

# **Containerization and Cloud Computing: A Practical Analysis of Docker and Cloud Service Models**

**Program: MSc Computing Portfolio**

**Module: Cloud Engineering Principles and Practice**

**Module Code: ACCA7027**

**Lecturer: Renuga Jayakumar**

**Name: Sohanur Rahman**

**Student ID: 2434329**

## Table of Contents

|                                         |    |
|-----------------------------------------|----|
| 1. INTRODUCTION.....                    | 2  |
| 2. DOCKER AND CONTAINERIZATION          | 2  |
| 2.1 Virtualisation Basics.....          | 2  |
| 2.2 What is Containerization .....      | 2  |
| 2.3 Building Docker Images .....        | 3  |
| 2.4 Running Containers .....            | 4  |
| 2.5 Publishing to DockerHub.....        | 5  |
| 2.6 Docker Volumes.....                 | 6  |
| 2.7 Docker Networks.....                | 6  |
| 3. CLOUD COMPUTING CONCEPTS..           | 7  |
| 3.1 Load Balancing .....                | 7  |
| 3.2 Scalability .....                   | 8  |
| 3.3 Application Gateways.....           | 8  |
| 3.4 How They Work Together .....        | 9  |
| 4. SERVERLESS VS CONTAINERS.....        | 9  |
| 4.1 Why Cloud Computing is Growing      | 9  |
| 4.2 What is Serverless .....            | 10 |
| 4.3 Comparing Serverless and Containers | 10 |
| 5. CONCLUSION.....                      | 11 |
| 6. REFERENCES .....                     | 1  |

# 1. INTRODUCTION

The current cloud computing has transformed the way applications are rolled out and managed today. One of the primary technologies that has enabled this to be possible is containerization, since it enables applications to be run in varying environments. This report not only considers Docker as a containerization tool, but it also discusses some key concepts in cloud computing, such as load balancing and scalability. The report has been divided into three major sections. First, there is a practical demonstration of Docker that demonstrates how to create containers, manage storage using volumes and set up networking. Second, the conceptual analysis of cloud computing with emphasis on the way cloud computing providers such as AWS and Azure manage load balancing and scaling. Third, the comparison between serverless computing and traditional containers in order to learn when to apply each one. The assignment work experience illustrates the simplistic process of containerization in generating an image to publishing the image on the internet. The presented examples are not complicated, but they demonstrate the principles of the general ones used in the more complex world.

## 2. DOCKER AND CONTAINERIZATION

### 2.1 Virtualisation Basics

Virtualisation fundamentally refers to the context of developing virtualised forms of computer resources in place of direct dependence on the actual hardware. This is achieved in the traditional virtualisation via a hypervisor, which is a layer through which more than one virtual machine (VM) is capable of executing on a single physical machine. VMs have their full operating systems and act to a great extent like single computers [35]. This design is effective; however, it consumes enormous proportions of system resources since each VM has to have its own OS and assigned memory, which rapidly accumulates when multiple VMs are operating simultaneously. Isolation is among the largest benefits of virtualisation. Each VM is an independent entity, and hence issues experienced in one VM, such as a crash or a software failure, do not impact the rest. This gives the system increased security. Portability is also another benefit: because all that the application requires is packaged within the VM, it becomes significantly easier to move the VM between systems in different servers or even different environments without any concern of compatibility problems. There are drawbacks to these advantages. VMs consume a lot of resources since they take a lot of memory and storage to operate several complete operating systems. Starting a VM may take a lot of time, even a few minutes, which is not acceptable in situations where a scale of applications is required [34]. The process of having multiple operating systems running on a single machine also restricts the number of applications that can actually be implemented successfully. Here, containerization comes to the rescue as a less heavy variant. Containers instead of virtualising the hardware, as VMs do, virtualise the operating system. This implies that many containers share the same OS kernel but are isolated to run. Due to this shared-kernel architecture, containers are significantly smaller, boot in less time, and consume fewer resources than VMs. Naturally, there are tradeoffs associated with containers [33]. They do not offer the same isolation as virtual machines since they rely on the host kernel. But in the context of most recent applications, particularly web applications and microservices, this degree of isolation is typically more than adequate. In practice, containers are more frequently selected due to the speed, efficiency, and scalability advantages, which make the performance more beneficial in numerous situations.

