

# 2218766

*by* Daniel James

---

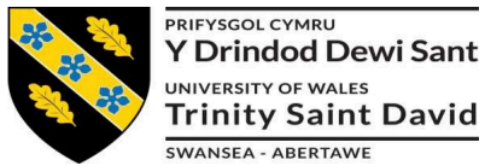
**Submission date:** 07-May-2026 11:21AM (UTC+0100)

**Submission ID:** 282573124

**File name:** 40166\_Daniel\_James\_2218766\_726729\_1769410243.docx (807.04K)

**Word count:** 4919

**Character count:** 29760



# DESIGN AND VALIDATION OF A PHYSICAL SDN DDOS DATASET FOR REAL-TIME DETECTION RESEARCH

Daniel James

2218766@student.uwtsd.ac.uk

ACCA7027: Cloud Engineering Principles and  
Practice

## Table of Contents

|                                                   |    |
|---------------------------------------------------|----|
| 1.0 Introduction .....                            | 3  |
| 2.0 Foundations of Cloud Computing .....          | 4  |
| 2.1 Virtualisation .....                          | 4  |
| 2.2 Containerisation .....                        | 4  |
| 3.0 Implementation Using Docker .....             | 6  |
| 3.1 System Overview.....                          | 6  |
| 3.2 Image Creation and Deployment.....            | 7  |
| 3.3 Data Persistence.....                         | 8  |
| 3.4 Container Networking .....                    | 10 |
| 3.5 End-to-End Application Workflow.....          | 10 |
| 4.0 Cloud Computing Concepts in Practice .....    | 12 |
| 4.1 Load Balancing .....                          | 12 |
| 4.2 Scalability.....                              | 12 |
| 4.3 Application Gateways.....                     | 13 |
| 4.4 Integration of Concepts .....                 | 13 |
| 5.0 Evolution of Cloud Computing.....             | 14 |
| 5.1 Driving Focus .....                           | 14 |
| 5.2 Serverless Computing vs Containerisation..... | 15 |
| 6.0 Conclusion .....                              | 16 |
| 7.0 References.....                               | 17 |

## **1.0 Introduction**

Cloud computing has become the standard approach for deploying modern applications, largely due to its ability to provide scalable, flexible, and on-demand resources [1]. Instead of relying on fixed physical infrastructure, organisations can now deploy services in environments that can be easily scaled and managed depending on demand [1]. This has significantly changed how applications are designed, particularly with the move towards distributed and cloud-based architectures [2].

A key concept behind cloud computing is virtualisation, which allows multiple systems to run on the same physical hardware [3]. However, more recently, containerisation has become more widely used as an alternative to traditional virtual machines [4], [5]. Containers allow applications to be packaged together with all required dependencies, ensuring consistent behaviour across different environments. This makes them particularly suitable for cloud-based deployments where portability and scalability are important [5].

This report focuses on the use of Docker to implement a containerised application and demonstrate key cloud engineering concepts in practice. As a part of this, a multi-container web application was developed consisting of a Flask-based web server, an NGINX reverse proxy for load balancing, a Redis instance for caching, and a PostgreSQL database for persistent storage. This setup allows multiple aspects of containerisation to be explored, including image creation, container networking, and the use of volumes for data persistence.

In addition to the practical implementation, the report also examines core cloud computing concepts such as load balancing, scalability, application gateways, and explains how these are applied in real world cloud environments. The relationship between scaling and load balancing is also discussed, particularly in the context of distributed systems.

Finally, the report explores the main factors driving the growth of cloud computing, including cost efficiency and flexibility [6], and compares container-based approaches with serverless computing models such as AWS Lambda and Azure Functions. This comparison highlights the trade-offs between control, scalability, and operational complexity [2].

## **2.0 Foundations of Cloud Computing**

To understand how modern cloud-based applications are designed and deployed, it is necessary to examine the underlying technologies that support them. Cloud computing is built upon mechanisms that enable efficient use of hardware resources while maintaining flexibility and scalability [1]. This section outlines the concepts used in virtualisation and containerisation.

### **2.1 Virtualisation**

Virtualisation is a key technology used in cloud computing that allows multiple systems to run on a single physical machine [3]. This is achieved through the use of a hypervisor, which acts as an intermediary between the hardware and the operating system. The hypervisor controls the allocation of resources such as processing power, memory, and storage among multiple virtual machines [7], [8].

Each virtual machine runs its own full operating system, which means it behaves like a separate computer [8]. This provides strong isolation between systems, which is important in cloud environments where multiple users may share the same physical hardware. Cloud providers such as Amazon Web Services and Microsoft Azure use this approach to run large numbers of virtual machines efficiently [9].

