



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

**Cloud deployment and Containerisation in Cloud Computing: A
Practical Implementation Using HanStream Application.**

Submitted by:

MD Azizul Khan

Student ID: 2514708

Module Code: ACCA7027

Submitted to:

Mark Huntly

Lecturer, Cloud Engineering Principles and Practice

Table of Contents

Introduction.....	3
1. Docker and Containerisation.....	4
1.1 The Principles of Virtualisation.	4
1.2 Containerisation and Docker.....	5
1.3 Docker Images and the Dockerfile.	6
1.4 Pushing pictures to Docker Hub.	9
1.5 Container Volumes and Networks	10
1.6 Docker and Microservices Architecture.	12
2. Load Balancing, Scalability and Application Gateways.....	13
2.1 Load Balancing	13
2.2 Scalability and Auto-Scaling	14
2.3 Application Gateways.....	15
2.4 Combining All Three: A Unified Architecture	16
3. Driving Forces Behind Cloud Expansion and Serverless Computing	17
3.1 Main Driving Forces Leading to Cloud Adoption.....	17
3.2 Serverless Computing: Concept and Implementation.....	18
3.3 Containers vs. Serverless: A Critical Comparison.....	19
Conclusion	21

Introduction

The nature of cloud computing has fundamentally re-architected the way organisations design, deploy and scale software systems. The resulting change of physical infrastructure to virtualised and then containerised environments have thus afforded a degree of operational flexibility, cost efficiency, and architectural agility hitherto previously unachievable except in the largest technology enterprises. The key element in this change is containerisation - a type of operating system-level virtualisation, which packages application code and its dependencies into portable, isolated units capable of executing in a consistent way no matter the host environment being used.

This report discusses containerisation in the light of microservices architecture using a Python Flask web application - the HanStream K-Drama and Movie Watchlist - as the vehicle of practical demonstration. The application has been developed, and containerised with the Docker application, and published to Docker Hub as part of the practical work underpinning this report. This report is divided into three parts: Section 1 includes the technical principles of virtualisation and containerisation and demonstrates the construction of a Dockerfile, the creation and running of container images, creation of volumes and networks, and publication of an image to Docker Hub. Section 2 critically analyzes the three fundamental concepts of cloud computing of load balancing, scalability, and application gateways, and with the example of Amazon Web Services as the cloud provider. Section 3 explores the key factors behind cloud growth and compares containerisation with the new serverless computing paradigm. Combined, these sections prove the main point of this report: that Docker-based containerisation is the technology that underlies the modern architecture of microservices, and that its use is driven by the same economic and operational factors that have continued to drive cloud computing around the world.

Video demonstration Link:

<https://drive.google.com/file/d/1etCPZxk2gRBRI0skPAaUptbs0O9K499L/view?usp=sharing>

1. Docker and Containerisation

1.1 The Principles of Virtualisation.

In determining the importance of containerisation, we would first have to look at the virtualisation technologies that came before it. Virtualisation is the process of abstraction of physical computing resources: processors, memory, storage and networking, into logical units, which can be managed and allocated without reference to underlying hardware. This abstraction can enable many isolated computing environments to run on a single physical server in a highly scalable way, dramatically improving the utilisation of hardware and the cost of operating infrastructure at scale [1].

Hypervisor-based virtualisation was the most common type of virtualisation before the emergence of containers. A hypervisor (also called a Virtual Machine Monitor (VMM)) is a software layer that provides the creation and management of virtual machines (VMs), each of which runs its own complete operating system called the guest OS. There are two known types of hypervisor: Type 1 hypervisor, a type of hypervisor that can be directly run on the actual hardware (bare metal) of the machine being virtualized, such as Microsoft Hyper-V and VMware ESXi; and Type 2 hypervisor, a type of hypervisor that can be directly run on top of a host operating system, such as VirtualBox. Type 1 hypervisors have better performance since they do not experience an extra latency of the host OS layer as opposed to Type 2 hypervisors, which are easier to install [1].

Although hypervisor-based VMs provided revolutionary benefits in terms of resource consolidation and isolation, they established a significant limitation in terms of architecture: each virtual machine has to contain its complete operating system. Ten VMs deployed in a single physical host thus needs ten separate OS installations, which in itself consumes memory, storage, and CPU cycles just to be up and running. This overhead is especially problematic when the workload being served is a small microservice of a focused application component that may only need a fraction of the resources used by itself as its own guest OS [2].

This inefficiency is specifically what containerisation is directly solving, and why Docker came to be a platform technology on which cloud-native application architecture is built.

1.2 Containerisation and Docker

Containerisation is a type of operating system-level virtualisation where applications and their dependencies are bundled into closed systems called containers. Containers do not have their own operating system kernel, unlike virtual machines; they share the kernel of the underlying operating system, and remain isolated from each other using Linux kernel features such as namespaces and control groups (cgroups). This architectural difference can result in significant reduction in resource overhead: a virtual machine can demand many gigabytes of memory to just run its guest OS, whereas a container only adds those application binaries and libraries not already provided by the host OS [2].

