HyperionDev

# Managed and Unmanaged Services

## Task

Visit our website

# Introduction

In this task, you'll explore the differences between on-premise infrastructure and cloud services, along with various cloud service models like infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS). You'll also learn about the benefits of managed services compared to unmanaged solutions, considering factors such as scalability, availability, and maintainability when designing cloud architectures.

By the end of this task, you'll be able to choose the right solutions for specific use cases and understand how to design cloud-based systems for optimal performance.

# Introduction to on-premises

## Data centres

Data centres are essential for modern businesses, providing a central hub for computing, storage, and application deployment. They enable efficient collaboration and allow developers to build and deploy applications without local installations. While cloud computing has reduced the need for in-house infrastructure, many organisations still maintain on-premise data centres for greater control.

## Running on-premises

Operating an on-premises data centre requires significant investment and resource allocation. Organisations must invest in physical locations, hardware, software, and personnel to manage these various components. To understand the responsibilities involved in running an on-premises data centre, it's helpful to examine the different parts of the infrastructure that need to be managed.

The infrastructure refers to the resources that comprise a data centre. It can be divided into two main components: **physical** infrastructure and **logical** infrastructure. Each layer of the infrastructure demands a distinct set of skills for effective management.

### Physical infrastructure

The physical infrastructure of a data centre refers to the physical hardware components that make up the data centre. When discussing physical infrastructure, we primarily focus on three key elements: compute, storage, and networking.

**Compute** refers to the servers that provide the computing resources necessary to execute various workloads. Servers typically consist of a central processing unit (CPU) and memory (RAM), along with other components as needed. The choice of server

depends on the specific workloads to be performed, as different servers have varying component configurations. For high-performance operations like statistical analysis or data processing, servers with more memory and a higher core count are beneficial. Visual processing or machine learning tasks may benefit from servers equipped with graphics processing units (GPUs) for accelerated performance.

**Storage** refers to the physical or virtual drives used to persistently save and manage various types of data generated by systems. Various storage drives offer different levels of performance for read and write operations. Faster storage options, such as solid-state drives (SSDs), are generally more expensive per gigabyte. To optimise costs while maintaining high performance, a combination of fast SSD storage and larger amounts of slower hard disk drives (HDDs) is often employed.

**Networking** hardware is responsible for facilitating communication between the different hardware and devices within the data centre. A typical data centre consists of multiple individual servers and numerous storage devices. Networking hardware enables these devices to be physically connected and work together. The software layer on top of the networking hardware allows these components to be logically connected, providing a single access point for their use.

## Logical infrastructure

The physical infrastructure itself does not directly provide business value. To effectively utilise the resources and handle various workloads, we need tools that can interact with both hardware and software. These tools are organised in layers, each offering a different level of abstraction between the user-facing tools and the underlying hardware.

**Virtualisation (optional):** This layer creates a separation between physical hardware and other logical infrastructure. It allows us to run multiple virtual machines on a single server, maximising resource utilisation. We can specify resource allocation based on overall availability, enabling efficient resource usage and isolation between virtual machines.

**Operating systems:** The operating system acts as an interface between hardware and software. It facilitates workload distribution, the creation of file systems, and communication with storage devices. Every computer system relies on an operating system to avoid the complexities of directly interacting with hardware, and choosing an operating system depends on specific needs. Large organisations often opt for enterprise solutions like RedHat Enterprise Linux or Microsoft Server, while open-source options like Ubuntu Server or CentOS are also popular.

**Middleware:** Different applications running on a single operating system might use incompatible runtimes, hindering communication. Middleware bridges this gap, allowing services using different runtimes to seamlessly interact. Common middleware

includes web servers for HTTP communication, message-oriented middleware for asynchronous communication, and database middleware like Open Database Connectivity (ODBC) for connecting to SQL databases in applications.

**Runtimes:** Application code execution requires specific resources, libraries, and environments. Runtimes provide these elements, enabling code execution on the server. Not all applications require a runtime, but most applications built with languages like Python or JavaScript rely on runtimes. Some popular examples include .NET (C#, F#, VB.NET), JRE (Java, Scala, Kotlin), Node.js (JavaScript), and Python runtime.

**Applications and data:** The logical infrastructure ultimately supports user applications and data storage. Applications run on their specific runtimes and interact with middleware to access databases. Data can be stored in database engines, file systems, or object storage.

An on-premise data centre requires the organisation to manage all these tools, ensuring proper installation, version control, security updates, alignment with business needs, licence management (if applicable), and data backups for disaster recovery.

# Introduction to the cloud

Setting up and managing a data centre involves significant complexities and costs, making it a daunting prospect for many businesses. Fortunately, cloud providers like AWS, Microsoft Azure, and Google Cloud Platform offer a viable alternative. These companies specialise in operating large-scale data centres and providing their resources to customers on a rental basis.