However, virtual machines can be resource intensive to run as they each include a full operating system [6]. This can make scaling slower, especially when compared to more lightweight approaches. As a result, while virtualisation is still widely used at the infrastructure level, other technologies such as containerisation are often used for running applications more efficiently [5], [6].

### **2.2 Containerisation**

Containerisation is a more lightweight approach compared to virtualisation [10]. Instead of running a full operating system for each instance, containers share the host's system kernel while keeping applications isolated [11]. This reduces overhead and allows containers to start much faster and use fewer resources [10]. Tools such as Docker make it easier to create and manage containers. Applications can be packaged with all their dependencies into an image, which can then be run consistently across different environments [5], [10]. This helps avoid issues where an application works in one environment but not in another.

In this project, containerisation is used to separate different parts of the system into individual services. For example, the web application, reverse proxy, cache, and database are all run in separate containers. This makes the system more flexible, as each component can be updated or scaled independently. It also supports horizontal scaling, where multiple instances of the same service can be run at the same time.

While containers are more efficient, they provide less isolation than virtual machines because they share the same operating system [10], [12]. For this reason, both approaches are often used together in cloud environments, with virtual machines handling the infrastructure and containers handling the applications [11].

### 3.0 Implementation Using Docker

This section demonstrates how containerisation was applied in practice using Docker. A multi-container application was developed to show how different services can be separated, managed, and scaled independently. The implementation includes a web application, reverse proxy, caching layer, and database, which together demonstrate key concepts in containerisation such as image creation, networking, and data persistence.

#### 3.1 System Overview

The architecture of the system is shown in Figure 1, which illustrates how user requests are processed through multiple layers of the docker application. Incoming requests are handled by NGINX, which acts as a reverse proxy and load balancer. Traffic is then distributed across multiple instances of the Flask web application, allowing the system to scale horizontally and handle increased demand more efficiently.

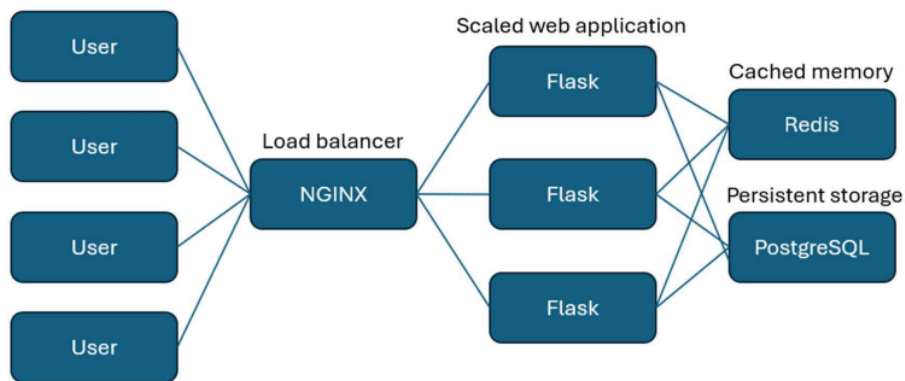


Figure 1: Container architecture showing load balancing and backend services.

Each Flask instance processes incoming requests and interacts with backend services depending on the required operation. A Redis container is used as an in-memory cache to store a visit counter, which is incremented on each request. This provides fast access to frequently updated data without the need to query the database. On the other hand, a PostgreSQL container is used for persistent storage, ensuring that user-submitted messages are retained even if the containers are restarted or deleted. The system is deployed as a set of independent containers managed through Docker Compose. Each service, including the reverse proxy, application instances, cache, and database, all run within its own container. This separation allows individual components to be scaled or

modified without affecting the rest of the system. However, as the number of services increases, managing multiple containers can introduce additional complexity compared to traditional application setups.

The running containers can be seen in Figure 3, where multiple Flask instances are active alongside the NGINX, Redis, and PostgreSQL containers. This demonstrates how the system supports horizontal scaling, with several application containers operating simultaneously behind a single load balancer.