### 2.2 What is Containerization

Containerization bundles an application and all the resources it requires to execute, such as the code, runtime, libraries, and configuration files, into a single unit. It is a solution to the extremely widespread problem in which an application performs well in development but not in the real world because the system configuration differs between the two machines. The

environment is now predictable and consistent, no matter where it is deployed, by putting all in a single container. The most contemporary popular containerization platform is Docker [32]. It provides all the tools to create container images, execute the container images as containers, and share the container images with other individuals or systems. Docker's major components are the Docker daemon, which manages the operations of containers in the background, the Docker CLI, which developers use to run their commands, container images that serve as blueprints and registries such as DockerHub, where images may be stored and accessed. The Containers operate based on the Linux kernel capabilities. Namespaces isolate processes, where one container is not able to access the processes and resources of another container. Control groups (cgroups) are used to control and restrict the amount of CPU, memory, and other resources a given container can use [31]. A layered filesystem is also employed by Docker, but it enables containers to share base layers and leave their own changes distinct. This reduces the size of images and their efficiency. In terms of development, containers are very beneficial as they bring a single environment throughout the workflow. The developers, testers, CI/CD pipelines and production servers all operate on the same container image, so unnecessary bugs that arise due to changes in the environment are eliminated. This also aids in simplifying debugging since the behaviour is seen by all. The virtual machines are very lightweight compared to containers, which is why a single server can support a much larger number of workloads. They are relatively quick to start (typically within a matter of seconds), which is helpful when an application must scale quickly, such as when user traffic suddenly increases. Containerised systems deal with security with a number of layers [30]. Although each container uses the same host operating system kernel, it is isolated by namespaces, among other security measures. The best security steps are to ensure that containers are launched with minimum privileges, the images are periodically vulnerable and updated to prevent known security threats. The combination of these steps is useful in sustaining a secure and reliable container environment.

### **2.3 Building Docker Images**

The fundamental unit in the process of containerization is Docker images. They are built based on a Dockerfile, which is nothing but a plain text file containing step-by-step instructions on how the image is to be built. Every command in a Dockerfile, like FROM, COPY or RUN, installs a new layer in the resulting image. These layers are built upon each other to give the ultimate environment the container will execute. Such a layered system has multiple benefits for Docker. The layers are stored independently; thus, the same base layers can be reused by different images, hence saving much disk space [29]. Docker also supports layer caching, which implies that when rebuilding an image, only the changed layers are rebuilt. The layers that are not changed are taken out of the cache, and hence the build process is much faster. The other advantage is that the individual layers can be checked or examined in case of an error, and it becomes easier to troubleshoot. To conduct a practical demonstration, a basic web server was developed with the help of nginx. The base image in the Dockerfile was nginx: alpine. This base image has a lightweight Linux, and nginx can be installed. Because it is small and efficient on a scale of the tiny Alpine Linux, container images are often built using Alpine since it is very small and efficient, which consequently enables the reduction of image size and faster deployments. The COPY command in the Docker file is the command which adds files to the image [28]. In this case, it copied the HTML file to the nginx web root directory to where the server can display the webpage. This is the way the application code and other resources are embedded in the final image. There is an added EXPOSE instruction that defines to what port the container will listen, and it is very clear how the application is to be accessed. The Docker build command is used to build the image [27]. In the build, Docker first verifies that the base image is already on the system; otherwise, it gets it out of the registry. It then executes each

