3.

[11] D. Bernstein, Containers and cloud: From LXC to Docker to Kubernetes, IEEE Cloud Computing 1 (3) (2014) 81–84, http://dx.doi.org/10.1109/mcc.2014.51.

[12] C. Boettiger, An introduction to docker for reproducible research, ACM SIGOPS Oper. Syst. Rev. 49 (1) (2015) 71–79, http://dx.doi.org/10.1145/2723872.2723882.

[13] M.H. Todorov, Design and deployment of Kubernetes Cluster on Raspberry Pi OS, in: 2021 29th National Conference with International Participation (TELECOM), IEEE, 2021, pp. 104–107, http://dx.doi.org/10.1109/telecom53156.2021.9659651.

[14] S. Telenyk, O. Sopov, E. Zharikov, G. Nowakowski, A comparison of Kubernetes and Kubernetes-compatible platforms, in: 2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 1, IEEE, 2021, pp. 313–317, http://dx.doi.org/10.1109/idaacs53288.2021.9660392.

[15] K. Gupta, T. Sharma, Changing trends in computer architecture : A comprehensive analysis of ARM and x86 processors, Int. J. Sci. Res. Comput. Sci Eng. Inf. Technol. (2021) 619–631, http://dx.doi.org/10.32628/cseit2173188.