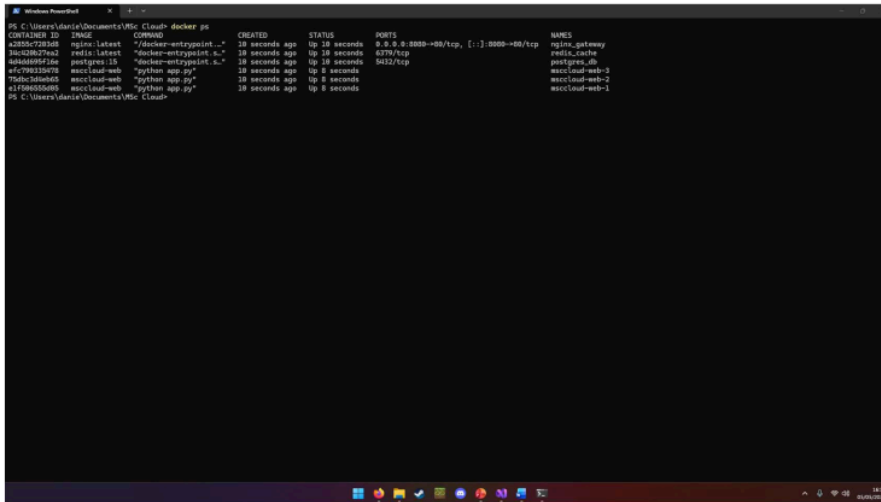


Figure 2: Running containers showing multiple Flask instances and supporting services.

This architecture reflects a typical cloud-based design, where multiple applications are split into smaller, independent services that can be deployed, scaled and managed separately [13]. This improves flexibility and allows the system to handle varying levels of demand more effectively.

### 3.2 Image Creation and Deployment

The application is packaged into a Docker image using a Dockerfile, which defines the environment required to run the Flask application. The file defines a base Python image, creates a working directory, transfers the application files into the container, installs the required dependencies, and specifies the command that launches the application. This process ensures that the application runs consistently regardless of the host system [5]. The image is built automatically through Docker Compose, which instructs docker to build the image from the current directory, allowing any changes to the application to be included in during the build process. This approach removes the need to manually build

and manage images separately. Once the image is created, Docker Compose is used to start all services defined in the configuration, including the application, reverse proxy, cache, and database. This simplifies deployment by combining image creation and container execution into one single step. The application code is copied into each Flask container during the build process, ensuring that each instance runs the same version of the application allowing scaling across multiple containers. As shown in Figure 3, the image was successfully uploaded and became available through Docker Hub. Publishing images in this way improves portability, as the application image can be downloaded and deployed on other systems without having to rebuild it locally. This reflects common practices used in cloud environments, where container images are shared through centralised registries to simplify the deployment stage and maintain consistency across different environments.

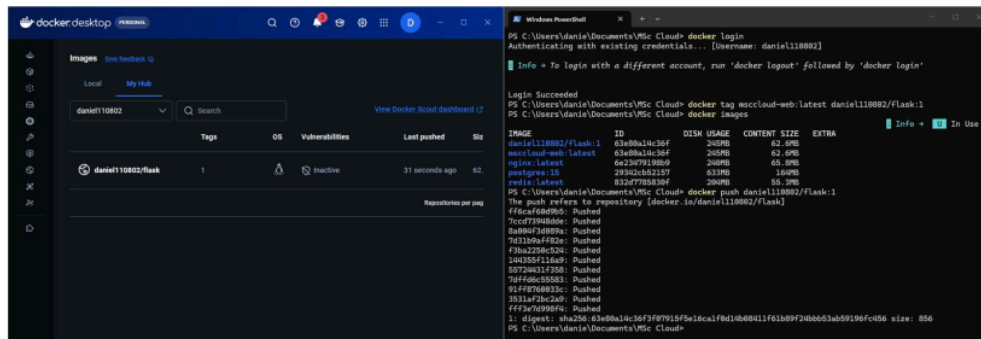


Figure 3: Flask application image successfully tagged and pushed to Docker Hub.

In addition to building a custom image for the Flask application, pre-built images from Docker Hub were used for supporting services such as PostgreSQL, Redis, and NGINX. These images are maintained by the community or official contributors and provide a reliable starting point without requiring manual setup [14]. By using these prebuilt images directly in the Docker Compose configuration, the services can be deployed quickly without the need to build them locally. This approach improves efficiency and reflects common practices in cloud environment, where standardised and reusable images are used to simplify deployment and ensure consistency across systems [9].

### 3.3 Data Persistence

By default, containers are ephemeral, meaning that any data they have stored in them is lost when the container is stopped or removed [15]. While this is acceptable or desired

for some services, it is not suitable for databases, where data must be retained over time. To address this, Docker volumes are used to store data outside of the container lifecycle [16].

In this project, a volume is used to persist data for the PostgreSQL database, as shown in Figure 4, the database container is configured to store its data in a name volume, which is mapped to the internal PostgreSQL data directory. This ensures that any data written to the database is stored independently from the container itself.

```
db:
  image: postgres:15
  container_name: postgres_db
  restart: always
  environment:
    POSTGRES_USER: ${POSTGRES_USER}
    POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    POSTGRES_DB: ${POSTGRES_DB}
  volumes:
    - db_data:/var/lib/postgresql/data
  networks:
    - docker-network
```

Figure 4: PostgreSQL container configuration showing volume mapping for persistent storage.