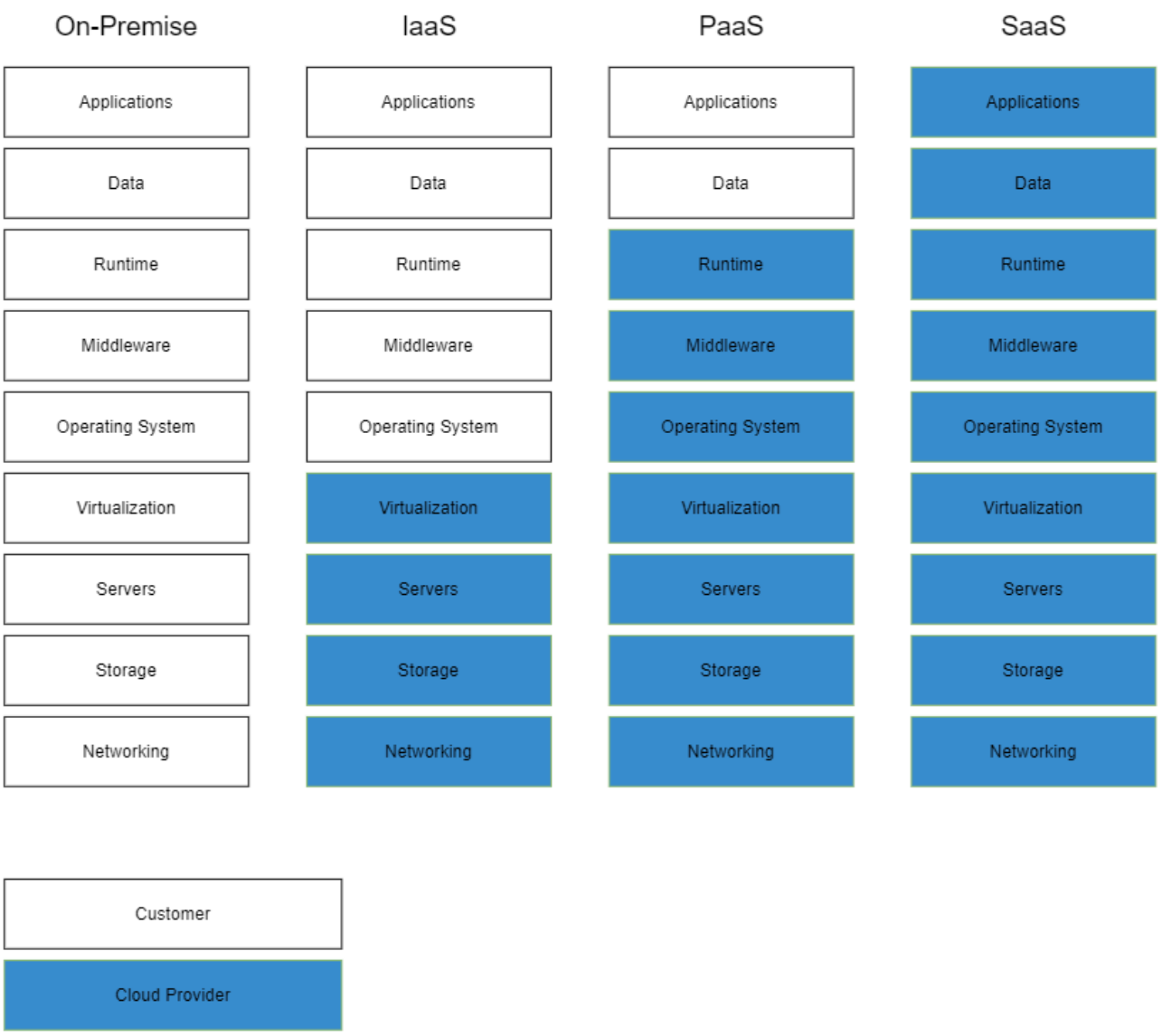
Cloud providers can invest heavily in maintaining their servers at the highest standards of performance and reliability. This enables businesses to leverage cutting-edge infrastructure without the need for substantial upfront investments or ongoing maintenance.

## Cloud service models

Cloud providers offer different levels of abstraction, known as cloud service models, to make cloud computing accessible to diverse users. By abstracting the underlying physical infrastructure, these models allow individuals and organisations with varying technical expertise to work effectively with cloud services, focusing on their operational priorities.

## Extra resource

Take a look at the following resources by IBM that talk about the **cloud service models**. Here are some videos that also explain the concepts: **IaaS**, **PaaS** and **SaaS**.




This graphic shows who manages what in **On-Premise**, **IaaS**, **PaaS**, and **SaaS** models—ranging from full customer control in On-Premise to complete provider management in SaaS.

Typically, there are three main levels of abstraction offered within these cloud service models.

### Infrastructure as a service (IaaS)

IaaS provides a flexible solution by offering virtual or bare-metal servers via the cloud, accessible online through protocols like SSH. This allows organisations to manage their logical infrastructure without investing in physical hardware, making it ideal for scaling capacity. Virtualisation underpins IaaS, enabling the creation of multiple virtual servers on a single physical machine. However, expertise in server management is essential to fully leverage its benefits.

### Platform as a service (PaaS)

PaaS builds on IaaS by adding an abstraction layer, offering pre-configured environments for services like databases and runtimes. This lets developers focus on applications without managing the underlying infrastructure. Virtualisation plays a role here too, as it simplifies scaling and resource allocation. By speeding up development and time-to-market, PaaS helps developers concentrate on product improvement rather than infrastructure handling.

### Software as a service (SaaS)

SaaS delivers fully functional applications over the Internet, removing the need for on-premise installation. The vendor manages infrastructure, updates, and maintenance, providing a seamless experience for customers. Built on virtualised infrastructure, SaaS enables users to access applications from various devices and collaborate without the need for hardware investments. Subscriptions are common, making it a cost-effective and accessible solution.

# Managed and unmanaged services

Cloud providers offer two main types of services: managed and unmanaged. Managed services require less customer intervention, as the provider handles most operational tasks, while unmanaged services leave these tasks to the customer. Though managed services often align with PaaS, not all PaaS solutions are managed. For example, AWS offers both managed and unmanaged options for its Relational Database Service (RDS).

## Choosing a solution

When choosing between managed and unmanaged solutions, it's important to weigh the higher cost of managed services against the effort required to maintain cheaper, unmanaged options. Small teams or individual developers may prefer managed

services to focus on development, offloading infrastructure tasks. Larger teams, with more resources, might benefit from unmanaged services, saving costs while managing infrastructure in-house for greater control.

When it comes to managing services, the following aspects can be taken into consideration:

- **Scalability:** How easily can the service scale to meet your changing needs?
- **Availability:** What level of uptime and reliability does the service offer?
- **Portability:** Can you easily migrate your data and applications to another provider?
- **Maintainability:** How complex is it to manage and maintain the service?

By carefully evaluating these factors, you can select the type of service that best aligns with your business requirements and goals.

## Scalability

Scalability is a crucial aspect of cloud computing, ensuring that systems can handle increasing workloads without affecting performance. It refers to the ability of a system to expand its resources to accommodate higher demand. This ensures that all users have a consistent experience, even during peak usage periods.

There are two primary approaches to scaling: vertical and horizontal. **Vertical scaling (scaling up)** involves adding more computational power to a single instance of a service. This method is best suited for centralised systems like relational databases, or monolithic applications. **Horizontal scaling (scaling out)** involves increasing the number of instances of a service. This approach is ideal for applications designed with a distributed architecture, where individual instances can perform tasks independently.

AWS includes features for auto-scaling when configuring managed services, and it can also be enabled for unmanaged services. However, for unmanaged services, auto-scaling cannot be set up during initial configuration; the customer must manually configure the virtual networks and configure additional services to allow for auto-scaling.