The most used container runtime and toolchain in the current market are Docker. Docker was introduced in 2013 and was a derived extension of the existing Linux container (LXC) model that provided a portable and developer-friendly abstraction that standardised the way containers were created, distributed, and executed [3]. The most valuable contribution made by Docker, as Bernstein observed, was not the invention of container isolation, which had been a part of Linux since years past, but the creation of a coherent toolchain that made containers viable to the everyday developer of software and the everyday deployer of software and services to a cloud provider [3]. The architecture of Docker will be in the form of a client-server system, whereby the Docker client will interact with the Docker daemon (dockerd), which is a background service; that is, it is a service that may be configured to be started automatically.

This architecture has a great practical value. In a hypervisor-based design, where each service holds its own instance of an OS, and where each service is dedicated to a particular task, Docker enables multiple containerised applications to run on the same host OS, sharing the kernel, but remaining completely separate from each other. According to course material in this module observed, the overhead of operating a microservice in a VM is OS overhead of about 610% of overall CPU load, compared to the much smaller overhead that can be achieved when the same microservice shares an OS kernel via Docker [2]. This efficiency benefit is not just theoretical - it has enabled containers to become the unit of deployment in a contemporary cloud-native architecture.

1.3 Docker Images and the Dockerfile.

The key point in the functioning of Docker is the image concept. A Docker image is an immutable, layered filesystem that includes all the bits necessary to run an application: The runtime environment, application code, libraries, configuration files, and dependencies. Images are implemented in layers, with each command in the Dockerfile producing a new layer on top of the existing ones, being read-only. This layered design has two significant advantages: unchanged between builds layers are cached and reused, which accelerates future builds; and shared layers can be shared across multiple images, which will reduce the cost of storage [5].

The declarative description in order to construct a Docker image is the Dockerfile. The file has every instruction that is associated with one build layer. The most important Docker file commands are the following [4]:

Table 1: Several important instructions of a docker file and how they are used to form the v2-flask image [4].

Instruction	Purpose	Usage in v2-flask Dockerfile
FROM	Specifies the base image from which the build begins; defines the starting OS and runtime layer.	FROM python:3.11-slim
WORKDIR	Sets the working directory inside the image; all subsequent instructions execute relative to this path.	WORKDIR /app
COPY	Copies files or directories from the host build context into the image filesystem.	COPY requirements.txt . / COPY . .
RUN	Executes a shell command during the image build process; each RUN instruction creates a new image layer.	RUN pip install -r requirements.txt
EXPOSE	Documents the network port the container listens on at runtime; does not publish the port to the host.	EXPOSE 5000

Instruction	Purpose	Usage in v2-flask Dockerfile
CMD	Specifies the default command executed when a container starts; can be overridden at runtime.	CMD ["python", "app.py"]

In the present project, the following Dockerfile was built and built successfully to containerise the HanStream K-Drama Watchlist application, a Python Flask web application, rendering a browsable catalogue of Korean dramas and films:

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
RUN mkdir -p data
EXPOSE 5000
CMD ["python", "app.py"]
```

The image was created with the command: `docker image build -t v2-flask`. where the `-t` option is used to give the image the tag `v2-flask`. The build output indicated that it was successfully built in layers, with each layer built successfully, finally writing the image to the local Docker image store. Figure 1 below shows the application itself was then run with the command `docker container run -d -p 5000:5000 v2-flask`, where `-d` is used to run the container in the detached mode and `-p 5000:5000` is used to map port 5000 on the host to port 5000 on the inside of the container.