This approach allows the database to retain information even if the container is restarted or recreated. As a result, user-submitted messages stored by the application are not lost even when the system is stopped and started again. This behaviour can be observed in Figure 5, where previously submitted data remains visible within the application.

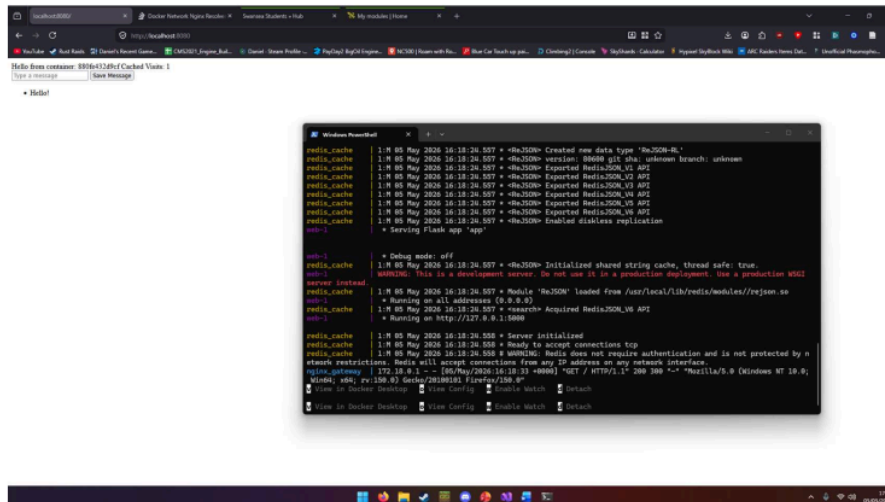


Figure 5: Application output showing persisted data stored in the PostgreSQL database.

In contrast to the PostgreSQL service, the Redis service used in this system acts as in-memory cache and does not require persistent storage. The data stored in Redis is temporary and is designed to improve performance rather than provide long-term storage. This difference highlights the importance of selecting the correct storage mechanism depending on the use-case of the service and environment.

The PostgreSQL configuration was implemented with support from external documentation, particularly when setting up environment variables, initialising the database, and how to interact with it within python [17].

### **3.4 Container Networking**

Docker provides built-in networking that allows containers within the same application to communicate with each other. When using Docker Compose, a default network is created automatically, and each service is connected to it. This allows containers to communicate using service names rather than IP addresses, simplifying configurations and improving portability [11].

In this project, all services are connected through a shared Docker network, allowing the application components to interact without requiring manual network setup. Instead of using fixed IP addresses, containers can be referenced using their service names, for example, the Flask application connects to Redis and PostgreSQL using the hostnames “redis” and “db”. This demonstrates how Docker’s internal DNS system resolves names to the correct container instances.

This approach makes the system more flexible, as container IP addresses may change when services are restarted or scaled. By relying on service names instead, the application remains stable even when multiple instances are running. This is particularly important in a scaled environment, where several Flask containers may be created and removed dynamically.

### **3.5 End-to-End Application Workflow**

The overall behaviour of this system can be understood by examining how a request is processed across all components. When a user accesses the web application, the request is first received by NGINX, which acts as a reverse proxy. NGINX then forwards the request to one of the available Flask application instances, allowing traffic to be distributed across multiple containers rather than relying on a single service. Each Flask

instance processes the request and interacts with the backend services depending on the required operation from the user. A request will trigger an update to the Redis cache, which is used to maintain a visit counter, and may involve storing or retrieving data from the PostgreSQL database. This separation of responsibilities allows the system to handle both fast, temporary data, and long-term persistent data efficiently.

To support increased demand, the application can be scaled horizontally by running multiple instances of the Flask service. This is achieved using Docker Compose, which allows additional containers to be created dynamically. As a result, multiple application instances can run simultaneously behind the NGINX reverse proxy, enabling requests to be distributed across them. This behaviour improves performance and demonstrates how load balancing and scalability work together in practice.

The integration of these components allows the system to function as a single application, despite being composed of multiple independent services. NGINX handles request distribution, Flask processes the web application, Redis provides fast access to frequently updated data, and PostgreSQL ensures data persistence. This reflects common design patterns used in cloud-based systems, where applications are built as modular and scalable services [9].

## **4.0 Cloud Computing Concepts in Practice**

Cloud based systems rely on several key concepts to ensure that applications are reliable, scalable, and able to handle varying levels of demand [18]. This section examines load balancing, scalability, and application gateways, and how they are implemented in real environments. Where relevant, these concepts will be linked back to the application to demonstrate how they are applied in practice.