[16] H. Shafiei, A. Khonsari, P. Mousavi, Serverless computing: A survey of opportunities, challenges, and applications, ACM Comput. Surv. 54 (11s) (2022) 1–32, http://dx.doi.org/10.1145/3510611.
[17] J. Shah, D. Dubaria, Building modern clouds: using docker, kubernetes & Google cloud platform, in: 2019 IEEE 9th Annual Computing and Communication Workshop and Conference, CCWC, IEEE, 2019, pp. 0184–0189, http://dx.doi.org/10.1109/ccwc.2019.8666479.
[18] V. Tamizhkumaran, MULTI-cloud: What is it and what are the ways to make best use of it? 2022, URL https://mobiritz.com/technology/multi-cloud-what-is-it-and-what-are-the-ways-to-make-best-use-of-it/.
[19] M.A. Kamal, H.W. Raza, M.M. Alam, M.M. Su'ud, Highlight the features of AWS, GCP and Microsoft Azure that have an impact when choosing a cloud service provider, Int. J. Recent Technol. Eng. (IJRTE) 8 (5) (2020) 4124–4132, http://dx.doi.org/10.35940/ijrte.d8573.018520.
[20] M. Deb, A. Choudhury, Hybrid cloud: A new paradigm in cloud computing, Mach. Learn. Tech. Anal. Cloud Secur. (2021) 1–23, http://dx.doi.org/10.1002/9781119764113.ch1.
[21] M. Kyryk, N. Pleskanka, M. Pleskanka, V. Kyryk, Infrastructure as code and microservices for intent-based cloud networking, in: Future Intent-Based Networking: On the QoS Robust and Energy Efficient Heterogeneous Software Defined Networks, Springer, 2021, pp. 51–68, http://dx.doi.org/10.1007/978-3-030-92435-5_4.
[22] K. Djemame, Serverless computing: Introduction and research challenges, in: International Conference on the Economics of Grids, Clouds, Systems, and Services, Springer Nature, 2022, pp. 15–23.
[23] S.K. Mondal, R. Pan, H.D. Kabir, T. Tian, H.-N. Dai, Kubernetes in IT administration and serverless computing: An empirical study and research challenges, J. Supercomput. 78 (2) (2022) 1–51, http://dx.doi.org/10.1007/s11227-021-03982-3.
[24] A. Png, H. Helskyaho, Exposing functionality with API gateway, in: Extending Oracle Application Express with Oracle Cloud Features: A Guide To Enhancing APEX Web Applications with Cloud-Native and Machine Learning Technologies, Springer, 2022, pp. 111–138, http://dx.doi.org/10.1007/978-1-4842-8170-3_4.
[25] A. Engelsrud, Managing PeopleSoft on the Oracle Cloud, A Press, 2019, http://dx.doi.org/10.1007/978-1-4842-4546-0.
[26] P.G. López, A. Arjona, J. Sampé, A. Slominski, L. Villard, Triggerflow: trigger-based orchestration of serverless workflows, in: Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems, 2020, pp. 3–14, http://dx.doi.org/10.1145/3401025.3401731.
[27] P. Raj, S. Vanga, A. Chaudhary, Setting up a Kubernetes Cluster using Azure Kubernetes Service, Wiley-IEEE Press (2023) http://dx.doi.org/10.1002/9781119814795.ch11.
[28] T. Melissaris, K. Nabar, R. Radut, S. Rehmtulla, A. Shi, S. Chandrashekar, I. Papapanagiotou, Elastic cloud services: scaling snowflake's control plane, in: Proceedings of the 13th Symposium on Cloud Computing, ACM, 2022, pp. 142–157, http://dx.doi.org/10.1145/3542929.3563483.
[29] E. Blem, J. Menon, K. Sankaralingam, A detailed analysis of contemporary arm and x86 architectures, 2013, URL https://minds.wisconsin.edu/handle/1793/64923.
[30] D.C. Schuurman, Step-by-step design and simulation of a simple CPU architecture, in: Proceeding of the 44th ACM Technical Symposium on Computer Science Education, 2013, pp. 335–340, http://dx.doi.org/10.1145/2445196.2445296.
[31] J. Phillips, Simulation of a simple CPU design and its use as an instructional tool in a computer organization course, J. Comput. Sci. Coll. 22 (6) (2007) 140–146.
[32] A. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, B. Esbaugh, Parallelism via multithreaded and multicore CPUs, Computer 43 (3) (2010) 24–32, http://dx.doi.org/10.1109/mc.2010.75.
[33] J. Hopkins, What is an ARM processor? comparison to x86 and its advantages and disadvantages, 2022, URL https://www.totalphase.com/blog/2022/03/what-is-arm-processor-comparison-x86-and-advantages-disadvantages.
[34] S. Jain, Advantages and disadvantages of ARM processor, 2020, URL https://www.geeksforgeeks.org/advantages-and-disadvantages-of-arm-processor.
[35] K. tej Koganti, E. Patnala, S.S. Narasingu, J. Chaitanya, Virtualization technology in cloud computing environment, Int. J. Emerg. Technol. Adv. Eng. 3 (3) (2013).
[36] J. Turnbull, The Docker Book: Containerization is the new virtualization, James Turnbull, 2014.
[37] C. Pahl, Containerization and the PaaS Cloud, IEEE Cloud Comput. 2 (3) (2015) 24–31, http://dx.doi.org/10.1109/mcc.2015.51.
[38] D. Reis, B. Piedade, F.F. Correia, J.P. Dias, A. Aguiar, Developing docker and docker-compose specifications: A developers' survey, IEEE Access 10 (2021) 2318–2329, http://dx.doi.org/10.1109/access.2021.3137671.
[39] T. Bui, Analysis of Docker Security, 2015, http://dx.doi.org/10.48550/ARXIV.1501.02967.
[40] B. Russell, Passive benchmarking with docker LXC, 2014, URL https://www.slideshare.net/BodenRussell/kvm-and-docker-lxc-benchmarking-with-openstack.
[41] A.P. Ferreira, R. Sinnott, A performance evaluation of containers running on managed kubernetes services, in: 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2019, pp. 199–208, http://dx.doi.org/10.1109/cloudcom.2019.00038.
[42] L.P. Dewi, A. Noertjahyana, H.N. Palit, K. Yedutun, Server scalability using kubernetes, in: 2019 4th Technology Innovation Management and Engineering Science International Conference (TIMES-ICON), IEEE, 2019, pp. 1–4, http://dx.doi.org/10.1109/times-icon47539.2019.9024501.
[43] V. Medel, R. Tolosana-Calasanz, J.Á. Bañares, U. Arronategui, O.F. Rana, Characterising resource management performance in Kubernetes, Comput. Electr. Eng. 68 (2018) 286–297, http://dx.doi.org/10.1016/j.compeleceng.2018.03.041.
[44] G. Budigiri, C. Baumann, J.T. Muhlberg, E. Truyen, W. Joosen, Network policies in Kubernetes: Performance evaluation and security analysis, in: 2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit), IEEE, 2021, pp. 407–412, http://dx.doi.org/10.1109/eucnc/6gsummit51104.2021.9482526.
[45] L. Mercl, J. Pavlik, Public cloud Kubernetes storage performance analysis, in: Computational Collective Intelligence, Springer International Publishing, 2019, pp. 649–660, http://dx.doi.org/10.1007/978-3-030-28374-2_56.
[46] Z. He, Novel container cloud elastic scaling strategy based on Kubernetes, in: 2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC), IEEE, 2020, pp. 1400–1404, http://dx.doi.org/10.1109/itoec49072.2020.9141552.
[47] S. Kenlon, 5 reasons to run Kubernetes on your Raspberry Pi homelab, 2020, URL https://opensource.com/article/20/8/kubernetes-raspberry-pi, (Accessed on 29 May 2023).
[48] T. Goethals, F.D. Turck, B. Volckaert, Extending kubernetes clusters to low-resource edge devices using virtual kubelets, IEEE Trans. Cloud Comput. 10 (4) (2022) 2623–2636, http://dx.doi.org/10.1109/tcc.2020.3033807.
[49] E. Kristiani, C.-T. Yang, C.-Y. Huang, Y.-T. Wang, P.-C. Ko, The implementation of a cloud-edge computing architecture using OpenStack and kubernetes for air quality monitoring application, Mob. Netw. Appl. 26 (3) (2020) 1070–1092, http://dx.doi.org/10.1007/s11036-020-01620-5.
[50] J. Laukemann, J. Hammer, G. Hager, G. Wellein, Automatic throughput and critical path analysis of x86 and ARM assembly kernels, in: 2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), IEEE, 2019, pp. 1–6, http://dx.doi.org/10.1109/pmbs49563.2019.00006.
[51] V. Medel, O. Rana, J.Á. Bañares, U. Arronategui, Modelling performance & resource management in kubernetes, in: Proceedings of the 9th International Conference on Utility and Cloud Computing, 2016, pp. 257–262,
[52] J. Goodacre, A.N. Sloss, Parallelism and the ARM instruction set architecture, Computer 38 (7) (2005) 42–50, http://dx.doi.org/10.1109/mc.2005.239.
[53] R.V. Aroca, L.M.G. Gonçalves, Towards green data centers: A comparison of x86 and ARM architectures power efficiency, J. Parallel Distrib. Comput. 72 (12) (2012) 1770–1780, http://dx.doi.org/10.1016/j.jpdc.2012.08.005.
[54] Y. Bouizem, N. Parlavantzas, D. Dib, C. Morin, Active-standby for high-availability in faas, in: Proceedings of the 2020 Sixth International Workshop on Serverless Computing, ACM, 2020, pp. 31–36, http://dx.doi.org/10.1145/3429880.3430097.
[55] P. Czarnul, Benchmarking performance of a hybrid intel xeon/xeon phi system for parallel computation of similarity measures between large vectors, Int. J. Parallel Program. 45 (5) (2016) 1091–1107, http://dx.doi.org/10.1007/s10766-016-0455-0.
[56] Docker Overview, 2022, URL https://docs.docker.com/get-started/overview/.
[57] N. Kratzke, Cloud computing costs and benefits: An IT management point of view, in: Service Science: Research and Innovations in the Service Economy, Springer New York, 2012, pp. 185–203, http://dx.doi.org/10.1007/978-1-4614-2326-3_10.
[58] B. Linzel, E. Zhu, G. Flores, J. Liu, S. Dikaleh, How can OpenShift accelerate your kubernetes adoption: A workshop exploring OpenShift features, in: Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON '19, IBM Corp., USA, 2019, pp. 380–381.
[59] H. Fathoni, C.-T. Yang, C.-H. Chang, C.-Y. Huang, Performance comparison of lightweight kubernetes in edge devices, in: Pervasive Systems, Algorithms and Networks, Springer International Publishing, 2019, pp. 304–309, http://dx.doi.org/10.1007/978-3-030-30143-9_25.
[60] B. Blieninger, A. Dietz, U. Baumgarten, Mark8s-A management approach for automotive real-time kubernetes containers in the mobile edge cloud, RAGE 2022 (2022) 10.
[61] R. Muddinagiri, S. Ambavane, S. Bayas, Self-hosted Kubernetes: deploying Docker containers locally with Minikube, in: 2019 International Conference on Innovative Trends and Advances in Engineering and Technology, ICITAET, IEEE, 2019, pp. 239–243, http://dx.doi.org/10.1109/icitaet47105.2019.9170208.