```
Terminal
PS C:\Users\azizu> cd "C:\Users\azizu\Downloads\cloud_assignment\flask-codebase"
PS C:\Users\azizu\Downloads\cloud_assignment\flask-codebase> docker image build -t v2-flask .
[+] Building 34.3s (11/11) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile                0.6s
=> => transferring dockerfile: 198B                                0.1s
=> [internal] load metadata for docker.io/library/python:3.11-slim 3.6s
=> [internal] load .dockerignore                                   0.1s
=> => transferring context: 2B                                       0.0s
=> [1/6] FROM docker.io/library/python:3.11-slim@sha256:a5b427ace4900267d93db34138e512325c6fa6af 11.9s
=> => resolve docker.io/library/python:3.11-slim@sha256:a5b427ace4900267d93db34138e512325c6fa6af8 0.1s
=> => sha256:6f92665ed17afc6850bfbeb3fb681d6e1038fe59e2020ab126b859ec572da21b 250B / 250B 0.3s
=> => sha256:57fb71246055257a374deb7564ceca10f43c2352572b501efc08add5d24ebb61 29.78MB / 29.78MB 5.5s
=> => sha256:01f59aef9b5c2caa2870aa8b9b8b5806ea3c36d893cd6e2467e252fc1b1fea46 1.29MB / 1.29MB 1.3s
=> => sha256:fb4c70443787d9baef637d0b257f21b935d5feb6481f1ccdf4d07f48b2e393c1 14.37MB / 14.37MB 3.8s
RAM 2.08 GB CPU 0.00% Disk: 1.71 GB used (limit 1006.85 GB)
```

Figure 1: Terminal output with a success notification of the build of the v2-flask Docker image.

The outcome is a fully operational web application, accessible at <http://localhost:5000>, which shows a search-enabled K-drama catalogue that has Watch Later and Watched tracking functionality - all served out of a Docker container. A special `/health` endpoint, at <http://localhost:5000/health> provides a JSON response to confirm the status of the services and the number of watchlists, demonstrating the API functionality of Flask as a component of a microservice.

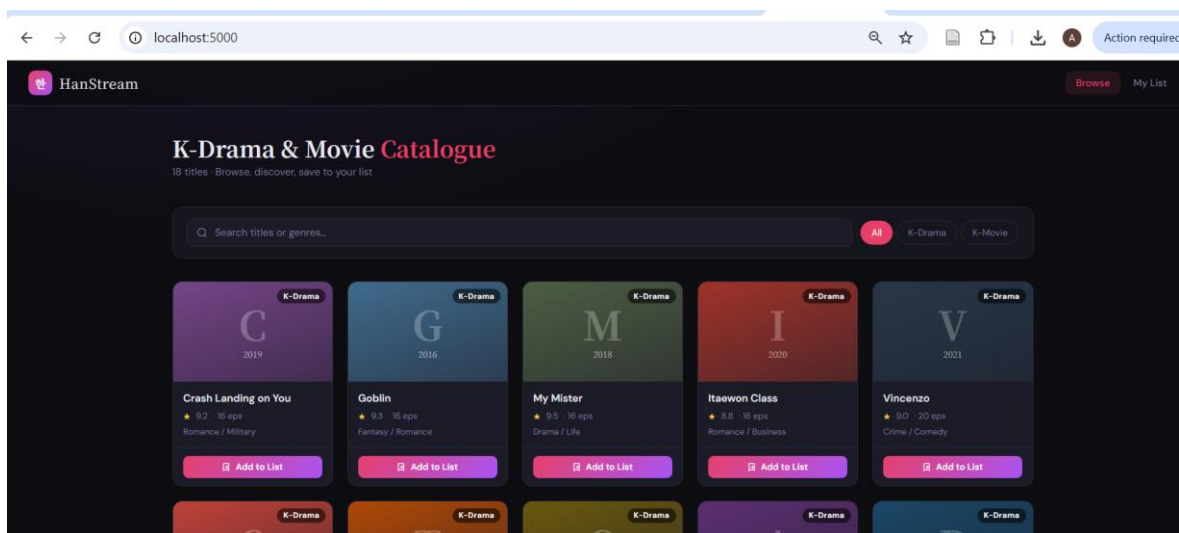
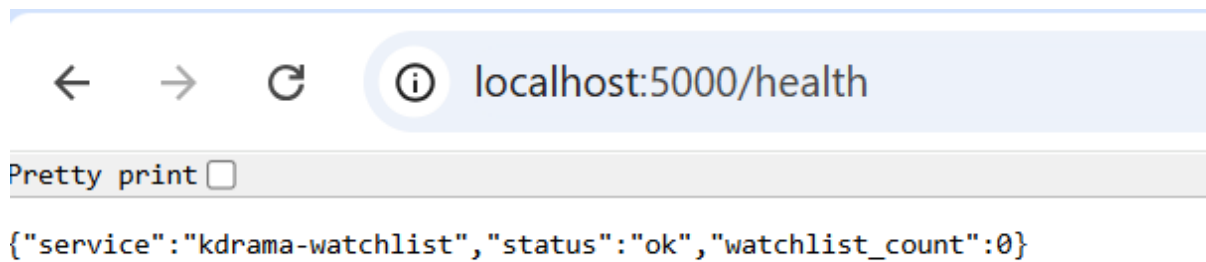


Figure 2: HanStream K-Drama Watchlist application running in the v2-flask Docker container, and accessed at localhost:5000.



[INSERT FIGURE 3 HERE -Browser at localhost:5000/health -JSON health response]

Figure 3: Flask microservice interface that displays the /health API endpoint returning a JSON status response.

1.4 Pushing pictures to Docker Hub.

After constructing an image, it may be pushed to a container repository to become accessible to be deployed to various computers or cloud-based systems. The default public registry, operated by Docker Inc., and serves as the primary channel of distribution of the both official base images and user-published images [5].

The publishing of an image consists of three steps. First, one has to tag the image with the target repository name using the following command: `docker image tag v2-flask: latest`, where `azizulkhaan` is the Docker Hub account user name and `latest` is the image tag. Second, it needs to be authenticated with `docker login`. Third, the tagged image is pushed to the registry using `docker push azizulkhaan/v2-flask:latest`.

As confirmed in Figure 4, all the image layers have been successfully transmitted to Docker Hub, with the final digest “`sha256:(##### a sequence of number)`” verifying the integrity of the pushed image. The containerised application can be pulled and run on any machine with Docker installed using a single command `azizulkhaan/v2-flask:pull` and `run` and the repository `azizulkhaan/v2-flask` is now publicly available, meaning that it is possible to pull and run the containerised application on any machine with Docker installed using a single command `azizulkhaan/v2-flask:pull` and `run`. One of the most remarkable practical advantages of container-based deployment is their portability [3].