## Availability

Availability is a crucial aspect of cloud services, measuring how often a service is operational and accessible within its expected working windows. Many enterprise-level applications offer service-level agreements (SLAs) that specify availability as a percentage, such as 99.99%. It is the responsibility of the system architect to ensure that their services meet these SLAs.

To achieve high availability, services often employ redundancy, which involves having multiple instances of the same service. There are two main types of redundancy:

- **Local redundancy:** Different instances of a service are created within the same data centre on separate physical servers.
- **Zone/geographical redundancy:** Instances are distributed across different servers in various geographical locations (availability zones).

Redundancy serves several purposes, including failover and data protection. In the case of failover, if one instance becomes unavailable, the system can automatically switch to another, ensuring continuity. For data protection, particularly in databases, maintaining multiple copies of data on different servers safeguards against data loss and enhances reliability.

Managed services typically require users to select a configuration option for availability. In contrast, unmanaged services require manual setup of additional instances and any other operations needed to keep the services in sync.

## Portability

Portability refers to a service's ability to operate seamlessly across different environments with minimal adjustments. For example, cloud providers often offer cloud-specific tools that may lead to vendor lock-in – a situation where it becomes challenging to move to a different provider due to reliance on proprietary features. Alternatively, more portable options, such as open-source tools or unmanaged services like virtual machines, enable configurations to be easily transferred between providers. Technologies like containers further enhance portability by allowing the same application **image** to run across multiple platforms with minimal setup.

## Maintainability

Maintainability refers to how easily a service can stay operational. Cloud providers like AWS offer tools like **CloudWatch** for system monitoring and **Simple Notification Service (SNS)** for event-based notifications to ensure system visibility and quick issue response.

In app development, maintainability also includes ease of deployment. Managed services often integrate seamlessly with tools like GitHub, while unmanaged services may require manual uploads or the creation of custom pipelines to streamline the process. Focusing on deployment automation and integration enhances the reliability and efficiency of cloud services.

# Designing a cloud architecture

Modern applications often consist of multiple interconnected services working together to perform specific tasks. Each service must be deployed independently and capable of effective communication with other services to ensure seamless application functionality. Additionally, the entire system should be scalable and highly available.

Architecture diagrams provide a valuable visual representation of the various services within a system. By examining these diagrams, we can easily identify direct communication paths between services and assess scalability potential. This visual perspective facilitates optimisation for cost-effectiveness and simplifies the implementation process.

# Case Study 1: Identity Card Generator

Let's explore a practical approach to designing the architecture of an application using the managed services offered by AWS. We will assess the system requirements, evaluate how the system will be used, and identify the best combination of AWS services to meet these needs.

Additionally, we will consider the running costs of the solutions we devise. The outcome of this exercise will include an architecture diagram, which can be used by cloud engineers to implement and maintain the solution, as well as a cost summary. This summary will assist solutions architects and management in planning and optimising costs for the business.

## Requirements

We will be designing the architecture of an electronic student ID processing service. The service is designed to integrate with a larger student management application.

Our service will need to allow for the following operations:

- Students should be able to generate student IDs based on the information stored on the main system as well as a photograph of themselves.
- Students should be able to request student IDs at any time allowing their cards to reflect the most up-to-date personal information.
- Students should be able to download the latest version of their student ID as a PDF.
- Generated student IDs should include QR codes that will be used to enter specific facilities on a university campus.

## System usage

There is no single correct approach to solving an architectural problem, even when dealing with identical requirements. When designing a cloud solution, it's essential to understand the context of how the application will be used. This ensures that services are selected not only based on their ability to meet technical requirements but also their alignment with business needs.

The following must be kept in mind:

- The system will be used by just under 3 million students.
- The first two months of the service will see all of the students creating their student IDs.
- It's expected that there will be around 90 000 requests per day to generate student IDs in the first two months of the academic year, also factoring in students making changes to their details and requiring new ID cards.
- On average, students will scan the QR code on their student IDs about five times a day to access their campuses and certain facilities.

## Designing the system

To design the architecture of our system, we can follow a few steps. These include understanding the types of services required to solve the problem, evaluating the available services on the cloud provider (AWS in this instance), and choosing the services that best address our needs before creating the final architectural design.

### Step 1: Identify the required services

For the first step, we're going to examine the requirements and determine what is needed to meet each one. This will give us a clear picture of the types of services we might want to leverage.

Following this step will provide clarity on what you're looking for in a cloud service to solve your problem. This understanding allows you to explore different cloud providers and identify the one that offers the best set of services to solve your problem in an easy and affordable way.

Based on the requirements, we know that we will need the following services:

- **Compute:** Running the operations for generating student IDs and processing QR code scans.
- **Networking:** Allowing the service to communicate with the main application.

- **Database:** Storing information about students and which facilities they are able to access.
- **Object storage:** Storing the generated student IDs.

### Step 2: Exploring cloud services

Once we know the types of services we need to solve our problem, it's important to consider all the available options and determine whether they offer the flexibility we require.

At this point, we can evaluate the managed and unmanaged aspects of each service to ascertain which best suits our needs.

When comparing different services, we should examine various aspects and consider whether the cloud provider will manage these services. Once we understand what each service offers and who is responsible for managing it, we can make a more informed decision.

- **Scalable:** The service can make use of auto-scaling.
- **Available:** The service is designed for high availability.
- **Maintainable:** The service easily connects to development tools.
- **Portable:** Resources created around the service can be easily transferred to another cloud provider.

| Name | Purpose | Auto-scaling | Available | Maintainable | Portable |
|---|---|---|---|---|---|
| **Lambda** | Serverless computing for code | ☑ | ☑ | ☐ | ☐ |
| **S3** | Object storage | ☑ | ☑ | ☐ | ☐ |
| **DynamoDB** | Serverless NoSQL database | ☑ | ☑ | ☐ | ☐ |
| **EC2** | Virtual machine | ☐ | ☐ | ☐ | ☑ |
| **DocumentDB** | Database based on MongoDB | ☐ | ☐ | ☐ | ☑ |
| **Elastic Beanstalk** | App service | ☑ | ☑ | ☐ | ☑ |

## Extra resource

AWS, like other cloud providers, offers a whole range of services and products. When developing cloud services, it's worth going over the different offerings and having a basic understanding of what they do and the problems they can solve. Take a look **Amazon's cloud services**.