### **4.1 Load Balancing**

Load balancing is used to distribute incoming network traffic across multiple instances of an application [18]. This helps prevent any single instance from becoming overloaded and improves both performance and reliability [6]. However, depending on the routing strategy and system design, load balancing may introduce additional latency. By spreading requests evenly, load balancing ensures that systems can continue to operate even if one instance fails. Cloud providers implement load balancing using managed services such as AWS Elastic Load Balancer and Azure Load Balancer [13]. These services automatically distribute traffic across multiple servers and can detect when an instance becomes unavailable, redirecting traffic to other instances. This improves availability and reduces the risk of downtime.

In this project, a similar concept is implemented using NGINX as a reverse proxy. NGINX distributes incoming requests across multiple Flask containers, allowing the application to handle increased traffic more efficiently. Although this is a simplified version, it demonstrates the same core principle.

### **4.2 Scalability**

Scalability refers to the ability of the system to handle increased demand by adding additional resources [6], [9]. This can be achieved either vertically, by increasing the capacity of a single machine, or horizontally, by adding more instances of an application. In cloud environments, horizontal scaling is more commonly used as it allows system to grow based on demand [19]. Cloud providers support scalability through services such as AWS Auto Scaling and Azure Virtual Machine Scale Sets [20]. These services automatically adjust the number of running instances depending on current traffic levels, ensuring that resources are used efficiently while maintaining performance.

In this project, horizontal scalability is demonstrated by running multiple instances of the Flask application using Docker Compose. Additional containers can be created as needed, allowing the system to handle more requests without modifying the underlying application. This reflects how cloud-based systems are designed to scale dynamically. Scalability is important because it allows applications to maintain performance under varying levels of demand, while also avoiding unnecessary resource usage when demand is low [20].

#### **4.3 Application Gateways**

Application gateways operate at the application layer and provide more advanced traffic management compared to basic load balancers [21]. They can make routing decisions based on factors such as URLs, headers, and request types [21], [22]. This allows traffic to be directed more intelligently depending on the requirements of the application. Cloud providers offer application gateway services such as AWS Application Load Balancer and Azure Application Gateway [13]. These services can perform tasks such as SSL termination, URL-based routing, and filtering of incoming requests, improving both security and performance.

In this project, NGINX performs a similar role by acting as both a reverse proxy and basic application gateway. While it does not implement all advanced features, it can route traffic to the appropriate backend service and distributing requests across multiple application instances.

#### **4.4 Integration of Concepts**

Load balancing and scalability are closely linked as they rely on each other to function effectively in a cloud environment [13]. Scaling increases the number of available application instances, while load balancing ensures that incoming traffic is distributed evenly across those instances. Without load balancing, additional resources would not be used efficiently, and without scaling, load balancing would be limited to a fixed number of instances.

In practice, these concepts are combined with application gateways to create a complete traffic management system. A typical cloud-based architecture routes incoming requests through an application gateway, which applies routing rules and security checks, before passing traffic to a load balancer [13]. The load balancer then

distributes requests across multiple instances of the application, which may scale dynamically depending on demand.

This combined approach improves system reliability and performance. If one instance fails, traffic can be redirected to others, and if demand increases additional instances can be created to handle the load [13], [18]. This ensures that the system remains responsive while making efficient use of resources.

The developed application reflects these principles on a smaller scale, with NGINX acting as both a reverse proxy and load balancer, and multiple Flask containers providing scalability. This demonstrates how these concepts work together to support modern cloud-based applications.

## **5.0 Evolution of Cloud Computing**

Cloud computing has continued to evolve as organisations require more flexible and efficient ways to deploy and manage applications [20]. Traditional infrastructure models have been gradually replaced by cloud-based approaches that allow systems to scale dynamically and be delivered as services over the internet [9]. This shift has been driven by several key factors, including cost, flexibility, and the need for faster deployment [20]. More recently, new models such as serverless computing have emerged, offering alternative ways to build and run applications without directly managing infrastructure.

### **5.1 Driving Focus**

A key factor contributing to the growth of cloud computing is cost efficiency. Traditional on-premises often involves substantial upfront spending on hardware alongside continuous maintenance. On the other hand, cloud platforms typically use a pay-as-you-go model, enabling organisations to pay only for the resources they consume [20]. This reduces the initial costs and makes it easier to scale systems without large financial commitments.