```
PS C:\Users\azizu\Downloads\cloud_asssignemnt\flask-codebase> docker push azizulkhaan/v2-flask:latest
The push refers to repository [docker.io/azizulkhaan/v2-flask]
01f59aef9b5c: Pushed
fb4c70443787: Pushed
faa54275695b: Pushed
f20a6b7d0f88: Pushed
57fb71246055: Pushed
e15806f92ed1: Pushed
aa1f568b3c13: Pushed
efa45399f912: Pushed
6f92665ed17a: Pushed
cc97f520d677: Pushed
latest: digest: sha256:3fafc4f3a3ddd9ea753e570f1a9c4007fb61d1c45222a079ae60568284621c4f size: 856
PS C:\Users\azizu\Downloads\cloud_asssignemnt\flask-codebase>
```

RAM 2.19 GB CPU 0.12% Disk: 1.71 GB used (limit 1006.85 GB)

Figure 4: Terminal output that shows that the azizulkhaan/v2-flask:latest image was successfully pushed to Docker Hub.

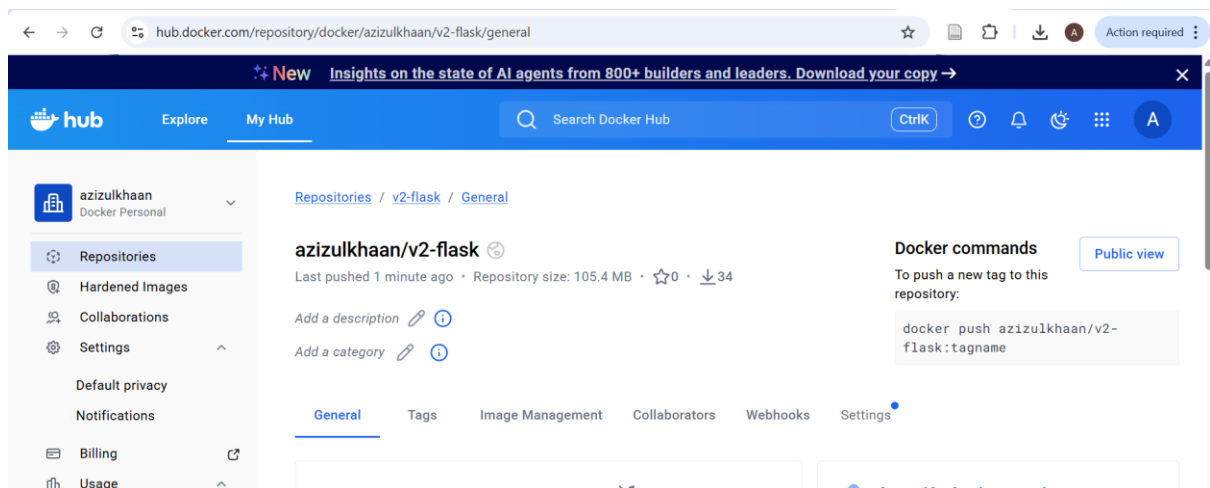


Figure 5: The Docker Hub repository page of azizulkhaan/v2-flask, which is showing the image that was pushed a minute ago.

1.5 Container Volumes and Networks

There are two additional Docker primitives that are worth looking at: volumes and networks. Both were developed as a part of the practical work on this project and both of them refer to the different architectural issues in the containerised system design.

A Docker volume is a technology of continuing data after the demise of a container. By default, any data written within the filesystem of a container is ephemeral - it exists as long as the container is running, and is lost when the container is removed. This is suitable to stateless

workloads, but applications needing to maintain some data across restarts need to have a separate persistence layer. The Docker daemon manages Docker volumes, which are stored outside of the writable layer of a container and may be mounted into a container or multiple containers at the same time [6]. To achieve this project, the volume v2-vol was created using `docker volume create v2-vol` and they were checked using `docker volume ls` as in Figure 6. This volume in a microservices deployment of production of watchlist JSON data would be used as a persistent store to maintain user data after container restarts and updates.

```

PS C:\Users\azizu\Downloads\cloud_asssignemnt\flask-codebase> docker volume create v2-vol
>> docker volume ls
v2-vol
DRIVER      VOLUME NAME
local       v2-vol
PS C:\Users\azizu\Downloads\cloud_asssignemnt\flask-codebase>
RAM 2.15 GB CPU 0.88% Disk: 1.71 GB used (limit 1006.85 GB)

```

Figure 6: Output of terminal verifying that Docker volume v2-vol has been created.

Docker networking is a layer of communication between containers. Containers can communicate with each other using container name - the internal DNS of Docker will automatically resolve the names of the containers to the corresponding IP address. This is how multi-container microservice architectures get constructed: a Flask API container, a database container and a caching container can each be independently operated and communicate over a common network [7]. As shown in Figure 7, the network v2-net was set up with the `docker network create v2-net`. The microservices architecture, which is depicted by this project, would have v2-net as the isolated communication channel through which the Flask application container will connect with supporting services.