[62] C. Pahl, P. Jamshidi, Microservices: A systematic mapping study, in: Proceedings of the 6th International Conference on Cloud Computing and Services Science, SCITEPRESS - Science and Technology Publications, 2016, pp. 137–149, http://dx.doi.org/10.5220/0005785501370146.

[63] G.G. Magalhaes, A.L. Sartor, A.F. Lorenzon, P.O.A. Navaux, A.C. Schneider Beck, How programming languages and paradigms affect performance and energy in multithreaded applications, in: 2016 VI Brazilian Symposium on Computing Systems Engineering, SBESC, IEEE, 2016, http://dx.doi.org/10.1109/sbesc.2016.019.

[64] S. Taherizadeh, M. Grobelnik, Key influencing factors of the kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications, Adv. Eng. Softw. 140 (2020) 102734, http://dx.doi.org/10.1016/j.advengsoft.2019.102734.

[65] P. Jogalekar, M. Woodside, Evaluating the scalability of distributed systems, IEEE Trans. Parallel Distrib. Syst. 11 (6) (2000) 589–603, http://dx.doi.org/10.1109/71.862209, URL http://dx.doi.org/10.1109/71.862209.

[66] T. Tanadechopon, B. Kasemsontitum, Performance evaluation of programming languages as API services for cloud environments: A comparative study of PHP, python, node.js and golang, in: 2023 7th International Conference on Information Technology (InCIT), IEEE, 2023, http://dx.doi.org/10.1109/incit60207.2023.10413079.

[67] G. Sayfan, Mastering Kubernetes, Packt Publishing, Birmingham, England, 2023.

[68] L. Larsson, W. Tärneberg, C. Klein, E. Elmroth, M. Kihl, Impact of etcd deployment on Kubernetes, Istio, and application performance, Softw. - Pract. Exp. 50 (10) (2020) 1986–2007, http://dx.doi.org/10.1002/spe.2885.

[69] D. Vohra, S.O. Service, Kubernetes Management Design Patterns : With Docker, CoreOS Linux, and Other Platforms, A Press, Berkeley, Ca, 2017.