Dockerfile instruction sequentially, making intermediate containers and recording changes as new layers.

```
C:\Users\sarka\Desktop\cloud-assignment\web-server>docker build -t my-webserver:v1
[*] Building 13.4s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> transferring dockerfile: 119B
=> [internal] load metadata for docker.io/library/nginx:alpine
=> [auth] library/nginx:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> transferring context: 2B
=> [internal] load build context
=> transferring context: 22B
=> [1/2] FROM docker.io/library/nginx:alpine@sha256:289decab414258121a93c3f1b8316b9c69966de3a8
=> resolve docker.io/library/nginx:alpine@sha256:289decab414258121a93c3f1b8316b9c69966de3a8
=> sha256:289decab414258121a93c3f1b8316b9c69966de3a8993757c42c4c964169ad4 10.33kB / 10.33kB
=> sha256:e4116b539937cebbf267d4f26af97b79b4b0bca381367891cf88841abfeda 2.58kB / 2.58kB
=> sha256:5d2c3a0126d3d91208b4b7b771ecf9766e32e155506f5cca568b63b1dce9 639B / 639B
=> sha256:a23f9a9d96277f4ebf2ba07801ccf6e16b3a01f93c2d5216b0d2b1b5 10.99kB / 10.99kB
=> sha256:014e56e613968f73cc0888120ca5fbc601d788899969a8ee59f31dc071a53 3.86MB / 3.86MB
=> sha256:dfad29ba5c259f8d1ec1938529f8ef602e335a2668047ad56d38e9727e1a10a 1.86MB / 1.86MB
=> sha256:abdec946283a31e986f184559f417a33c3a8936a801532cf9fa208a44ad08 954B / 954B
=> extracting sha256:014e56e613968f73cc0888120ca5fbc601d788899969a8ee59f31dc071a53
=> sha256:51c38d93937c33b0b5668daed09996f558d88877b05a5e18551a0a05e1f 483B / 483B
=> extracting sha256:dfad29ba5c259f8d1ec1938529f8ef602e335a2668047ad56d38e9727e1a10a
=> sha256:ad5b5da02cfbd43daa87043b07851f3816a10eb7719938d8cb9a96ee828d471 1.21kB / 1.21kB
=> sha256:fc1353283672b70e7dd76ae52f2743b14326883c31935c86d7d77c68019dea 1.40kB / 1.40kB
=> extracting sha256:5d2c3a0126d3d91208b4b7b771ecf9766e32e155506f5cca568b63b1dce9
=> extracting sha256:abdec946283a31e986f184559f417a33c3a8936a801532cf9fa208a44ad08
=> extracting sha256:51c38d93937c33b0b5668daed09996f558d88877b05a5e18551a0a05e1f
=> sha256:136bc6976c2023e336e6688167d08019fccc3e756c162c5875a3819b40b6 17.26MB / 17.26MB
=> extracting sha256:ad5b5da02cfbd43daa87043b07851f3816a10eb7719938d8cb9a96ee828d471
=> extracting sha256:fc1353283672b70e7dd76ae52f2743b14326883c31935c86d7d77c68019dea
=> extracting sha256:136bc6976c2023e336e6688167d08019fccc3e756c162c5875a3819b40b6
=> [2/2] COPY index.html /usr/share/nginx/html/index.html
=> exporting to image
=> exporting layers
=> writing image sha256:cbef4f0b2abb88f207edc83394d3489b7ebec42b4da779e11ecbe51755ca
=> naming to docker.io/library/my-webserver:v1
```

Figure 1: Docker image build process using Dockerfile

The caching mechanism used in Docker is relevant. To give a case in point, when the HTML file is changed and the image is rebuilt, Docker does not rebuild the whole build. It uses the laid-out cache of everything up to the stage at which a change took place, and then only reinvents the layers after the COPY command [26]. This is very fast whenever there are frequent changes. Lastly, images may be given meaningful names and version numbers and managing the various releases of an application becomes very easy.

## 2.4 Running Containers

After creating an image, it is possible to create containers with the usage of Docker run command. This command uses the picture and creates a new container instance on the basis of the picture. A container is, in many ways, a lightweight, isolated application environment to start. There are a number of flags available with docker run that allow you to influence the behavior of the container such as setting up port mappings, restricting resource usage such as CPU or memory, defining restart policies, and the type of networking the container uses. All these options enable the container to be flexible to be used in various scenarios, both developmental and production.

```
C:\Users\sarka\Desktop\cloud-assignment\web-server>docker run -d -p 8080:80 --name my-container -v my-
data-volume:/usr/share/nginx/html/data my-webserver:v1
a8ee7800e2807c74c45a6a34b1b9c2805b463311eab5f908e2ef40894a66e90f
```

Figure 2: Docker container deployment command

The reason why port mapping is significant is the fact that containers have their own in-house network and are not connected to the host by default. The -p flag binds a host port with a container port, such as 8080:80 allows one to access an nginx container on localhost:8080. The -d flag executes the container in detached mode, hence it will still be in the background [25]. After the container has been started, the web application can be accessed using the port on which it has been mapped in the same way as any local service is accessed. Although it is an isolated environment, Docker allows an external user or clients to communicate with it with ease. This demonstrates one of the largest advantages of Docker: the same container can be deployed across any machine or cloud platform with consistency, and, as a consequence, deployment, testing, and scaling are made much easier.