---

### Step 3: Choosing the right service

When building cloud solutions, context should drive development. Success with one approach for a team doesn't guarantee it will work for others, as applications vary in user patterns, business needs, and volumes, which are all crucial in selecting the right services.

For our system, we know that it will experience different volumes of traffic at specific times of the year. The student ID generation operation will peak in the first two months, while QR code scanning will experience consistent usage throughout the school year.

Building a single API would be sufficient; we could allocate more resources at the beginning of the year to accommodate the higher demand for the ID generation feature and then scale down for the rest of the year.

Although we can ensure the availability of the solution as we know we'll never exceed 3 million concurrent users, there may be issues with edge cases, such as a university rebranding and requiring new designs for student IDs mid-year.

We will likely need more resources than we'll use, as it's improbable that over 60% of users will be on campus, leading to underused QR code scanning. While we could allocate fewer resources, this risks being unprepared for unexpected events. Ensuring availability is essential to avoid students losing access to facilities due to service failures.

For our system, we can make use of the following services:

**Lambda**

AWS Lambda is a function as a service (FaaS) offering that lets you upload code to AWS and execute operations via HTTP requests. It abstracts complexities like server setup and resource allocation, allowing you to focus on writing code. Lambda dynamically scales based on demand, using zero resources until invoked, making it cost-effective for unpredictable workloads. Each function scales independently, ensuring high performance without manual resource management. For our application, we can split

key operations into separate Lambda functions, enabling automatic scaling and efficient resource use.

### Simple Storage Service (S3)

Amazon S3 (Simple Storage Service) is AWS's object storage solution designed to store a wide variety of data types. Unlike traditional database systems like SQL or NoSQL databases, which are used for structured data, object storage solutions like S3 are ideal for handling unstructured data.

Unstructured data refers to information that doesn't follow a fixed schema, such as images, videos, audio files, text documents, and PDFs. This type of data cannot be directly written to a traditional database and is therefore stored in object storage systems like S3, where it is managed as individual objects.

For our application, we can use S3 to store the PDFs that will be generated when a student ID is created.

### DynamoDB

DynamoDB is a serverless NoSQL database from AWS that scales dynamically based on usage. It's suitable for our solution because it doesn't require many data tables and supports **sharding** for better query performance.

### DocumentDB

AWS offers DocumentDB, a database service compatible with MongoDB, designed to simplify migration between cloud providers. It integrates well with tools like Mongoose for JavaScript applications, making development and migration more straightforward. However, DocumentDB can be more expensive and less flexible compared to other options, as it requires manual scaling management.

### API Gateway

When building a solution with several serverless functions, it is crucial to simplify access to each function. Rather than having the client-side application reference multiple locations directly, an API Gateway can be used to centralise these references. The API Gateway acts as a single entry point, storing references to each function and providing abstracted endpoints that the client application can easily call.

An API Gateway is similar to a router in an Express-based API. You define the endpoints and methods, then link them to the appropriate controllers. In the case of serverless solutions, the controller would be the cloud function for a specified operation.

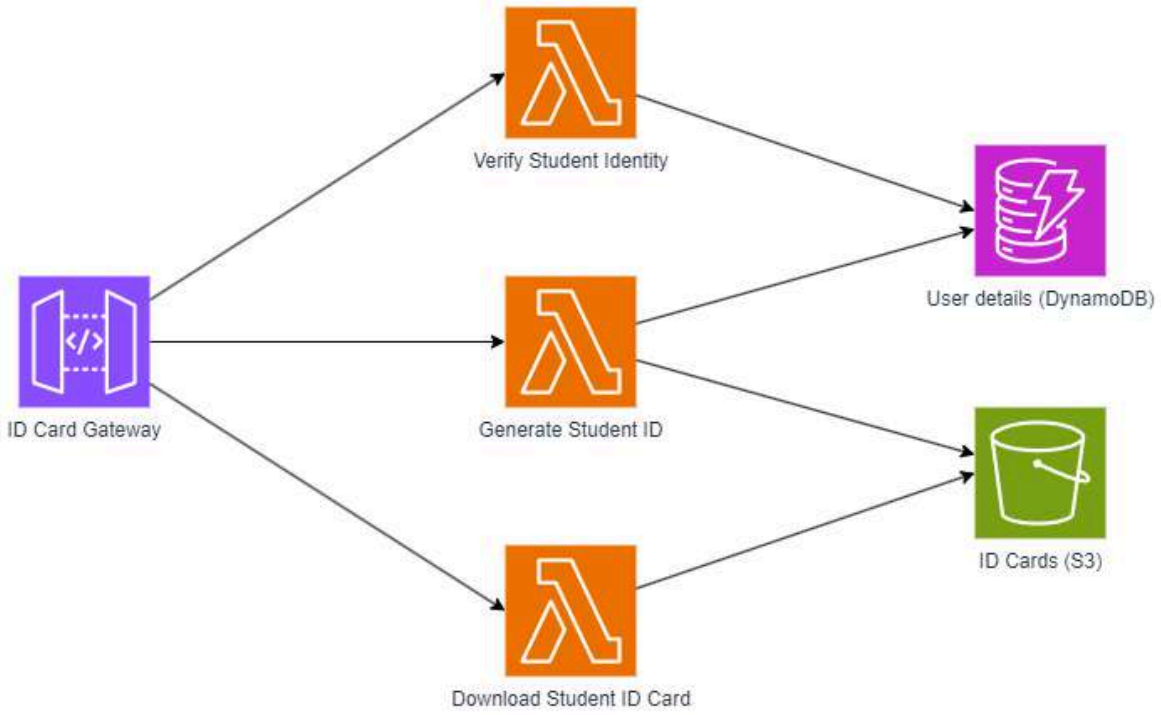
API Gateways are not limited to serverless functions; they can be used for any code that interacts with HTTP protocols. They are commonly employed to connect various

services and add an additional layer of security, making them a versatile tool in modern cloud architectures.

### Step 4: Creating the architecture design

Below is the architecture diagram for the solution. It's important to note that more sophisticated architecture diagrams would include additional elements such as backups, monitoring, and performance optimisations like caching. However, the diagram below presents a simple and straightforward approach that can be used to address this problem.

The diagram illustrates the relationships between the various services. The API Gateway serves as the main entry point for any client application and connects to the three Lambda functions. The "Verify Student Identity" and "Generate Student ID" functions both interact with the DynamoDB service, as they require access to user data. Additionally, the "Generate Student ID" and "Download ID Card" functions connect to the S3 bucket to manage the student ID PDFs.




*Architecture diagram for an ID card generator*

## Cost calculations

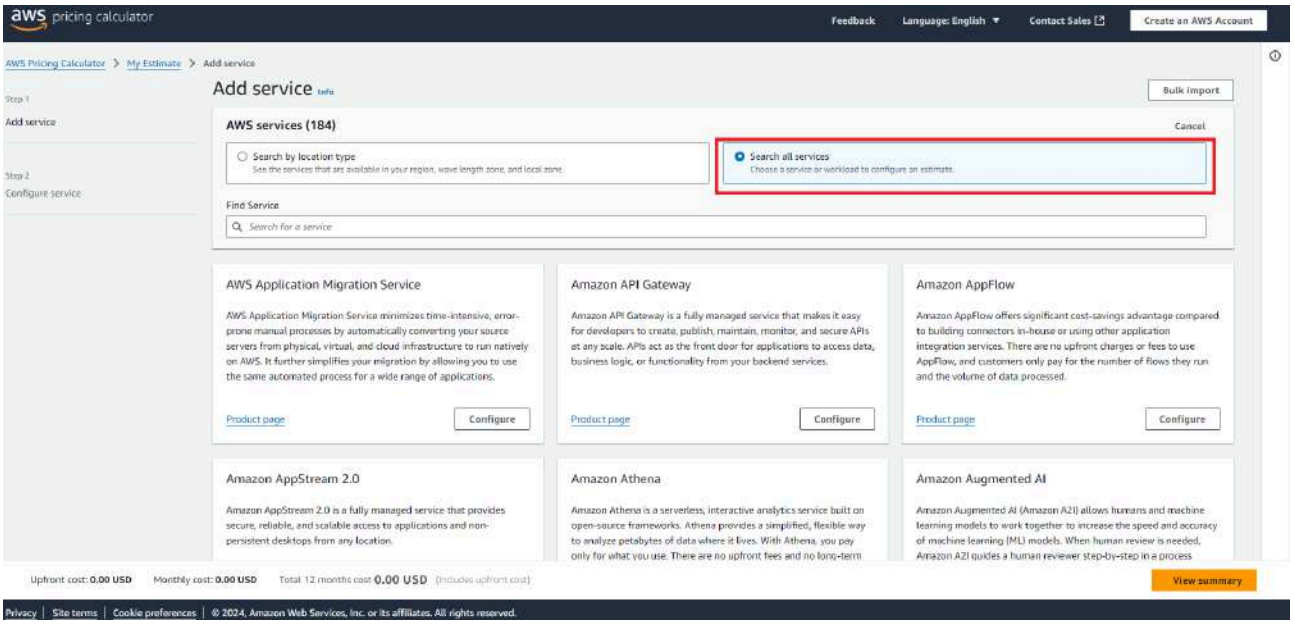
When choosing solutions it's important to consider the cost of each service being used and how much it might increase based on the expected load. Even after making these estimates, it is essential to have a final overall estimate to understand the potential monthly costs of all services.

AWS provides a pricing calculator that enables us to estimate the monthly running costs of the different services we intend to use.

## Step 1: Add service page

Go to the **AWS Pricing Calculator** and make sure the “Search all services” option is selected.

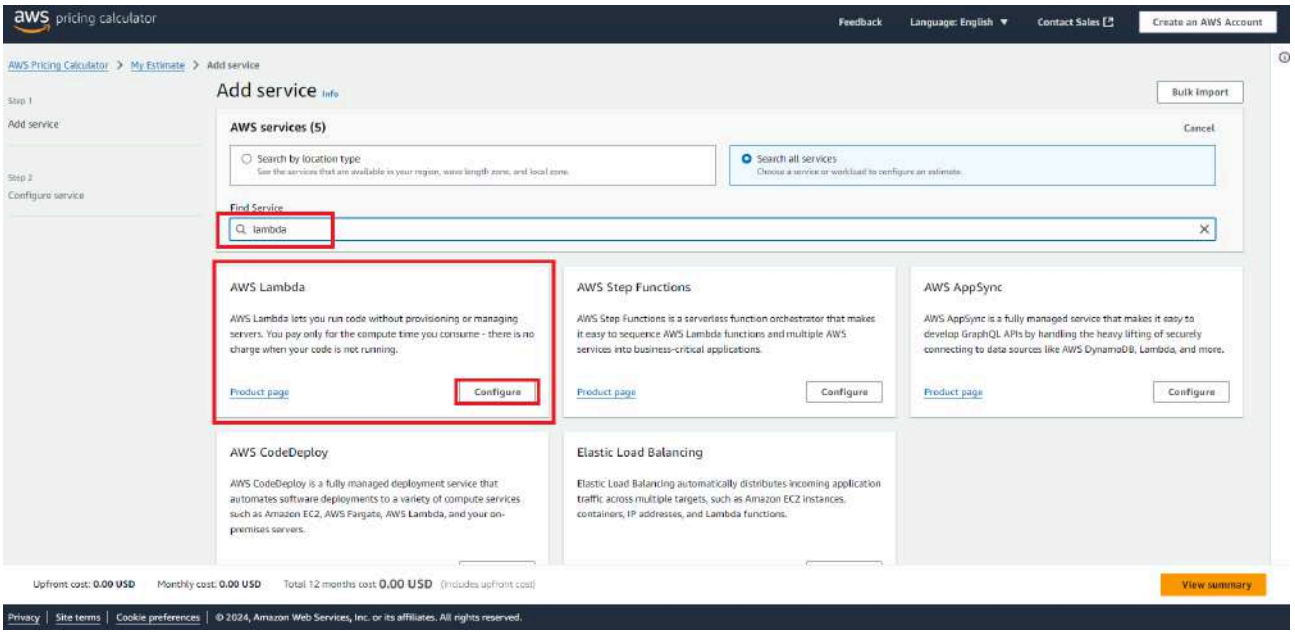
On this page, we can view all the available AWS services, along with a brief description of the purpose of each service. Once we’ve selected our services, we can then view a summary of the total costs and see which services contribute to those costs.




## Step 2: Finding a service

To find a service, we can type the service’s name into the search bar to find any services matching the search term. Alternatively, we can scroll through the various offerings until we locate a specific service.

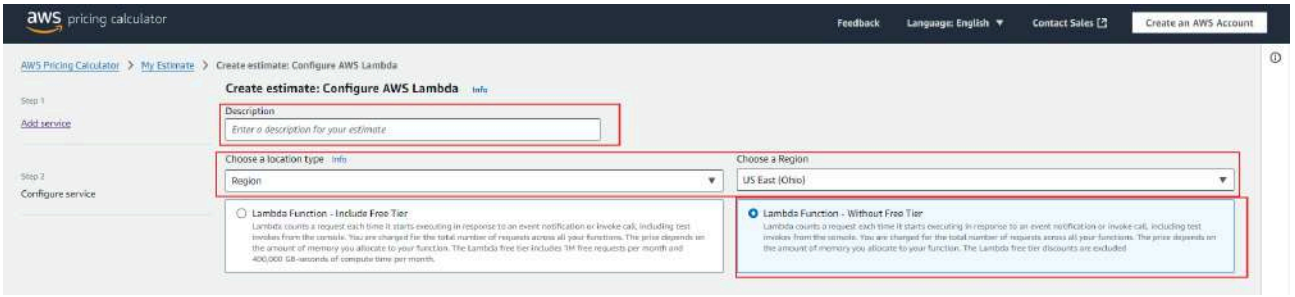
To add a service to the cost calculation, click on the “Configure” button for the chosen service, and you will be taken to the price calculation page.




## Step 3: Configuring a service

On the configuration page, the first inputs are for “Description” and “Region”. Adding a description clarifies the service’s purpose, especially when using multiple instances of the same type. The “Region” indicates where the service will be hosted; some regions may have higher costs or limited availability. For optimal performance, choose the data centre closest to users.

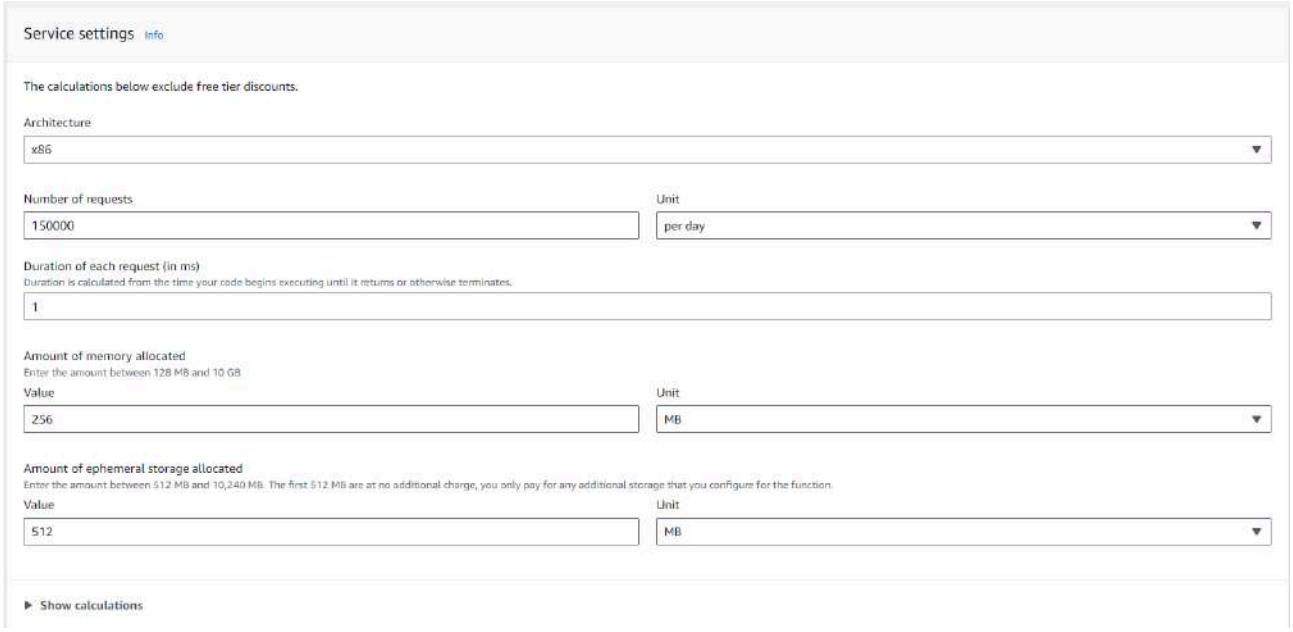
You may also see an option to include or exclude the free tier in cost estimates. It’s best to exclude it, as free tier benefits apply per account, not per service, and could already be in use by another service on the account.




## Step 4: Adding estimated usage

Different services have unique input sections based on their usage. For instance, a service like Lambda tracks calls and duration, while a storage service like S3 focuses on data size. Enter the total expected usage; for example, estimating traffic with 100 users making 10 calls daily would involve multiplying users by daily calls and then by days in a month for the monthly usage.

If you have usage data, using averages helps avoid cost overestimates or underestimates. Also, account for seasonal variations, as some industries experience higher demand at specific times, which may require separate calculations for those periods.