Another factor affecting the use of cloud computing is flexibility which allows applications to be deployed and accessed from anywhere, making it easier to support distributed systems and remote access [9]. Resources can be adjusted quickly depending on demand, which is particularly useful for applications with varying workloads. Scalability on cloud platforms allow systems to scale dynamically, adding or

removing resources as needed which also plays a key role [19]. This ensures that applications can handle changes in demand without affecting performance. This concept is reflected in the containerised application, where multiple instances of the application can be created to handle increased load.

Finally, the speed of development has contributed to the growth of cloud computing, as applications can be developed, tested, and deployed much faster using cloud services compared to traditional infrastructure [23]. Tools such as Docker further support this by allowing applications to be packaged and deployed consistently across different environments [11].

## **5.2 Serverless Computing vs Containerisation**

Serverless computing is a cloud-based model where applications are executed without the need to manage servers directly [24]. Instead of running continuously, code is executed in response to specific events, and resources are allocated automatically by the cloud provider [25]. Services such as AWS Lambda and Azure Functions are commonly used to implement this approach.

One of the main advantages of serverless computing is its ability to scale automatically [25]. Resources are allocated only when needed, which can reduce the costs for applications with unpredictable or low usage. In addition, there is less operational overhead, as the cloud provider manages infrastructure, scaling, and maintenance [25]. However, serverless computing also has limitations. There is less control over the execution environment compared to the container-based systems, and applications may be subject to constraints such as execution time limits and cold start delays [10], [26]. This can make serverless less suitable for long running or highly customised applications.

Conversely, containerisation provides more control over the application environment. Containers run continuously and can be configured to meet specific requirements, making them more suitable for complex systems [10]. In this implemented application, containerisation allows multiple services to be managed and scaled independently, providing flexibility and consistency.

Both approaches are influenced by the same driving forces, such as cost efficiency and scalability, but they address these challenges in different ways [20]. Serverless focuses on reducing management overhead and optimising cost for event-driven workloads, while

containerisation provides greater control and flexibility for more complex applications [10], [25].

Overall, the choice between serverless and containerisation depends on the specific requirements of the application, with many modern systems combining both approaches to achieve the best balance between control, scalability, and cost [9].

## **6.0 Conclusion**

This report has examined the principles of cloud computing and demonstrated how containerisation can be used to build and deploy a scalable application. The use of Docker allowed the application to be divided into multiple independent services, including a web server, reverse proxy, caching layer, and database. This approach improved flexibility and made it possible to manage and scale each component separately. The practical implementation demonstrates how these concepts are applied together within a working system. By combining load balancing and horizontal scaling, the application can distribute traffic across multiple instances while maintaining performance. The use of a reverse proxy, caching layer, and persistent database further highlighted how different components can be integrated to create a reliable and efficient service.

The comparison between containerisation and serverless computing shows that while both approaches aim to improve scalability and reduce operational overhead, they are suited to different types of applications. Containerisation provides greater control over the environment and is better suited to systems that require multiple interacting services, whereas serverless computing is more appropriate for smaller, event-driven workloads.

Overall, the results of this project show that modern cloud applications are built by combining multiple technologies rather than relying on a single approach. The ability to scale, distribute, and manage services independently is a key advantage of cloud-based systems, and containerisation plays an important role in supporting this.