## 2.6 Docker Volumes

By default, when a container is deleted, all the data inside it gets deleted too. This is fine for stateless applications, but problematic for things like databases that need to keep data permanently. Docker volumes solve this problem by providing persistent storage that exists outside the container. Volumes are managed by Docker and stored in a specific location on the host machine [23]. They're separate from the container's filesystem, which means they survive even if the container is removed. Multiple containers can also share the same volume if needed. Creating a volume is straightforward with the Docker volume create command. Named volumes are easier to work with than anonymous volumes because they can be referenced by name. The Docker volume inspect command shows details about the volume, including where it's actually stored on the host system.

```
C:\Users\sarka\Desktop\cloud-assignment\web-server>docker volume inspect my-data-volume
[
  {
    "CreatedAt": "2025-12-18T09:43:47Z",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/my-data-volume/_data",
    "Name": "my-data-volume",
    "Options": null,
    "Scope": "local"
  }
]
```

*Figure 6: Creating and inspecting a Docker volume for persistent data storage*

Volumes get attached to containers when they're created using the -v flag in docker run. From inside the container, the volume just looks like a normal directory. Applications can read and write to it without knowing it's actually a volume. Common use cases include database storage where files need to persist across container restarts and log files that need to be collected and analysed [22].

## 2.7 Docker Networks

Container networking handles communication between containers and with the outside world. Docker provides several networking options depending on what's needed. The default bridge network lets containers on the same host communicate with each other while isolating them from external networks. Custom bridge networks add features like automatic DNS resolution, where containers can find each other by name instead of IP address [21]. The Docker network create command makes a new bridge network with its own subnet. Docker automatically manages IP addresses for containers connected to the network.

```

C:\Users\sarka\Desktop\cloud-assignment\web-server>docker network create my-app-network
8ef97daca835889b8572fe52ae0362217dfae826af8af7351a653e7cb240b0dc

C:\Users\sarka\Desktop\cloud-assignment\web-server>docker network inspect my-app-network
[
  {
    "Name": "my-app-network",
    "Id": "8ef97daca835889b8572fe52ae0362217dfae826af8af7351a653e7cb240b0dc",
    "Created": "2025-12-18T09:59:43.329687892Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv4": true,
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {},
    "Options": {
      "com.docker.network.enable_ipv4": "true",
      "com.docker.network.enable_ipv6": "false"
    },
    "Labels": {}
  }
]

```

*Figure 7: Creating and configuring a Docker network for container communication*

Custom Docker networks are helpful in microservices setups where different services need to communicate with each other. Each service runs in its own container but shares the same network, and sometimes a frontend container may connect to multiple networks for example, one for backend communication and another for external traffic [20]. Port publishing allows containers to be accessed from outside the host machine. Using the `-p` flag maps a host port to a container port. For example, `-p 8080:80` lets you reach the container's port 80 by visiting `localhost:8080` on the host.

**Youtube Video Link:** [https://www.youtube.com/watch?v=jSnt6aXm\\_sg](https://www.youtube.com/watch?v=jSnt6aXm_sg)

## 3. CLOUD COMPUTING CONCEPTS

### 3.1 Load Balancing

Load balancing is a technique used to distribute incoming traffic evenly across multiple servers so that no single server becomes overloaded. This helps improve overall performance because requests are spread out, and it also increases reliability—if one server goes down, the remaining servers can continue to handle the traffic. In the case of no load balancing, a rush in users will slow down or crash the single server and the whole application[19]. The modern load balancers operate on various levels of the network model. The layer 4 load balancers base their routing decisions on such information as IP addresses and ports. They are very quick and efficient and do not examine the content of the request. In contrast, layer 7 load balancers also examine HTTP traffic and can route requests using URLs, hostnames, HTTP headers, cookies and any other data used as application boundaries. This can be used to do smarter routing, like make particular API calls to particular backend services. Elastic load balancing (ELB) is available in three types offered by AWS. Application Load Balancers (ALB) can be used with modern web applications and microservices since they utilise Layer 7 and are capable of supporting more sophisticated routing rules [18]. NLBs are used to achieve extremely high-performance workloads working at Layer 4 and can support millions of requests per second with very low

latency. Classic Load Balancers (CLB) are older and provide basic load balancing at the transport and application layer. Similar load-balancing tools are available in Microsoft Azure. Azure load balancer operates at Layer 4, and it is applied in high-throughput and low-latency applications. Azure Application Gateway operates at Layer 7 and offers such features as URL-based routing and Web Application Firewall (WAF). Both of them support both public load balancing of internet-facing applications and internal load balancing of applications that communicate inside a virtual network only. Health checks are one of the main characteristics of load balancers. These checks keep on checking the backend servers by submitting small test requests to them to determine that they are functioning correctly. When one of the servers fails the health checks, the load balancer will automatically route traffic out of it until it comes back online. This automatic failover is very much better in enhancing availability compared to manual intervention. SSL termination is another concept to note[17]. This is the time the load balancer does the encryption and decryption of the HTTPS traffic, and not the backend servers.