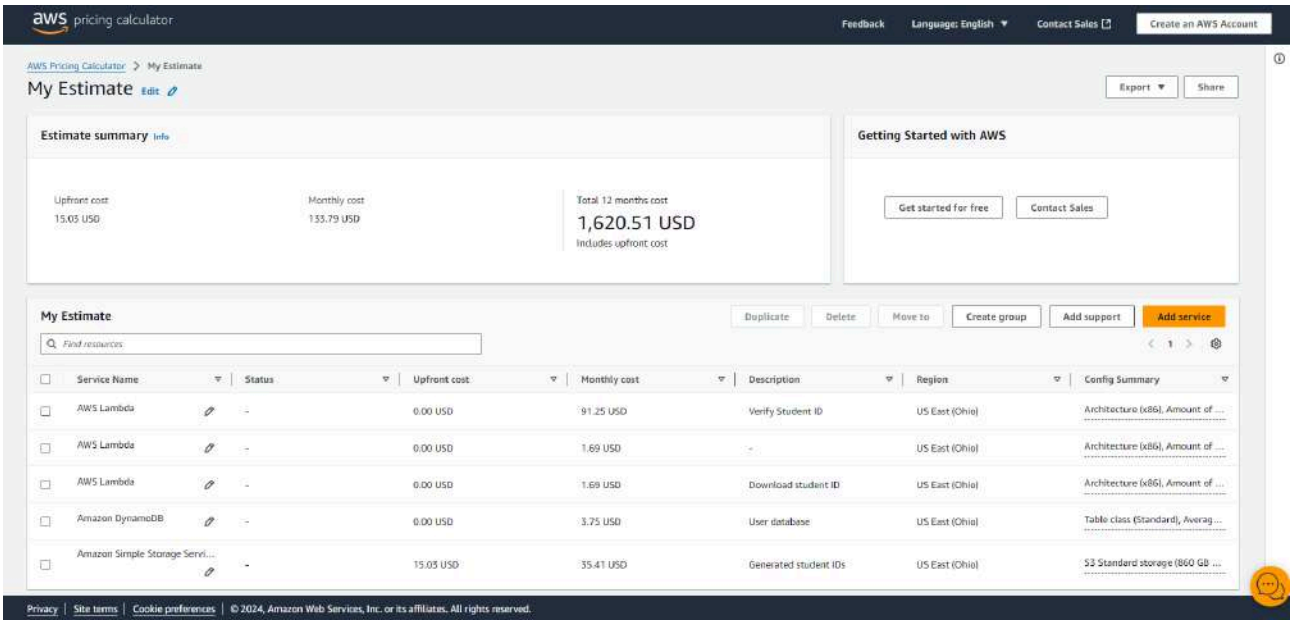



## Step 5: Saving the estimates

Once the information has been added, you can save the calculations for the service. You will then be taken to the services page, where you can either select a new service or view the full summary:


## Step 6: Cost summary

Once all the services have been selected and the costs calculated, you can create a summary of the costs. From the main page, there will be an option to view the summary. Clicking this option will take you to a page displaying all the selected services. From here, you can add more services, delete existing services, or update current ones.




### Step 7: Print report

To have a persistent copy of the services and their costs, you can click the download option at the top of the screen and choose to download the costs as a CSV, JSON, or PDF.

## Values to enter per service

The values used for each of the services can be found below. Keep in mind that these values are based on the total expected usage. For instance, a service like ID verification, which anticipates five calls per student, will result in a total of 15 000 000 calls, given that there are 3 000 000 students using this service. A similar approach was taken for storage, where the total storage used was calculated based on the types of operations performed.

**Verify Student ID (Lambda)**

| Property Name | Value | Reasoning |
|---|---|---|
| Architecture | x86 | Common architecture |
| Amount of ephemeral storage allocated | 512 MB | Default |
| Invoke mode | Buffered | Default |
| Number of requests | 15 000 000 per day | Number of students x requests per student |

**Generate Student ID Card (Lambda)**

| Property Name | Value | Reasoning |
|---|---|---|
| Architecture | x86 | Common architecture |
| Amount of ephemeral storage allocated | 512 MB | Default |
| Invoke mode | Buffered | Default |
| Number of requests | 90 000 per day | Provided in the usage stats |

**Download Student ID Card (Lambda)**

| Property Name | Value | Reasoning |
|---|---|---|
| Architecture | x86 | Common architecture |
| Amount of ephemeral storage allocated | 512 MB | Default |
| Invoke mode | Buffered | Default |
| Number of requests | 90 000 per day | Assumes that a student will download the ID once every time they generate a new one |

**User Database (DynamoDB)**

| Property Name | Value | Reason |
|---|---|---|
| Average item size (all attributes) | 1 KB | Text data is very small; basic user information will be a few bytes |
| Data storage size | 15 GB | The total number of students x the estimated size of their information |
| Choose DynamoDB features | DynamoDB provisioned capacity | Based on the expected number of students, we can get pre-allocated storage |
| Table class | Standard | |

**Generated Card (S3)**

| Property Name | Value | Reasoning |
|---|---|---|
| S3 Standard storage | 860 GB per month | Number of students x size of the final ID rounded off |
| S3 Standard Average Object Size | 300 KB | Provided in requirements |

| | | |
|---|---|---|
| Data returned by S3 Select | 799 GB per month | Based on the calls made by the download student ID function |
| GET, SELECT, and all other requests from S3 Standard | 2790000 | Based on the calls made by the download student ID function |
| PUT, COPY, POST, LIST requests to S3 Standard | 2790000 | Based on the calls made by the generated student ID function |

# Case Study 2: Deploy to S3

S3 is an object storage solution for files like images, audio, and video, accessible over the Internet when configured properly. Static websites, composed of HTML, JavaScript, and CSS can be deployed through S3.

For this example, we'll use **LocalStack** (a tool that lets developers run and test AWS services locally, enabling faster development and cost-free offline testing) to emulate AWS locally. You will need to ensure that Docker is installed in order to make use of LocalStack.

## Installing Docker

You can refer to the "**Additional Reading – Installation, Sharing, and Collaboration**" guide to complete the Docker installation. The guide will walk you through everything you need.

Before following the steps below, create a new React project or use an existing one. All subsequent steps will be carried out within the React project folder.

### Step 1: Building a React Project

When running a React project locally, you're probably accustomed to using the `npm run dev` command. This command is used to start a server that runs the application on localhost.

To turn your React project into a static website, you can run the following command in the root directory of the project:

```
npm run build
```

If you look at the project folder, you will see the **dist** folder. This folder will contain the bundled React project.

### Step 2: Create a deployment folder

Before we can get into the LocalStack environment, we need to prepare the content that will be used to host our application.

Create a folder called **my-site/** in the root directory of the React project and move the **dist/** folder to the **my-site/** folder.

Inside the **my-site/** folder, create a new file called **policy.json** with the following:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PublicReadGetObject",
      "Effect": "Allow",
      "Principal": "*",
      "Action": ["s3:GetObject"],
      "Resource": ["arn:aws:s3:::my-site/*"]
    }
  ]
}
```

The JSON code above defines a policy for an S3 bucket that allows public access to read objects. Here's a simple breakdown of what each part does:

- `"Version"`: This specifies the policy version, typically left as `"2012-10-17"` (the date AWS introduced a stable policy format).
- `"Statement"`: A list of permissions defined for the bucket.
    - `"Sid"`: A unique identifier for the permission block.
    - `"Effect"`: `"Allow"` means the action is permitted.
    - `"Principal"`: `"*"` allows access to everyone (public access).
    - `"Action"`: Specifies what can be done. Here, `s3:GetObject` allows objects in the bucket to be read.
    - `"Resource"`: Points to the specific S3 bucket and all objects within it (e.g., `my-site/*`).

This policy ensures that all objects in the `my-site` bucket are publicly readable, enabling access without requiring authentication.

### Step 3: Setting up the Docker Compose file

We will be running the AWS emulator in a Docker environment. To make the startup process easier, we will make use of Docker Compose, which allows us to configure the environment within a configuration file, removing the need to write a long command in the CLI every time we need to set up the environment.

In the main project folder, create a file called **docker-compose.yml** (the name has to be exact in order for the run command to work). Inside the file add the following:

```
version: “3.8”

services:
  localstack:
    container_name: stacky
    image: localstack/localstack
    ports:
      - "127.0.0.1:4566:4566"
      - "127.0.0.1:4510-4559:4510-4559"
    volumes:
      - ./my-site:/opt/code/localstack/my-site
```

In the **docker-compose.yml** file, the `container_name` field refers to the name that we will use to reference the container when working with the Docker CLI. In this case, we are calling the container `stacky`.

In the `volumes` section, we are moving the content from the folder we created to the LocalStack container. `./my-site` references the **my-site** folder in the current directory, and `/opt/code/localstack/my-site` is the default path for the LocalStack container where we want the **my-site** folder to be placed.

### Step 4: Running the container

To run the container using Docker Compose, you need to make sure that you’re in the same directory as the **docker-compose.yml** file and run the following command:

```
docker compose up -d
```

The `-d` flag will ensure that the container is running in the background.

To get into the container and make use of the AWS emulation, you can run the following command:

```
docker exec -it stacky bash
```

Note the use of `stacky`. This is the `container_name` we put inside the Docker Compose file.

Once you enter the container environment, you can run the following command to see the different folders within it:

```
ls
```

You should see the **my-site** folder along with other files and folders. Change directories to the **my-site** folder, where we will be working in for the rest of the steps:

```
cd my-site
```

### Step 5: Setting up an S3 bucket

Now it's time to create the S3 bucket using the LocalStack CLI commands. This can be done with the following command:

```
awslocal s3api create-bucket --bucket my-site
```

We can get details about the buckets we have created using the following command:

```
awslocal s3api list-buckets
```

### Step 6: Upload index.html to S3 bucket

We can upload the HTML file to the bucket using the following command:

```
cd dist
awslocal s3 website s3://my-site --index-document index.html
```

### Step 7: Upload the remaining files

Our website still needs styling and scripts. We can use the following command to upload the rest of the content:

```
cd ..
awslocal s3 sync dist s3://my-site
```

### Step 8: Allow public access to the website

We now need to allow access to the files in our S3 bucket using a policy configuration. The following command can be run to configure the policy:

```
awslocal s3api put-bucket-policy --bucket my-site --policy file://policy.json
```

### Step 9: Access website

You will be able to access the website on the following link:

**http://my-site.s3.localhost.localstack.cloud:4566/index.html**

### Step 10: Clean up

To exit the container environment, you can run the following command:

```
exit
```

Once you exit the container, you can use the following command to clean up any artefacts generated in the previous steps:

```
docker compose down
```

## Take note

The tasks below are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, and you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

---

# Auto-graded task 1

Follow these steps:

- Navigate to the **AWS products** or **AWS calculator** page to answer the following questions on the services offered by AWS.
    - Name two services that can be used to deploy a web application.
    - If you were building a video chat application, which service would you use to handle the video streaming?
    - Take a look at MemoryDB. Which open-source tool is it based on?
    - Name three relational databases that AWS supports.
- Task submission instructions:
    - Create a file named **task-1-answers.txt** (a plain text file or a .md file if you'd prefer).
    - Create a heading for each question and add your answers below.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

# Auto-graded task 2

Follow these steps:

- For this task, you can use **draw.io** along with the **AWS calculator**.
- You will need to create an architecture diagram for a post service that can be integrated into a social media application. The service will need to have the following features:
    - Users should be able to create posts.
        - A post consists of an image and text.
    - Users should be able to delete posts.
    - Users should be able to like posts.

    - Users should be able to retrieve posts from people that they follow.
- Get the estimated monthly cost of running the solution based on the following estimated usage:
    - Users per day: 500
    - Posts shown per day, per user: 100
    - Post image size: 5 MB
    - Post content size: 5 KB
    - Posts created per day: 400
    - Post edits/deletes per day: 6 000
    - Post likes per day: 90 000
- Create a folder called **task_2** and include the following:
    - An image of your architecture diagram.
    - A PDF with the cost estimates.

Be sure to place files for submission inside your task folder and click “Request review” on your dashboard.

# Auto-graded task 3

Follow these steps:

- For this task, you will need to deploy one of your old React projects to a LocalStack S3 instance using Docker:
    - Build the React project (you can find the build command in the **package.json** file of your project).
    - Create a separate project folder containing the **dist** folder and create the **policy.json** file.

    - Create the **docker-compose.yml** file.
    - Run the Docker Compose file.
    - Enter the LocalStack container environment.
    - Enter the project folder.
    - Create an S3 bucket.
    - Move the **index.html** file to the S3 bucket.
    - Move the rest of the files in the **dist/** folder to the S3 bucket.
    - Configure the bucket policy.
    - Check if the website has been deployed by visiting: **http://<bucket-name>.s3.localhost.localstack.cloud:4566/index.html**
- Create a GitHub repository in your personal GitHub account for the project. The repository should include the following:
    - The code for the application, excluding the **dist/** and **node_module/** folders.
    - Make sure the **policy.json** and the **docker-compose.json** files are present.
    - Provide a README.md file (or a plain text file) that goes over the steps required to deploy the application on S3 using LocalStack.
    - Create a file named **task-3.txt** (a plain text file or a .md file if you'd prefer) and include a link to your GitHub repository.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

## Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Do you think we've done a good job or do you think the content of this task, or this course as a whole, can be improved?

Share your thoughts anonymously using this **form**.