Overview

Task description
You will implement a REST Application Programming Interface to some application that you have deployed to the cloud using a software container.

What you need to do
If you haven't already, read the introduction to the unit's assessment items in the Assessment Overview
In this assessment you will lay the groundwork for your cloud project by designing and implementing all the application logic so that later assessments will be primarily about cloud services and architecture. Along with developing the application logic, you will get started with some basic cloud services by deploying your application on a AWS VM using Docker.
You will submit the following artifacts to be graded:

- The source code for your application
- A 5 minute video demonstrating the functionality of your application
- A **short** document summarising how your application meets the marking criteria

Further information on what to submit appears below.
Some notes:

- While this assessment item is strictly individual, later assessments will allow for you to work with a partner, in which case you can choose one of your two applications to continue with, or you can start a new one. You may want to plan for this now, but keep in mind that we expect you to work individually for this assessment item.

Below we give a summary of the criteria. The complete assessment criteria will be posted on the Canvas assignment submission page.
Note that your application should satisfy the conditions in a non-trivial way, creating a cohesive application. Features that are added purely to satisfy criteria without meaningfully adding functionality to the application are not acceptable.

Core criteria (20 marks)
These are the core tasks of the assessment that you must attempt, and are also required to support core criteria in later assessment items. They are worth up to 20 out of 30 marks. You should prioritise these over the additional criteria below. There are six core criteria that are weighted at 3 marks each, and one that is weighted at two marks -- totalling 20 marks.
More details on how to achieve these criteria are given in the marking rubric. Be sure you are completing the tasks in such a way that the marking rubric is satisfied.

CPU intensive task (3 marks)
Your application uses at least one CPU intensive process that can be triggered by requests to load down the server. In later assessment, you will need to load down more servers so here, try to exceed 80% CPU usage for an extended time (5 minutes is enough).

CPU load testing (2 marks)
You demonstrate a script or manual method for generating enough requests to load down the server to >80% CPU for an extended time (5 minutes is enough) with enough headroom on your network connection to load down an additional 3 servers.

Data types (3 marks)
Your application stores at least two different types of data, not including user login data.

Containerise the app (3 marks)
Your application has been bundled into a Docker container and stored on our AWS ECR instance.

Deploy the container (3 marks)
Your application container has been pulled from AWS ECR and deployed to an EC2 instance.

REST API (3 marks)
Your application has a REST-based application programming interface (API) which is considered its primary interface.

User login (3 marks)
Your application has basic user login and session management with JWT. Logins must have meaningful distinctions for different users.

Additional criteria (10 marks)
These should have lower priority than the core criteria above. We have provided five options but you should only attempt four of these. There is also an open-ended *additional* that requires approval by the unit coordinator.
Attempting more than four additional tasks is allowed but you cannot achieve more than 10 marks from these tasks. We will mark only the four you explicitly tell us to consider. You should choose the best four, as we will not mark a fifth.

Each additional is weighted at 2.5 marks.
More details are given in the marking rubric. Be sure you are completing the tasks in such a way that the marking rubric is satisfied.

Extended API features (2.5 marks)
Your API extensively uses features of HTTP and follows REST principles. Here we would be looking for a solution that could be leverage by any client application (e.g., a mobile app, a command-line interface, automation workflows, etc.). Some features may be versioning, pagination, filtering and sorting.

External APIs (2.5 marks)

Your application uses an external API for data and uses that data in a novel way. Implementing what would effectively be a proxy is insufficient.

Additional types of data (2.5 marks)

Your application includes types of data beyond what is required in the core task. Here, we are considering types of storage, formats and manipulation techniques.

Custom processing (2.5 marks)

Your application implements a significant amount of custom processing, particularly in the CPU intensive process (as opposed to using off-the-shelf implementations).

You may consider implementing your own novel method that is CPU-intensive, or an innovative combination of existing tools.

Infrastructure as code (2.5 marks)

Your application extensively uses infrastructure-as-code technologies such as Docker Compose, CloudFormation, or the CDK.

Web client (2.5 marks)

Your application includes a client, accessible with a web browser, that interfaces with all endpoints of your API.

Upon request (2.5 marks)

Other functionality or aspects that demonstrate a high level of achievement in the project

- This excludes things assessed in later assessment items.
- Speak to a coordinator if there is something specific that you'd like to do and get additional credit for.