## 7.0 References

- [1] P. Mell and T. Grance, 'The NIST Definition of Cloud Computing'.
- [2] A. Amiri, U. Zdun, and A. Van Hoorn, 'Modeling and Empirical Validation of Reliability and Performance Trade-Offs of Dynamic Routing in Service- and Cloud-Based Architectures', *IEEE Trans. Serv. Comput.*, vol. 15, no. 6, pp. 3372–3386, Nov. 2022, doi: 10.1109/TSC.2021.3098178.
- [3] G. S. S. Ruthvik, M. S. Subrahmanyam, P. S. Chandh, and J. V. Ananthi, 'Network Function Virtualization for Cloud and Edge Computing: A Comprehensive Survey and Future Research Directions', in *2025 International Conference on Intelligent Computing, Information and Control Systems (ICOIICS)*, Lalitpur, Nepal: IEEE, Nov. 2025, pp. 710–714. doi: 10.1109/ICOIICS67115.2025.11390513.
- [4] E. C. Yildiz, D. E. Aktas, E. Unal, H. Tuzun, and M. S. Aktas, 'Management of Virtualization Technologies with Complex Event Processing', in *2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, Istanbul, Turkey: IEEE, Jun. 2020, pp. 1–4. doi: 10.1109/ICECCE49384.2020.9179215.
- [5] H. Manninen, V. Jaaskelainen, and J. O. Blech, 'Performance Evaluation of Containerization Platforms for Control and Monitoring Devices', in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vienna, Austria: IEEE, Sep. 2020, pp. 1061–1064. doi: 10.1109/ETFA46521.2020.9211901.
- [6] A. V. Papadopoulos and M. Maggio, 'Virtual machine migration in cloud infrastructures: Problem formalization and policies proposal', in *2015 54th IEEE Conference on Decision and Control (CDC)*, Osaka: IEEE, Dec. 2015, pp. 6698–6705. doi: 10.1109/CDC.2015.7403274.
- [7] M. S. Dildar, N. Khan, J. B. Abdullah, and A. S. Khan, 'Effective way to defend the hypervisor attacks in cloud computing', in *2017 2nd International Conference on Anti-Cyber Crimes (ICACC)*, Abha, Saudi Arabia: IEEE, Mar. 2017, pp. 154–159. doi: 10.1109/Anti-Cybercrime.2017.7905282.
- [8] S. Sotiriadis, N. Bessis, F. Xhafa, and N. Antonopoulos, 'Cloud Virtual Machine Scheduling: Modelling the Cloud Virtual Machine Instantiation', in *2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems*, Palermo, Italy: IEEE, Jul. 2012, pp. 233–240. doi: 10.1109/CISIS.2012.113.
- [9] R. Gohil and H. Patel, 'Comparative Analysis of Cloud Platform: Amazon Web Service, Microsoft Azure, And Google Cloud Provider: A Review', in *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, Kamand, India: IEEE, Jun. 2024, pp. 1–5. doi: 10.1109/ICCCNT61001.2024.10725914.
- [10] A. M., A. Dinkar, S. C. Mouli, S. B., and A. A. Deshpande, 'Comparison of Containerization and Virtualization in Cloud Architectures', in *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, Bangalore, India: IEEE, Jul. 2021, pp. 1–5. doi: 10.1109/CONECCT52877.2021.9622668.
- [11] W. Wang, 'Research on Using Docker Container Technology to Realize Rapid Deployment Environment on Virtual Machine', in *2022 8th Annual International Conference on Network and Information Systems for Computers (ICNISC)*,

Hangzhou, China: IEEE, Sep. 2022, pp. 541–544. doi: 10.1109/ICNISC57059.2022.00112.

- [12] X. Lyu, 'Balancing Three Important Goals for Runtimes - Isolation, High Performance, and Resource Efficiency', in *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, CA, USA: IEEE, Sep. 2022, pp. 60–62. doi: 10.1109/ACSOSC56246.2022.00031.
- [13] S. Sharma, R. P. Mahapatra, D. Seth, and M. Kapil, 'Advanced Deployment Strategies for Elastic Load Balancing in AWS: A Comprehensive Study on Multi-Tier Architecture Optimization', in *2024 International Conference on Communication, Computing and Energy Efficient Technologies (I3CEET)*, Gautam Buddha Nagar, India: IEEE, Sep. 2024, pp. 285–289. doi: 10.1109/I3CEET61722.2024.10993779.
- [14] M. Aldiabat, Q. M. Yaseen, and Q. A. Ein, 'An Efficient Random Forest Classifier for Detecting Malicious Docker Images in Docker Hub Repository', *IEEE Access*, pp. 1–1, 2024, doi: 10.1109/ACCESS.2024.3506663.
- [15] I. Hassan, 'Leveraging Apache Guacamole, Linux LXD and Docker Containers to Deliver a Secure Online Lab for a Large Cybersecurity Course', in *2022 IEEE Frontiers in Education Conference (FIE)*, Uppsala, Sweden: IEEE, Oct. 2022, pp. 1–9. doi: 10.1109/FIE56618.2022.9962510.
- [16] K. Sri. B and K. Mahalakshmi, 'Exploring Large Scale Docker Image Vulnerability and Storage Performance Analysis Using High Performance Container-Based Docker Environment', in *2024 International Conference on System, Computation, Automation and Networking (ICSCAN)*, PUDUCHERRY, India: IEEE, Dec. 2024, pp. 1–7. doi: 10.1109/ICSCAN62807.2024.10894527.
- [17] 'PostgreSQL 18.3 Documentation', PostgreSQL Documentation. Accessed: May 06, 2026. [Online]. Available: <https://www.postgresql.org/docs/18/index.html>
- [18] S. Joshi and U. Kumari, 'Load balancing in cloud computing: Challenges & issues', in *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, Greater Noida, India: IEEE, Dec. 2016, pp. 120–125. doi: 10.1109/IC3I.2016.7917945.
- [19] K. Rai, B. Sahana, A. N. Pai, S. Gautham, and U. Dhanush, 'Vertical Scaling of Virtual Machines In Cloud Environment', in *2021 International Conference on Recent Trends on Electronics, Information, Communication & Technology (RTEICT)*, Bangalore, India: IEEE, Aug. 2021, pp. 458–462. doi: 10.1109/RTEICT52294.2021.9573715.
- [20] N. A. Polani and U. Haider, 'Comparing the Performance and Cost of Different Cloud Service Providers for Small Applications: A Comparison between AWS and Azure', in *2024 IEEE/ACM 17th International Conference on Utility and Cloud Computing (UCC)*, Sharjah, United Arab Emirates: IEEE, Dec. 2024, pp. 182–184. doi: 10.1109/UCC63386.2024.00032.
- [21] 'What is an Application Load Balancer? - Elastic Load Balancing'. Accessed: May 06, 2026. [Online]. Available: <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html>
- [22] M. Song, C. Zhang, and E. Haihong, 'An Auto Scaling System for API Gateway Based on Kubernetes', in *2018 IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, Beijing, China: IEEE, Nov. 2018, pp. 109–112. doi: 10.1109/ICSESS.2018.8663784.