### **3.2 Scalability**

Scalability is the characteristic of a system to cope with the growing workload with the addition of new resources. Scaling can be done in two ways [16]. Vertical scaling involves adding more capacity to a single server, such as adding additional CPU, RAM, or storage. This is simple but physically restricted; you can only upgrade a machine so much. Horizontal scaling, conversely, means processing the load by adding additional servers. The workload is divided among the various small machines as opposed to having one large machine. Horizontal scaling can suit cloud computing much better since the cloud environment provides an opportunity to add or remove servers when there is a demand [15]. It implies that organisations do not have to purchase costly hardware to be able to absorb peak traffic on rare occasions. Rather, they can spin up additional instances temporarily during peak demand and spin them down temporarily during low traffic flows. This strategy not only enhances performance but also makes sure that the expenditure corresponds to the actual utilisation. Auto-scaling extends this idea beyond the limits and automates the process. Auto-scaling uses such indicators as CPU load, network traffic, or the amount of incoming traffic. New instances are launched automatically when these metrics exceed a certain threshold [14]. In case of a drop in demand, the system stops the redundant cases to prevent resource and money wastage. AWS Auto Scaling is compatible with EC2 instances, container workloads, among other AWS services. Users can create elaborate scaling policies which entail what metrics to monitor and how the system is to respond. Predictive scaling is also available in AWS, and it relies on machine learning to examine the past trends of traffic and size the system in advance of a spike taking place. Azure Autoscale is also available to both app services and virtual machines. The users will be able to come up with rules like the addition of two other instances when the CPU usage remains above 75% for over five minutes. Azure has the support of schedule-based scaling, which is suitable for applications whose usage patterns are predictable, such as increased usage during business hours and reduced usage during the night [13]. The architecture of the application is also crucial in scaling the application. Stateless applications do not store the data locally back on the server, hence scale extremely well as any instance can process any request. Stateful applications need to use external storage systems like databases, caches or session stores to store state when there are many instances of the application running.

### **3.3 Application Gateways**

Application gateways are enhanced versions of load balancers with a special design in that they can handle web traffic. Application gateways are unlike simple load balancers that can only route based on IP address and ports, but they base their routing decision on the contents of the HTTP request [12]. This consists of the URLs, headers, hostnames and cookies. Due to this more precise examination, they can do smarter and more dynamic routing of new web applications. Azure Application Gateway has a number of handy options. Among them is URL-

based routing, whereby various requests, such as /api and /images, may be redirected to entirely diverse back-end servers. This proves useful particularly in microservice arrangements. Azure also provides multi-site hosting, i.e. multiple sites with different domains may be served using the same gateway. It even offers automatic redirection, including redirecting all HTTP traffic to HTTPS, which can be used to ensure security and a stable user experience. One of the biggest advantages of the Application Gateway is that it can be integrated with a Web Application Firewall (WAF). The WAF scans incoming traffic against the pattern of different attacks known to it, such as SQL queries, cross-site scripting or malicious bots [11]. It blocks the request even before it ever communicates with the backend servers when it finds something suspicious. This significantly minimises the number of areas of attack and introduces an additional layer of defence against web applications. SSL offloading is another important characteristic. The gateway performs the work of encrypting and decrypting the HTTPS traffic instead of the workload being carried out by all backend servers. This not only enhances the opening of backend servers, but also makes the handling of certificates a little easier, as one only needs to install the certificate on the gateway.

### **3.4 How They Work Together**

Load balancing and auto-scaling are two concepts that go hand in hand since they address different but complementary issues. Load balancers dwell on the equal distribution of the traffic received on more than one server, whereas auto-scaling changes the number of servers in accordance with the demand in real time [10]. They form systems that can automatically react to such alterations in the workload without the need to be manually controlled. This is of particular interest to applications whose traffic spikes are unpredictable. In the case of auto-scaling, the instances are automatically registered to the load balancer to ensure that they start receiving traffic immediately. In the same way, as the auto-scaling group determines the instances should be terminated during low usage, the load balancer will stop transmitting traffic to these instances before they are terminated. All that occurs is behind the scenes. Health checks that are part of the load balancer are also a feedback loop, which allows the auto-scaling system to make more reasonable choices about which instances to keep running [9]. This combination results in the effective and clever usage of resources. Auto-scaling will also lower the instance count during idle times, and the load balancer will just spread the load among the rest. As the traffic grows, new instances are added and immediately added to the load-balancing pool. What is produced is a system that is easily scaled up when the need arises and easily scaled down to conserve on costs. These technologies used together offer several important advantages: high availability due to redundancy and failover, better performance due to balanced traffic distribution, enhanced security when combined with application gateways and WAFs, and high cost optimisation due to dynamic scaling. All in all, they are integrated to ensure the sustainability of contemporary web applications in a dynamic environment to preserve high reliability, responsiveness, and cost-efficiency.

## **4. SERVERLESS VS CONTAINERS**

### **4.1 Why Cloud Computing is Growing**

Cloud adoption is motivated by a number of factors. One is cost, which is huge in that an organisation does not need to spend a lot of money to purchase hardware, but pays as they use it. This transforms the capital expenditure into the operational expenditure that is usually easier to manage. Elasticity enables organisations to increase and decrease resources depending on necessity [8]. Conventional infrastructure necessitates primarily idle capacity planning. The resources within the clouds can be dynamically scaled, and only the amount that is required is paid for. The deployment rate has increased tremendously. Traditional infrastructure could be provisioned in weeks or months. APIs and automation can ensure that cloud resources are ready in a few minutes. This expedited provisioning allows hastened time to market new products.

Cloud providers with global data centres have easier access to a global reach. Applications can be installed in various locations without incurring costs to construct infrastructure in every location. Managed services eliminate the infrastructure maintenance overhead. Organisations can concentrate on their core business and not on the management of the infrastructure [7]. Moreover, cloud usage increases resilience and reliability by providing automated backup, built-in redundancy, and disaster recovery features, which are typically expensive to deploy in-premise. High-level security features such as encryption, identity and access management services, and continuous monitoring are also becoming the norm, assisting organisations to address the compliance needs. Cloud computing can also facilitate innovation because it allows access to new technology, including artificial intelligence, machine learning, big data analytics, and Internet of Things products, without significant initial investment[6]. In addition, teamwork and distance work are enhanced, because the systems based in the cloud can be safely accessed remotely, to facilitate the contemporary, remote and distributed workplaces and business continuity.

## **4.2 What is Serverless**

Serverless computing refers to the execution of code without server management, yet there exist servers in the background. Provisioning, scaling, and maintenance are managed by the cloud provider. Function-as-a-Service is the most widespread model, and developers deploy small functions which execute on receiving an event, such as an HTTP request, database update, or file upload [5]. One of the top serverless systems is AWS Lambda. Code is uploaded by developers, triggers are set, and Lambda itself can automatically scale between a small number of requests to thousands of requests per second. Azure functions provide the same functionality, are multilingual, and can be easily connected to other services in Azure. The price is determined on a per-execution basis or a higher performance tier. Serverless functions are also stateless, meaning that with every execution, each function begins with no data and no data is persisted on the device. Any ongoing state should be stored in storage facilities such as databases. Moreover, serverless computing makes the operations less technical by removing infrastructure issues like patching, capacity planning, and fault tolerance [4]. It is suitable to implement microservices-based and event-driven applications by supporting the fast development and deployment cycle. Serverless services go hand in hand with cloud-native applications such as message queues, API gateways, and monitoring applications, allowing workflows to scale with little configuration required. There are issues, including cold start latency, lock-in by vendors and execution time and memory limits. Nevertheless, these limitations do not stop the proliferation of serverless usage considering its effectiveness, economies of scale with intermittent workloads, and compatibility with agile and DevOps.

## **4.3 Comparing Serverless and Containers**

Container and serverless are two latest strategies to deploy an application, and despite being frequently compared, they complement each other. Each has its advantages, and it is important to know when to employ either of the two to make better architectural choices. One of the largest ones is the operational model. In the case of serverless, developers do not control any infrastructure whatsoever [3] They just code and deploy, and the cloud provider takes care of all the coding in the background, such as servers, scaling and maintenance. Containers, on the other hand, need a greater operational awareness. Developers need to build Docker images, establish dependencies, and frequently control orchestration systems, such as Kubernetes, ECS, or other container systems. Containers are not less set up than serverless functions, even with managed services[2]. The two models are different when it comes to scaling. Serverless functions grow automatically to tens of thousands of executions with no configuration and are best suited to unexpected or unforeseen spikes. Containers may also scale, but they need rules that auto-scale and typically maintain at least some instances running, and hence never entirely cease operation. The cost model also varies. Serverless only costs the time of real execution,

which is highly effective when the workloads are not scheduled regularly. Containers are expensive when they are active, even when idle, hence are useful in long-running, consistent applications. Another difference is in performance. The initial request is slow due to cold starts, which Serverless can experience, whereas containers maintain constant and low-latency responses[1]. Use cases differ as well. Serverless is suitable for event-driven applications, bursty API traffic, scheduled jobs, and data processing. Long-running services, multi-component systems and applications that require full control of their runtime environment are fitted in containers. On the whole, both technologies are expected to enhance cloud efficiency, agility and cost minimisation, but in various manners.

## **5. CONCLUSION**

This report has discussed the concept of containerization using Docker using real-life examples as well as examined the major concepts of cloud computing, such as load balancing, scaling and serverless computing. The Docker example demonstrated the process of making images based on Dockerfiles, starting containers, using persistent storage in the form of volumes, and networking. The release of the image to DockerHub illustrated the ability to distribute and deploy the containers and deploy them in various environments in the same manner. Such practical assignments support the theory and give hands-on experience with the basic cloud technologies. The cloud computing analysis demonstrated the collaboration of load balancing, scalability and application gateways to offer highly available and performant services. These concepts are used through the advanced services of AWS and Azure, which have similar concepts. The analysis of serverless and containers showed that they are complementary strategies and not competitive. Serverless is a good fit for event-driven variable workloads that are automatically scaled. Containers are more effective in more complex applications, sustained workloads, and those applications that require predictable performance. The work experience and theoretical work provide be informed of the basic cloud engineering concepts and technologies.

## 6. REFERENCES

- [1] Yadav, P.S., 2024. Optimizing Serverless Architectures for Ultra-Low Latency in Financial Applications. *European Journal of Advances in Engineering and Technology*, 11(3), pp.146-157.
- [2] Castellanos-Rodríguez, Ó., Expósito, R.R., Enes, J., Taboada, G.L. and Touriño, J., 2024. Serverless-like platform for container-based YARN clusters. *Future Generation Computer Systems*, 155, pp.256-271.
- [3] Safeer, C.M., 2023. *Architecting Cloud-Native Serverless Solutions: Design, build, and operate serverless solutions on cloud and open source platforms*. Packt Publishing Ltd.
- [4] Cherukuri, B.R., 2024. Serverless computing: How to build and deploy applications without managing infrastructure. *World Journal of Advanced Engineering Technology and Sciences*, 11(2).
- [5] Manner, J., 2024. *A Simulation Framework for Function as a Service (Vol. 43)*. University of Bamberg Press.
- [6] Jain, N.S., 2022. Pioneering the future of technology: integrating advanced cloud computing with artificial intelligence for scalable, intelligent systems. *World Journal of Advanced Research and Reviews*, 13, pp.847-62.
- [7] González-Granadillo, G., González-Zarzosa, S. and Diaz, R., 2021. Security information and event management (SIEM): analysis, trends, and usage in critical infrastructures. *Sensors*, 21(14), p.4759.
- [8] Mahida, A., 2022. Comprehensive review on optimizing resource allocation in cloud computing for cost efficiency. *Journal of Artificial Intelligence & Cloud Computing*. SRC/JAICC-249. DOI: doi. org/10.47363/JAICC/2022 (1), 232, pp.2-4.
- [9] Sharvani, G.S. and Resma, K.S., 2021. An auto-scaling approach to load balance dynamic workloads for cloud systems. *Turkish Journal of Computer and Mathematics Education*, 12(11), pp.515-531.
- [10] Rout, S.K., Ravindra, J.V.R., Meda, A., Mohanty, S.N. and Kavididevi, V., 2023. A Dynamic Scalable Auto-Scaling Model as a Load Balancer in the Cloud Computing Environment. *EAI Endorsed Transactions on Scalable Information Systems*, 10(5).
- [11] Dhote, S., Singh, S., Raigar, D.D., PICT, P. and Magdum, A., 2024. A Comprehensive Survey of ML-Based WAFs with Signature and Anomaly Detection. *Strad Research*, 11(4), pp.1-28.
- [12] Brar, D., 2021. *Traffic Load Balancing Techniques*.
- [13] Lin, L. and Chien, A.A., 2022. Synergistic Datacenter Load Adaptation: Sharing Information with the Grid is Key.
- [14] Erickson, B.J. and Kitamura, F., 2021. Magician's corner: 9. Performance metrics for machine learning models. *Radiology: Artificial Intelligence*, 3(3), p.e200126.

- [15] Subrahmanyam, V., Kumar, S., Srivastava, S., Bist, A.S., Sah, B., Pani, N.K. and Bhambu, P., 2023. Optimizing horizontal scalability in cloud computing using simulated annealing for Internet of Things. *Measurement: Sensors*, 28, p.100829.
- [16] Ahsan, M.M., Mahmud, M.P., Saha, P.K., Gupta, K.D. and Siddique, Z., 2021. Effect of data scaling methods on machine learning algorithms and model performance. *Technologies*, 9(3), p.52.
- [17] Perumal, R.A., 2024. *Azure Architecture Unleashed: Design, Secure, and Optimize Cloud Solutions*. Radhakrishnan Arikrishna Perumal.
- [18] Vasireddy, I., Ramya, G. and Kandi, P., 2023. Kubernetes and docker load balancing: State-of-the-art techniques and challenges. *International journal of innovative research in engineering and management*, 10(6), pp.49-54.
- [19] Ibrahim, I.M., Ameen, S.Y., Yasin, H.M., Omar, N., Kak, S.F., Rashid, Z.N., Salih, A.A., Salim, N.O. and Ahmed, D.M., 2021. Web server performance improvement using dynamic load balancing techniques: A review. *system*, 19, p.21.
- [20] Gaffurini, M., Flammini, A., Ferrari, P., Fernandes Carvalho, D., Godoy, E.P. and Sisinni, E., 2024. End-to-End emulation of LoRaWAN architecture and infrastructure in complex smart city scenarios exploiting containers. *Sensors*, 24(7), p.2024. Gaffurini, M., Flammini, A., Ferrari, P., Fernandes Carvalho, D., Godoy, E.P. and Sisinni, E., 2024. End-to-End emulation of LoRaWAN architecture and infrastructure in complex smart city scenarios exploiting containers. *Sensors*, 24(7), p.2024.
- [21] Johnson, R., 2025. *Network Address Translation Protocols and Design: Definitive Reference for Developers and Engineers*. HiTeX Press.
- [22] Kadri, S., Sboner, A., Sigaras, A. and Roy, S., 2022. Containers in bioinformatics: applications, practical considerations, and best practices in molecular pathology. *The Journal of molecular diagnostics*, 24(5), pp.442-454.
- [23] Kim, B.S., Lee, S.H., Lee, Y.R., Park, Y.H. and Jeong, J., 2022. Design and implementation of cloud docker application architecture based on machine learning in container management for smart manufacturing. *Applied Sciences*, 12(13), p.6737.
- [24] Elamin, M. and Kierkels, R., 2021. Analysis of a rarely implemented security feature: signing container images with a Notary server.
- [25] Hillion, J., 2022. *Void Processes: Minimising privilege by default*.
- [26] Chin, S., McKay, M., Ruiz, I. and Sadogursky, B., 2022. *DevOps Tools for Java Developers: Best Practices from Source Code to Production Containers*. " O'Reilly Media, Inc."
- [27] Badisa, N., Grandhi, J.K., Kallam, L., Eda, M.R. and Bulla, S., 2023. Efficient docker image optimization using multi-stage builds and nginx for enhanced application deployment.
- [28] Aung, L.H., Funabiki, N., Aung, S.T., Zhou, X., Xiang, X. and Kao, W.C., 2023. A web-based Docker image assistant generation tool for user-PC computing system. *Information*, 14(6), p.300.

- [29] Strömberg, G., 2021. Converting Hardware to a Container Solution and its Security Implication.
- [30] Chen, L., Xia, Y., Ma, Z., Zhao, R., Wang, Y., Liu, Y., Sun, W. and Xue, Z., 2022. SEAF: A Scalable, Efficient, and Application-independent Framework for container security detection. *Journal of Information Security and Applications*, 71, p.103351.
- [31] Nunes, D.J.S., 2023. CGroup Caching@ Graalvisor.
- [32] Cherukuri, B.R., 2024. Containerization in cloud computing: Comparing Docker and Kubernetes for scalable web applications. *International Journal of Scientific Research and Application*, 13(01), pp.3302-3315.
- [33] Nagase, A. and Lundholm, J., 2021. Container gardens: Possibilities and challenges for environmental and social benefits in cities. *Journal of Living Architecture*, 8(2), pp.1-19.
- [34] Rice, L., 2025. Container Security: Fundamental Technology Concepts That Protect Cloud Native Applications. " O'Reilly Media, Inc."
- [35] Dakić, V., Kovač, M. and Slovinac, J., 2024. Evolving High-Performance Computing Data Centers with Kubernetes, Performance Analysis, and Dynamic Workload Placement Based on Machine Learning Scheduling. *Electronics*, 13(13), p.2651.