```

PS C:\Users\azizu\Downloads\cloud_asssignemnt\flask-codebase> docker network create v2-net
>> docker network ls
c17b84bc6289efa43800690ef90ce13174cd18304d60b6db5c9a45d7296f3073
NETWORK ID      NAME      DRIVER      SCOPE
d0fa9ea71199    bridge   bridge      local
c8f3f828ffef    host     host        local
286b3f619733    none     null        local
c17b84bc6289    v2-net   bridge      local
PS C:\Users\azizu\Downloads\cloud_asssignemnt\flask-codebase> 
RAM 2.14 GB CPU 1.00% Disk: 1.71 GB used (limit 1006.85 GB)

```

Figure 7: Terminal output that Docker network v2-net had been created.

1.6 Docker and Microservices Architecture.

The Flask application designed in this project is not just a standalone web service - the application is a single microservice in a larger cloud-native system. Microservices architecture An architecture of software design where an application is broken down into a set of small, independently deployable services, each having a single business capability, and communicating over clearly defined interfaces, usually HTTP APIs or message queues [3].

The enabler technology of this architectural pattern is Docker. Since every microservice is packaged as its own container image it can be built, tested, deployed and scaled completely independently of the other services in the system. Any change of the watchlist service does not mean that the authentication service or the recommendation service has to be redeployed: both of them can evolve independently of each other, on a release cycle. This independence removes the deployment coupling that accompanies monolithic applications that might need the entire application to be rebuilt and redeployed with a single change [2].

Figure 8 shows the placement of the Flask microservice deployed in this project in a larger cloud-native deployment model, with multiple container instances running behind a load balancer on a shared Docker network.

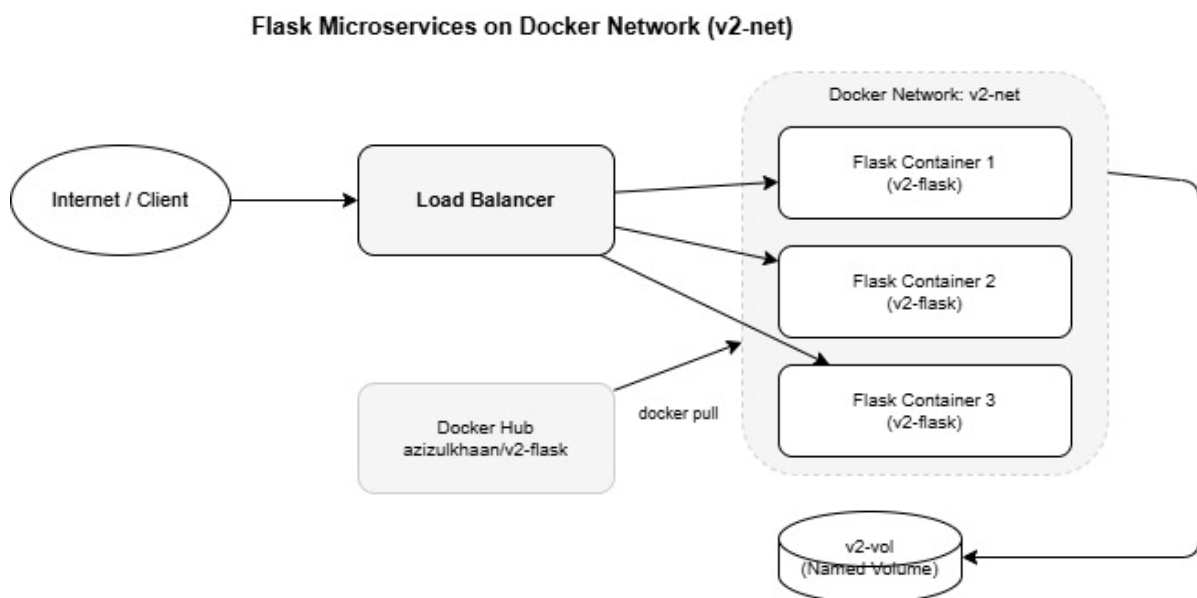


Figure 8: Microservices architecture showing three Flask container instances on v2-net, with a load balancer distributing incoming requests and Docker Hub providing the shared image registry.

Pahl showed that containers are uniquely well-suited to platform-as-a-service (PaaS) cloud environments specifically because they can be the unit of microservice packaging each container packaging a single service, deployable to any cloud host, and orchestrable using systems like Kubernetes or Docker Swarm [2]. The HanStream application reflects this principle: it is packaged as a single image, pushed to a public registry, and runnable on any infrastructure that supports Docker.

2. Load Balancing, Scalability and Application Gateways.

2.1 Load Balancing

With containerised microservices being scaled out, a service running on a single instance is easily a bottleneck. There is no redundancy in having a single Flask container to handle all user requests; conversely, when the demand peaks and response times are slow, then there is no redundancy in the single Flask container. The technology that addresses this constraint is load balancing, which involves the distribution of the incoming traffic across the network to multiple instances of the backend server so that no single server has a disproportionate amount of traffic [8].

There are three fundamental elements of a load balancer. The frontend IP address pool is the public or private IP address by which clients communicate - in the eyes of the client, they are communicating with one endpoint even though there may be many backend servers. The load balancing rule is the mapping between a given frontend IP and port to a backend pool and port that defines how traffic is sent between the entry point and the serving infrastructure. The address pool of the backend is the set of servers or containers that will handle the requests - each member of the pool can potentially serve the same request, and thus traffic can be evenly distributed across all the members of the pool [8].

Practically, load balancers would work at OSI model (Layer 4) network transport layer. On this level the load balancer will make routing decisions based on the IP address and port number only, and does not examine the content of the request to make routing decisions. This causes Layer 4 load balancing to be very fast and protocol-agnostic but not able to differentiate between different types of requests arriving on the same port [8].

Amazon Web Services achieves this feature with the help of the Elastic Load Balancer (ELB) which automatically distributes incoming application traffic across more than one registered target EC2 instances, containers, or IP addresses in one or more availability zones [8]. In the case of a containerised Flask application running on AWS, the ELB would have a number of targets in a backend pool, which are container instances of the application. HTTP requests would be redirected to them based on the configured algorithm. The AWS supports a variety of ELB variants, but the Application Load Balancer (ALB) which AWS uses as its term equivalent to what Azure refers to as an Application Gateway is the most relevant to deploy a web application as it adds Layer 7 routing capability on top of basic load distribution [8].

2.2 Scalability and Auto-Scaling

Load balancing allocates traffic among available instances, but does not provide answers itself to the question of how many instances should be in existence at any given time. This is where scalability is important. Scalability in cloud computing is defined as the ability to scale-up or scale-down infrastructure resources dynamically in response to changing demand and is one of the defining characteristics of cloud platforms [9].

There are two dimensions of scaling that are applicable. Vertical scaling (scaling up) is the increase of the computational resources provided to a single instance - adding CPU or memory to an existing server. Although simple, with a hard ceiling which is dictated by the maximum specification of available hardware, and often needs downtime to carry out the resize process. Horizontal scaling (scaling out) is an expansion of the number of instances in the backend pool - adding additional replicas of a container, instead of expanding the size of the individual container. The benefits of horizontal scaling include that it is architecturally more suitable to microservices because it can scale indefinitely, it provides natural redundancy, and it has a natural affinity to stateless and independently deployable nature of containers [9].