- [23] Shuai Zhang, Xuebin Chen, Shufen Zhang, and Xiuzhen Huo, 'The comparison between cloud computing and grid computing', in *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*, Taiyuan, China: IEEE, Oct. 2010, pp. V11-72-V11-75. doi: 10.1109/ICCASM.2010.5623257.
- [24] 'What is AWS Lambda? - AWS Lambda'. Accessed: May 06, 2026. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>
- [25] G. G. Hegde, V. V. Murthy, and M. A. K. Sab, 'Serverless Computing: The Future of Cloud Infrastructure and Development', in *2025 Third International Conference on Emerging Applications of Material Science and Technology (ICEAMST)*, Bengaluru, India: IEEE, Nov. 2025, pp. 1166–1171. doi: 10.1109/ICEAMST67459.2025.11336042.
- [26] S. Sivaranjani and R. Anandhi, 'Cold Start Challenges in Serverless Computing for Cloud Enabled IoT: Performance Impact and Mitigation Strategies in Intelligent Metropolitan Zones', in *2025 IEEE First International Conference on Innovations in Engineering and Next-Generation Technologies for Sustainability (ICINVENTS)*, Coimbatore, India: IEEE, Nov. 2025, pp. 1–5. doi: 10.1109/ICINVENTS64613.2025.11402364.

## FINAL GRADE

## GENERAL COMMENTS

77/100**Engagement with Literature Skills**

26 references, almost entirely Tier 1 (22 IEEE papers, 3 official vendor docs), citations integrated throughout.

[1] incomplete (no year, publisher, URL). [10] truncated first author. [21] and [24] missing access dates. [23] a 2010 paper cited for a current-practice claim.

13/15

**Knowledge and Understanding Skills**

Virtualisation, containerisation, load balancing, scalability, gateways, and serverless are all covered with named AWS/Azure services and limitations noted.

Driving forces (5.1) lists factors without analysis. Cloud provider comparisons stay descriptive and lack critical awareness of trade-offs.

13/20

**Cognitive and Intellectual Skills**

Trade-offs and limitations are identified in places, and Section 4.4 synthesises load balancing, scalability, and gateways coherently.

Project design choices (Flask, NGINX, PostgreSQL, Redis) are not justified. Driving forces are listed not weighed. The conclusion summarises rather than argues.

13/20

**Practical Application Skills**

docker ps (Figure 2) confirms six running containers (nginx\_gateway, redis\_cache, postgres\_db, three Flask instances). DockerHub push evidenced in Figure 3. Compose YAML (Figure 4) shows volume mapping and docker-network. Application serving

requests with Redis counter and PostgreSQL persistence confirmed in Figure 5.

Task 2 cloud concepts and Task 3 serverless are theoretical only - Live demonstration presented on Teams. Good clear presentation of the system.

35/40

### **Transferable Skills for Life and Professional Practice**

Title page carries the wrong title from a different submission. ToC and numbered headings present. All five figures captioned (Figure 1-5). No table of figures (required at 4+). One cross-reference error: running containers called "Figure 3" in body, captioned "Figure 2".

DNS, SSL, and URL unexpanded on first use. No first-person language. IEEE format consistent. Spelling broadly good.

3/5

**Total: 72/100**

|         |
|---------|
| PAGE 1  |
| PAGE 2  |
| PAGE 3  |
| PAGE 4  |
| PAGE 5  |
| PAGE 6  |
| PAGE 7  |
| PAGE 8  |
| PAGE 9  |
| PAGE 10 |
| PAGE 11 |

PAGE 12

---

PAGE 13

---

PAGE 14

---

PAGE 15

---

PAGE 16

---

PAGE 17

---

PAGE 18

---

PAGE 19

---