Anti-criteria

These are things that you will need to do later. You probably don't want to do these now as we will have particular criteria for how they will be implemented in later assessment items. There is no penalty for these other than the additional work that you might make for yourself. **You can ignore these if you like.**

- **User management and login method** (other than a simple hard-coded user list and very basic login, see details below). You will later implement full user management and federated single-sign-on using a cloud service.
- **Cloud services** other than a single virtual machine and ECR. You will add various cloud services later.
- **Databases.** You will later use managed services for databases, so you may want to hold off or make use of a database that is supported by AWS (eg. MySQL, PostgreSQL, MariaDB are all fine)
- **Multiple processes.** Keep everything in a single runtime. You will later break your application into microservices running separately using a variety of communication mechanisms that you will learn about in later weeks.

The Application

In this section, we will give some guidance on your application and details for some of the core criteria.

You are going to be spending a fair amount of time on this project, and you may later want to use this project as part of a portfolio to show prospective employers. So it's a good idea to put some thought into what you want to do to make sure that you will be able to maintain interest in it and do a good job.

How to choose an application to build

Nothing in this section will be assessed; it is merely a suggestion for basic questions you should have answered to ensure that you will have a good project. **You can ignore these if you like.**

Here's some key questions you should answer for yourself to help design your application:

- What **application domain** or **technology** are you interested in? (eg. machine learning, video production, physics, etc.)
- What will users want to do with your app, or what user problem(s) will it solve (**use cases** or **user stories**)?
- What **CPU intensive process(es)** will you use to support those use cases or solve those problems?
- What **data** is required to support the above processes and use cases?
- Where will be the **source** of the data?

Here are some example answers for a common type of application from past years based around sharing videos:

- **Application domain**: videos
- **Use cases**: watch a video in high quality, watch a video over a low bandwidth network connection, upload a video to share with others, see videos I have uploaded
- **CPU intensive process**: transcode video to a common format in multiple resolutions and qualities
- **Data**: videos, video metadata, lists of user videos
- **Data source**: user upload

CPU intensive process and load testing

A central part of the last assessment item is to implement auto-scaling, where additional compute resources (eg. VMs) are added when the application is under high load. For demonstration purposes you will need to be able to trigger auto-scaling by loading down your application. For this to work you need a way of generating load for your application. Your should be able to push your application to the maximum amount of load that you can practically generate. Strongly consider automating this with a script that generates HTTP requests.

In real world applications the bottleneck in dealing with high loads might be CPU, RAM, disk access, network bandwidth, or various other factors. In our context it is usually not possible to generate enough network load for that to be the bottleneck; AWS has more network bandwidth than you do. On the other hand, it isn't hard to find things that load down a CPU, so this is the approach that we will use.
Some CPU intensive processes that are suitable for this unit are:

- Videos:
  - transcoding
  - generation (eg. render 3D animation)
  - other processing (eg. stabilise)
- Audio transcoding or other processing (eg. BPM adjustment, noise filter)
- Images:
  - manipulation (eg. resize, create GIF, create stop-motion)
  - generation (eg. text to image, rendering, procedurally generated images)
  - classification
- Large language models (of modest size, see resources below)
- Training or using machine learning models
- Various data analysis tasks
- Physics simulations
- Compiling a codebase, running unit test suites, code static analysis
- Minecraft or other game server instance(s)
- Text processing (sentiment analysis, etc.), but note that students frequently have trouble generating sufficient load this way
- Various other kinds of processing tasks, such as rendering LaTeX to a PDF, compressing large files with high compression ratio, etc.. Use your imagination and *test how much load you can generate early on*.

It is not acceptable to generate CPU load through busy-work. For example, calculating the factorial of an extremely large number is not sufficient. The CPU load should support meaningful output (in the context of your application).
Video transcoding is a useful default choice if you don't have other preferences or ideas, but all of the above have be used in successful projects (other than game servers, yet.).
In many cases it is a good idea to test out your planned CPU intensive process before you spend too much time developing your app. Here is an outline for how you might do this:

- Create a minimal app that runs your CPU intensive process in response to a HTTP query, for example, by creating a small server using Express.
- Deploy this minimal app on an EC2 instance. It's a good idea to use a t3 instance type (see next section for more information about this)
- Make repeated requests to your minimal server with Postman or some other method (eg. a Node/Python script)

- Watch the CPU utilisation for your EC2 instance. You can find this by going to the Instance Details page for your EC2 instance on AWS, click on the *Monitoring* tab, and looking at the graph for CPU utilisation.
- Ideally you should be able to keep your EC2 instance above about 90% CPU utilisation. Keep in mind that you'll later need to do this for four EC2 instances, so you want to have some spare network capacity for generating more load.

Resources

While we want you to have a CPU intensive process, we don't have the resources to support arbitrarily large computations. Some guidelines:

- The general purpose instance types, t3.micro and t3a.micro are recommended as a starting point.
    - Be mindful that t instance types generally come with *burst creditsLinks to an external site.* that allow the VM to exceed its CPU baseline performance when it's under load. In "standard mode" (the default for t2 instances), this is until you run out of burst credit. This can get in the way of loading down your app as a CPU that can't burst will be limited to its baseline performance (5% to 30% depending on the instance type). The t3 and t3a instance types default to "unlimited mode", which will allow your instance to burst indefinitely, making testing much easier.
- Having less CPU resources can make it easier to load down your application, so use the smallest EC2 instance type that will run your app.
- While t3 and t3a are general purpose, you may wish to explore instance types of other classesLinks to an external site.. Speak with your tutor to figure out the best option.
- Note: Some machine learning applications like image generation and large language models might only be feasible with small models
- A list of EC2 instance types available on the CAB432 account can be found on the AWS Services Available page on Canvas.

Keep in mind that we are not evaluating the performance of your app as an app; your app is a vehicle for evaluating your implementation using cloud resources. So if the CPU intensive process in your app doesn't have the performance you'd like (for example, because of using a small language model, or processing takes a bit long), that is perfectly fine for our purposes. In the later assignments, your interfaces should not hang because of server load; they should immediately respond to requests, but processing steps can still take their time. We will introduce the concept of queueing tasks and technologies that enable asynchronous processing.

Load testing

The purpose of load testing is to show that you can generate enough load on your app to load down the CPU, which will be needed to demonstrate auto-scaling in the final assessment item. Load testing does not need to make use of all parts of your app. Some approaches are:

- Use Postman or similar to generate HTTP requests to your app

- Shell script that generates HTTP requests using wget, curl or similar
- Consider asking a generative AI to write a small script for you. By providing it information about your API endpoint, it should easily generate something to get you started.
- You will likely need to generate load many times during development, so you may want to script the load testing process even if it is feasible to do it manually.

Network speed vs. CPU speed

In the past, some students had trouble loading down their application even when using a processing step that requires a large amount of computing power. This typically happens when generating a request to do some processing requires uploading a large file. If there is a restriction on the client's network throughput --- typically a slow upload speed on a home internet connection --- such that it takes longer to upload the file than it takes to process it, then the CPU will remain idle for some of the time.

For example, suppose that it takes 1 minute to upload a file for processing, and 30 seconds to process it. Then the CPU will never go above about 50% CPU. Put differently, if the maximum rate of creating requests for processing is lower than the maximum rate of processing those requests, then the network will be maxed out before the CPU. In this scenario processing one file while uploading another will not help since only 30 seconds of work for the CPU is being generated every minute. Uploading multiple files in parallel will also not help since the network throughput is the bottleneck, and throughput will be divided among parallel requests, making them take proportionally longer. Note that the network bottleneck is on the client side, so this is not a situation where would want to scale the cloud application.

Here are some suggestions for how to overcome this problem:

- Have separate requests for uploading files and processing files. Then files can be uploaded in advance and multiple requests to process them can be made in a short amount of time when you want to test loading the CPU. This is the recommended method as it will work well with the cloud architecture concepts you will learn later.
- Test from on campus, which has very good network throughput, or from a separate EC2 instance
- Making multiple requests in parallel rather than in series can be helpful if the main bottleneck is not network throughput but latency. This can happen if the requests and processing times are very short, comparable to the network latency.
- Adjust your processing if possible to make it take longer (eg. higher compression quality, different codec, etc.)

Remember that you will later on need to be able to load down multiple EC2 instances (4 in total), so you will need to have a substantial amount of head room in your testing method when using only one EC2 instance as in this assessment item. In terms of time, you want it to take 1/4 as long to generate a request as it takes to

process it. In terms of throughput, you want to be using at most 1/4 of your client network throughput to max out one EC2 instance.

Data

In assessment item 2 you will make use of cloud persistence services to store your data. There are different types of persistence services suited to different types of data, and we'll ask you to use at least two different services. To support this your application should use at least two different types of data. We will distinguish data based on size and how your application interacts with it:

- **Unstructured data**: this is data that your application, for the most part, doesn't look inside of. This is likely to be domain-specific data associated with your CPU-intensive process. This kind of data is typically large (kilobytes or larger). Examples include images, videos, source code, datasets, and general files of unspecified type. Broadly speaking, this is "files".
- **Structured data without ACID requirements**: this is data that your application will interact directly with (searching, inserting, deleting, updating, ...). Individual entries will typically be small. Broadly speaking, this is data that you put in a database.
- **Structured data with ACID requirements**: as above, but the application has a strong requirement that the data is always consistent. Bank balances are a good example; you don't want one person to think that there is money available while others think that there isn't.

Most students will end up having some structured data and some unstructured data related to the CPU-intensive process. Some examples of structured data might be:

- Information associated with processing tasks that have been requested (eg. parameters for the processing step)
- User associated information, eg. roles, permissions, favourites, playlists, site settings, etc.. Note that this does not include user identity data, which will be discussed in the next section below.
- Metadata associated with unstructured data, eg:
    - correspondence between input data (say, a video file) and output data (say, the corresponding transcoded video file)
    - file ownership and permissions
    - file name (if using another identifier for storage, which is a good idea to avoid name collisions)
    - application specific metadata, eg. resolution, codec
- Folder structures, possibly owned by a user
- Session information

Note that user *identity* data (eg. username, email address, password) won't be counted towards the 2 types of data as this will be handled through a separate cloud service. See below.

Users

In the next assessment item you will implement user management through a cloud service that provides a user directory, authentication and authorisation. For this reason:

- User identity information doesn't count towards the two different types of data in the core criteria. This includes usernames, email addresses, passwords, etc.
- Your application needs to have some meaningful user-related functionality

Some examples of meaningful user-related functionality are:

- User roles or permissions allowing different users access to different functionality (eg. admin or regular user)
- User ownership of data
- User site settings
- Various kinds of application-specific functionality (bookmarks, playlists, likes, comments, etc.)

For this assessment item, to support demonstrating the user application logic, hard-coded credentials are acceptable (with at least 2 separate users). For example, you can have a simple login end-point that takes a username and password which are checked against a list hard-coded in your app. Then the remaining application logic will make use of the JWT that is returned upon successful authentication.

If you are using an external API

Using an external API is optional and contributes to the "additional criteria" for marking. **You don't need to use one**, but if you do:

- There must be no cost to QUT for use of your service -- so if you wish to use an API that attracts a fee, you may do so, but it is totally at your expense.
- Request your service accounts and API keys ***as early as possible***. These may be instantaneous or they may take quite a while. Talk to us early if delays become a problem. Don't just wait until the day before submission hoping that the key will appear.
- Do not commit your API keys to git. Do not hard code your API keys. Your keys must be provided to your application using some flexible mechanism, e.g., using environment variables.
- You must be mindful of API limits and restrictions. If the third-party API suddenly becomes unavailable in the moments before demonstrating your application then you cannot expect us to do anything about that. It is also out of our control. In addition to that and as per QUT policy, technical failures are not eligible reasons for requesting an extension.
- Scraping is not recommended; use a proper API.

Here are some sources for external APIs:

List	Link
GitHub: public-apis / public-apis	https://github.com/public-apis/public-apisLinks to an external site.
GitHub: trntv / apis-list	https://github.com/trntv/apis-listLinks to an external site.
API List dot fun	https://apilist.fun/Links to an external site.

Development environment:

- We ask that you develop your application using the technologies taught in the unit. There are many other possibilities, but we need to limit things to ensure that the tutors can support students without having to learn a new language/framework/runtime combination for each student. See special requests below.
- You may develop the application on a local Linux machine, WSL or a local VM under Virtual Box, or on a cloud instance. The final deployment must be on AWS.
- Please talk to your tutor if you are using a Mac with Apple silicon (M1, M2, M3 CPUs) -- we need to make sure early that your image will run on a standard VM as this has been a problem.
- Generally, to address the intended unit learning outcomes of CAB432, at minimum, use the technologies that we cover in class.

Deployment:

- You are required to deploy the app via a Docker container on top of a publicly available Linux VM. Ubuntu has been chosen because of its widespread adoption, and we will allow Ubuntu 24.04.
- We require that you set up (i.e., install and configure) your software stack in a Docker container. You should follow a similar strategy to the Docker practical, creating a Dockerfile which will allow you to create an image and to then deploy the container. Please pay *very* close attention to the port mapping. Hint: sort this out early with a very simple server and then use the same ports and port mappings forever afterwards.
- Once the image is created, you should push your Docker image to a ECR repository in the CAB432 AWS account. Deployment will then be a simple matter of pulling the image from the repo and running it on the target VM. You will need to demonstrate this deployment.
- The intention should be that the workflow to deploy your app and make it function is very simple. This fits with the nature of using Docker.
- If you are using a database, such as MySQL or MariaDB, we recommend that you deploy this using a public Docker image and use Docker Compose for deployment

Recommendations for application structure

These are not assessed criteria, but a little bit of additional work at this stage will make things easier for yourself in later assessment items. These are primarily for your later convenience. **You can ignore these if you like.**

- Think about the different parts of your application's logic and implement them in a modular fashion, ideally with separate source files.
- Use separate files for each route and make use of Express routers. This will make it easy to move these into separate processes later on.
- Encapsulate your application's interaction with data using functions or objects. Later you can modify the implementation of these functions and objects to use cloud services and leave your application logic intact.

Special requests

Cloud computing and service applications are both complex, with many different ways of doing things. Many of you will be interested in exploring some of that diversity by doing things a little differently than what we have specified above. We will consider reasonable, well justified special requests for variations on the requirements.

If you request to use some technology from outside the unit then the teaching staff might not be able to help you if you run into trouble with it. You will need to accept that risk and we recommend having a contingency plan for if the technology doesn't work out.

Make it abundantly clear that you understand the technologies you are proposing to use, and how you will use them. Without sufficient detail to convince us, we will default to rejecting your request, simply because the unit is already complex enough.

Pre-approved technologies

The following ***do not*** require special requests:

- **Cloud technologies:** AWS, EC2, Docker, ECR, CDK, Cloudformation, Systems Manager, Secrets Manager, Code Commit
- **Software development:**
    - Node, Javascript, Typescript, Express, Express middleware, npm modules
    - Python, FastAPI/Flask, pip
- **Domain specific technologies:** See the next section

For the above, it is a good idea to explore your chosen tools early to help identify any problems that might come up. Teaching staff are unlikely to have much knowledge about software or particular technologies not discussed in the unit.

Domain-specific technologies

Your app will very likely include some code or software related to its application domain. We will used the following guidelines for such technologies:

- Domain specific code generally won't be supportable by the teaching staff, so try to use a well established software package unless you are confident in your skills.
- You don't need special permission to use well established software modules (eg. ffmpeg), or software installable with apt and invoked from the main application process (eg. using exec() or spawn()). Before committing to anything, you should test these to make sure that you can get them to work.
- If you are doing a small amount of programming in another language (eg. just configuring and calling a library or similar tasks) or more involved installation and invocation of other software then talk to the teaching staff before proceeding.

- If you are doing a substantial amount of domain-specific programming then you should seek special permission. If you are using another language for the domain-specific code then we will still expect you to use Node/Python server component; we need to be able to understand this code. You should have prior experience with whatever you are proposing and make a clear argument of what you want to do and what your prior experience is.
- We may allow using another language for serving *domain-specific* API requests where the language matches your domain-specific code. The main functionality of your app (other than the domain-specific API endpoint) should always be written in Python or JavaScript.

Justifying your request:

- Explain how your request will help you to create an awesome application.
- Explain how the standard requirements and/or options in the pre-approved list above are not sufficient for what you want to do.

Some limitations to special requests:

- We can't approve special requests which would defeat the *purposes* of the core criteria or short-circuit the work of building the application (eg. reusing a previous project).
- We will typically deny requests to use a technology if the *only* reason is to avoid learning the pre-approved tools.
- If you want to write the application in a language other than Python or Javascript then we will very likely say no as we need to be able to read this code. You can still ask for special permission, but you'll have to have a very good argument for this.
- With the large number of students that we have in the unit, we don't want to be swamped with requests. We will set the bar high, so ensure that you have a good case.

If a student uses non-approved technology without special permission then we will reduce the marks for criteria where a marker without knowledge of the technology would have trouble verifying that the criteria are met from the source code.