Auto-scaling goes even further by automating the scaling decision, i.e. it does not require human intervention. According to Lorigo-Botran et al., the auto-scaling techniques can be classified into five main categories: the technique of using a fixed threshold and applying the rule to the incoming traffic; the technique of using a fixed threshold and applying the rule to the incoming traffic; the techniques of control theory; the techniques of reinforcement learning; the techniques of queuing theory; the techniques of time series analysis [10]. In practice, auto-

scaling based upon a threshold is the most commonly used technique in cloud deployments, in which scaling operations are triggered when monitored metrics (CPU utilisation, request rate, memory consumption) cross defined thresholds. AWS achieves this by using EC2 Auto Scaling, which constantly checks the health and utilisation of the registered instances and adds or removes capacity in accordance with predefined scaling policies [9]. During periods of high demand on the Flask watchlist application, such as at peak evening times when users are browsing K-drama catalogues, Auto Scaling would be used to create more instances of the container and add them to the load balancer, making sure that the response time would not be affected. With the subsiding demand, termination of instances occurs to prevent unnecessary cost.

The connection between load balancing and auto-scaling is thus necessarily complementary as opposed to independently valuable. Auto-scaling, but no load balancing, yields a pool of instances that cannot be effectively accessed by the clients; load balancing, but no auto-scaling, yields a fixed-capacity system that cannot effectively respond to changes in demand. They would constitute a closed-loop system: the load balancer uses the traffic to distribute load across available instances, metrics based on this traffic are used by the auto-scaler, and the auto-scaler uses these metrics to adjust the number of instances registered with the load balancer. This dynamic resource management capability was one of the essential characteristics of cloud computing as compared to traditional hosted infrastructure [11].

2.3 Application Gateways

Although a Layer 4 load balancer is adequate to distribute traffic equally across the same instances of the backend, modern microservices architectures generally involve a number of distinct services, each with a different domain of functionality. The approach of routing all traffic via a single Layer 4 load balancer would result in all requests going to the same backend pool, irrespective of the nature of the request being made. Application Gateway overcomes this limitation by operating at Layer 7 of the OSI model the application layer that can inspect the entire contents of HTTP and HTTPS requests which include URL paths, headers and WebSocket connections before making routing decisions [12].

The implications in real life are important. Application gateway can look at the URL path of an incoming request, and redirect it to a suitable backend pool. A request to /catalogue could

be directed to the Flask watchlist service, a request to /auth might be directed to an authentication service, and a request to /recommendations might be directed to a separate recommendation engine all sharing the same public IP address, and being intelligently routed by the gateway and never needs any backend service to know about the others. This routing is based on URL paths, this routing feature is one of most important features of the Application Gateway as specialised backend pools, independent per-service scale and cost optimisation through proportional allocation of resources to each service is enabled by this feature of the Application Gateway [12].

AWS also implements this functionality using the Amazon API Gateway which serves as the entry point to microservices based on APIs, including request routing, authentication, rate limiting and protocol transformation [12]. In the case of a containerised Flask application at scale, all inbound traffic would be directed to the API Gateway which would in turn authenticate and validate requests against configured authorisers and forward validated requests to the corresponding ELB-backed container pool. It also does not couple the external API surface to the internal implementation the number of internal URLs and ports, and the number of container can be varied without affecting the external API endpoint called by clients.

In addition to routing, Application Gateways often include a Web Application Firewall (WAF), which checks incoming requests against a ruleset of known attack patterns - SQL injection attempts, cross-site scripting, malformed header - and blocks malicious traffic before reaching the backend [12]. This security layer has an architecturally significant role: it centralises the threat protection on the entry point of the system, and the individual microservices do not need to implement their own security filtering. The intelligent routing coupled with centralised security make the Application Gateway the choice of entry point of production web applications.

2.4 Combining All Three: A Unified Architecture

The above three technologies are even more effective when used in combination. As the full flow of traffic can show: the Application Gateway takes in and inspects the incoming requests at Layer 7, running WAF rules and path-based routing; the Load Balancer then forwards the verified and routed traffic to the backend instances at Layer 4; and Auto Scaling continues to adjust the number of those instances based on the real-time demand metrics [12]. All of these

components add a unique capability, security and intelligent routing, traffic distribution, and elastic capacity, which cannot be provided by the others alone.

In the case of a Flask-based microservices deployment on AWS, this would directly translate into a production-grade architecture. Amazon API Gateway is used to deal with all incoming traffic to the watchlist application, and to redirect requests to the corresponding service based on the URL path. The Application Load Balancer uses the requests to distribute them among pool of containerised Flask instances registered as ECS (Elastic Container Service) tasks. The EC2 Auto Scaling monitors CPU and request rate metrics and scales the container pool between a minimum and maximum number of instances based on demand [9]. The customer experience is exactly the same whether the system is serving ten concurrent users or ten thousand - which is the core value proposition of the cloud-native microservices architecture and the reason why organisations at a scale of Amazon itself have adopted this pattern to their own platform infrastructure [8], [9], [12].

3. Driving Forces Behind Cloud Expansion and Serverless Computing

3.1 Main Driving Forces Leading to Cloud Adoption

The last two decades of such growth in cloud computing has not been caused by something but rather by a combination of economic, operational, and strategic pressures that have made on-premises infrastructure even less justifiable. These forces are vital in putting these phenomena into perspective as to why technologies like Docker and containerisation have attained such a high rate and level of adoption.

The most commonly mentioned driver is cost efficiency. Conventional on-premise infrastructure involves capital expenditure in hardware that is sized to its peak load - although in normal operation most enterprise servers have only a utilisation of about 15 per cent. of the hardware they are powered by [13]. Cloud computing substitutes this model with pay-as-you-go consumption where organisations are charged only the amount of resources they actually consume. Model The shared infrastructure concept of cloud providers enables the efficiency of scale to be passed on to consumers through reduced per-unit pricing. The survey of 759 of the global IT decision-makers in the Flexera 2025 State of the Cloud Report found that cloud spend

management remained the most critical issue facing 84% of the global IT decision-makers surveyed in the Flexera 2025 State of the Cloud Report [14].

The second significant driving force is flexibility and agility. On-demand self-service, high-speed elasticity and access to the widest possible network are defined as the key features of the cloud model (NIST definition of cloud computing) [13]. These properties imply that an organisation can provision a new server in minutes, as opposed to weeks, scale its infrastructure up or down as the demand changes without capital commitment and access services at any location without depending on its physical location to hardware. In the case of a microservices based application such as the HanStream Flask watchlist service the latter translates directly into the capability to deploy updated container images on a global scale, scale individual services without the need to plan infrastructure upgrades, and respond to user demand without incurring infrastructure planning cycle.

The less commonly quantified, but critically important driver is innovation velocity. When not charged with the maintenance of more tangible infrastructure, engineering teams can redraw their efforts toward the capabilities that make their products different. The operational complexity of patching, hardware failure, network management and disaster recovery are absorbed by cloud providers, and developers are free to concentrate on application logic. This advantage is further enhanced by containerisation: since the application and its dependency are packaged into a portable image, Docker completely eliminates the category of environment-related failures that in the past consumed large amounts of engineering time - the so-called works on my machine problem that arises when local development environments do not match production [1].

3.2 Serverless Computing: Concept and Implementation.

Function-as-a-Service (or FaaS) is also known as serverless computing, and is an additional development of the cloud execution model. In serverless architecture, developers code discrete functions, i.e., small, single-purpose units of code, and deploy them to a cloud platform that manages all aspects of execution infrastructure: provisioning, scaling, load balancing and billing. Serverless is a misleading architectural term that servers still exist and are managed, but it accurately describes the developer experience, where servers are completely abstracted away [15].

The most popular serverless platform is AWS Lambda. An event, such as an HTTP request through API Gateway, an upload to S3, a message on an SQS queue, a scheduled timer - and an executed Lambda function is triggered by an event - and executed in a managed runtime environment that is on-demand provisioned and post-execution torn down [15]. The equivalent provided by Microsoft is Azure Functions that works based on the same principles and supports a wide variety of trigger types and programming languages such as Python, JavaScript, C#, and Java [16]. Both platforms charge at the granularity of milliseconds depending on the number of functions calls and the time it takes to execute the functions, so that a function that runs infrequently will only incur almost zero cost during idle times.

In the widely-read Berkeley View on Serverless Computing, Jonas et al. describe the serverless model as a major simplification of cloud programming - one that compares to the historical transition of assembly language to high-level programming languages in the degree of its success [15]. Their analysis states that automatic scaling, elimination of operational burden, and granular billing are the key benefits, and some current limitations exist in relation to the execution duration, latency of cold start and appropriateness with long-running or stateful workloads.

3.3 Containers vs. Serverless: A Critical Comparison.

The notions of serverless and containerisation are often introduced as opposing technologies, yet a more accurate key point analysis reveals that the two are complementary technologies, which are applicable to various problem profiles. This difference can be conceptualized on four dimensions: control, statefulness, execution model and cost profile.

The comparison across four key dimensions summarised in Table 2 below with direct reference to the HanStream Flask application developed in this project [3], [14], [15], [16].

Table 2: Comparative study of Docker containerisation and serverless computing on four dimensions

Dimension	Containers (Docker / v2-flask)	Serverless (AWS Lambda / Azure Functions)
Portability & Control	Fully portable across any Docker-compatible host. The same	Tightly coupled to a specific provider's runtime and services.

	azizulkhaan/v2-flask image runs on AWS ECS, Azure Container Instances, or a local machine without modification [3].	Migrating between Lambda and Azure Functions requires meaningful re-engineering [15].
Cost Model	Continuous cost whether handling requests or idle. More economical for consistently high-throughput workloads where per-request serverless pricing exceeds container cost [14].	Billed per invocation at millisecond granularity. Near-zero cost during idle periods; advantageous for infrequently called or low-traffic applications [15].
Statefulness	Supports stateful workloads via Docker volumes. The Flask app persists watchlist data in a JSON file on v2-vol, surviving container restarts [3].	Inherently stateless; each invocation starts with a clean context. Lambda max execution is 15 minutes. Unsuitable for session-based or long-running applications [15].
Dev Complexity	Requires Dockerfile authoring, image build/push, volume and network configuration. Higher operational overhead, but grants full control over the runtime environment [4].	Minimal setup for narrowly scoped tasks. No Dockerfile, no network config. Provider manages all infrastructure. Ideal for event-driven triggers such as file uploads or scheduled jobs [16].

It is not then that one model is better than the other but that the decision must be made to be governed by the nature of the workload. Serverless is most appropriate to a stateless, event-driven, infrequently invoked functions; containers are the right choice to a persistent, stateful, or continuously available services like the HanStream application. In practice, full-fledged cloud-native architectures often blend the two, with core application services provided by containers, and peripheral tasks handled by serverless functions [15], [16].

Conclusion

This report has shown that the Docker containerisation is not simply a convenience of deployment but is a fundamental architectural technology that makes possible the microservices design patterns, scale of operation and portability of deployment that define modern cloud-native applications. This report has demonstrated the entire lifecycle of a containerised microservice, including Dockerfile, running container, publicly accessible registry image.

This analysis of load balancing, auto-scaling, and application gateways in Section 2 has shown that the three technologies are mutually reliant and that each of the three technologies addresses a specific weakness of the others. Load balancing spreads demand, auto-scaling adjusts capacity, and the application gateway enforces intelligent routing and security - all of which provide the availability, resilience and scalability that are promised by cloud platforms. The realisation of these capabilities in the Elastic Load Balancer, EC2 Auto Scaling, and API Gateway are a good example of how this architecture is realised at production scale.

The analysis of driving forces in Section 3 revealed that ultimately cloud adoption is determined by the economic logic of the pay-as-you-go model, elimination of idle capacity costs, and redirection of engineering energy towards innovation instead of infrastructure maintenance. The analogy to serverless computing was that both models serve the same underlying drivers but in contexts of stateless, event-driven, intermittently invoked functions and containers, respectively. Neither a binary decision on which of the two paradigms to choose but a mature interpretation of when each is suitable, is the conclusion, and this is the most effective that the cloud engineering discipline can be.

References

- [1] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, 2014. [Online]. Available: <https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment>
- [2] C. Pahl, "Containerization and the PaaS cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24-31, May 2015. DOI: 10.1109/MCC.2015.51
- [3] D. Bernstein, "Containers and cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81-84, 2014. DOI: 10.1109/MCC.2014.51
- [4] Docker Inc., "Dockerfile reference," Docker Documentation, 2024. [Online]. Available: <https://docs.docker.com/reference/dockerfile/>
- [5] Docker Inc., "Docker Hub," Docker Documentation, 2024. [Online]. Available: <https://hub.docker.com>
- [6] Docker Inc., "Manage data in Docker," Docker Documentation, 2024. [Online]. Available: <https://docs.docker.com/engine/storage/volumes/>
- [7] Docker Inc., "Networking overview," Docker Documentation, 2024. [Online]. Available: <https://docs.docker.com/engine/network/>
- [8] Amazon Web Services, "What is an Application Load Balancer?" AWS Documentation, 2024. [Online]. Available: <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html>
- [9] Amazon Web Services, "What is Amazon EC2 Auto Scaling?" AWS Documentation, 2024. [Online]. Available: <https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html>
- [10] T. Llorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559-592, Dec. 2014. DOI: 10.1007/s10723-014-9314-7
- [11] R. Buyya et al., "A manifesto for future generation cloud computing: Research directions for the next decade," *ACM Computing Surveys*, vol. 51, no. 5, Art. 105, Nov. 2018. DOI: 10.1145/3241737

- [12] Amazon Web Services, "What is Amazon API Gateway?" AWS Documentation, 2024. [Online]. Available: <https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html>
- [13] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Special Publication 800-145, 2011. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>
- [14] Flexera, "2025 State of the Cloud Report," Flexera, 2025. [Online]. Available: <https://resources.flexera.com/web/pdf/Flexera-State-of-the-Cloud-Report-2025.pdf>
- [15] E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," UC Berkeley Technical Report, arXiv:1902.03383, Feb. 2019. [Online]. Available: <https://arxiv.org/abs/1902.03383>
- [16] Microsoft, "Introduction to Azure Functions," Microsoft Learn, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview>