CO3404 Distributed Systems Assignment 2025-2026

Contents

Academic Year: 2024-2025

Assessment Introduction:

Course: various	**Module Code:** CO3404 **Module Title:** Distributed Systems
Title of the Brief: Distributed System	**Type of assessment**: Implementation

This assessment is worth 50% of the overall module mark

This document consists of a detailed assignment brief and hand-in instructions. If you need additional support, contact me at ajnicol@lancashire.ac.uk or make use of the services detailed in the provided module support document.

Make sure you attach a fully completed front cover with your submission which contains your statement of originality.

How, when, and where to submit:

Submission is via Blackboard. Details of what to submit are covered in this document.

Submission should be before **13:00 hours on Friday March 13th**, after which point, submissions will be automatically recorded as late.

Feedback should be provided within three weeks of submission.

You should aim to submit your assessment in advance of the deadline.

You can submit as many revisions as you want up to the deadline but note: any further submissions should include the **whole set of deliverables**, not just what you have changed. i.e. you need to submit the code, report and video again or I won't see them as a set. Also, the only content that will be marked is the final submission before the deadline which should be a complete submission. Any submission received after the deadline will be automatically marked late by Blackboard and will be subject to the assignment capping rules unless mitigating circumstances have been approved and documented.

Note: If you have any valid mitigating circumstances that mean you cannot meet an assessment submission deadline and you wish to request an extension, you will need to apply online, via MyUCLan with your evidence **prior to the deadline**.

Further information on Mitigating Circumstances via this link.

Learning outcomes

Learning outcome from the module specification covered by this assignment, in full or in part:

1.	Critically evaluate patterns, technologies, and frameworks
2.	Compare potential technologies for the development of a distributed enterprise system
3.	Implement a distributed application using appropriate technology and frameworks

Implement a distributed system architecture

Overview

This assignment will assess your ability to implement, test and deploy a distributed system based on a given solution architecture.

In industry, a solution architect will create the high-level solution design, and the software developers implement, test, deploy and maintain the solution.

Business analysts are responsible for eliciting and documenting detailed requirements; as this assignment reflects an industry-based solution, there are quite a lot of requirements to read – this will be good practice for you in preparation for industry. Read them carefully. Make sure you fully understand what is required- or ask for clarification. **I suggest you tick each one off as you implement it to avoid missing any**.

This assignment enables you to gain experience of developing and deploying a solution using modern software architecture patterns, utilising a modern hosting approach and using tools popular in industry, particularly in large organisations. It will also test your critical evaluation skills by requiring you to propose, with evidence, the grade you think your solution deserves.

You are offered four architectural options depending on the grade you are aiming for. Each successive option is an extension of the previous one so if you make better progress than anticipated, you can develop your solution into the successive option, but all requirements of a **lower option must be completed before moving on to a higher option**:

Option1: Third class, **Option2**: 2:2 class, **Option3**: 2:1 class, **Option4**: First class

Note: much of the functionality for this assignment has been implemented as lab exercises or in code demos I've discussed and posted on Blackboard so make use of them.

Architectural Diagrams for Options 1 to 4

The architecture options are illustrated in the provided PowerPoint document. I've provided it in this form so you can edit it to enter IP addresses or other annotations, print it and keep it by your side for reference.

Code complexity is intentionally basic as this is a systems' module so is focussed on service independence, resilience, scalability and security, so don't over-complicate it – especially the user interfaces (UIs) which just needs to be functional.

NOTE: if deploying to cloud, develop and test using local host before any deployment, so you do not exhaust your Azure credit. **Stop or delete virtual machines when not in use**. Stopping a machine will stop credits being used for it, however, you will still incur a credit charge for the associated disks and public IPs but this is relatively small.

It is **your responsibility to monitor your cloud usage** which is something a developer needs to be aware of in industry.

If you want to delete all resources, delete the resource group that holds them. That way you won't miss any resources created by default. Alternatively, build the infrastructure using Terraform so the whole infrastructure can be deleted with one command then recreated with one command.

General points

- Any communication between components (containers) **within** a microservice should use a Docker network and Docker DNS service names (shown in green on the diagram).

- Communication between microservices deployed on separate Azure virtual machines (VM) should use VM private IP addresses and the Azure private network (shown in blue on the diagram) NOT VM public IP addresses (public internet shown in red on the diagram).

- All VMs need to be in the same virtual network to enable them to communicate using the Azure private network. If you use the Azure portal to create your resources, putting them all in the same resource group will by default put them in the same VNet. If using Terraform, you need to create the VNet then place the VMs into it.

- If you run out of CPU cores (Student account only allows 4 per region), put some resources in one region in one VNet, and others in another region in a different Vnet, then peer the VNets – this will enable traffic to traverse both VNets as if they were one. Just make sure your IP addresses are unique across the VNets.

- Each microservice should be resilient to any or all the other microservices failing; this should be tested and demonstrated in the video for all options. i.e., stop one application, e.g. joke, then show in the video that the other applications, still work.

Do this for all applications. This is an illustration of testing a non-functional requirement of system resilience.

- By default, private IPs are dynamic so don't survive a server restart. You can make them static at no cost.

- By default, public IPs are dynamic. You can make them static if you want to but there is a slightly higher cost.

- If any of the following requirements are ambiguous or you don't fully understand them, don't guess. Ask. A professional developer would go to the business analyst, or product owner, or architect for clarification; implementing incorrectly is expensive to the business.

- An Azure student account limits you to **3 public IP addresses**. This is not a problem as you don't need any for option 1, you only need two for option 2 and only one for options 3 and 4. Use public IPs for testing then deallocate the IP when using private IPs.

- If it's not a requirement, don't implement it. Over-engineering is a drain on company resources and extends delivery times and will attract no additional marks. Make your code flexible and maintainable so future enhancements are relatively simple to add and code is reusable.

- Each code submission **must be accompanied by a report and video**. Carefully read the detailed requirements for the report, the video and your chosen solution option specified within this document. Any of the three components missing will be considered a non-submission as the assignment submission consists of a set of three artefacts.

- **Make sure you review each of the requirements to ensure you have fully satisfied them for the chosen solution option**. Check that your report and video clearly demonstrate that each and all the requirements for a specific option have been fully satisfied. Any that are not, should be taken into account when proposing your grade; I will review and grade your work in exactly the same way so we should come to a similar conclusion.

- You will need to submit all the code required to rebuild and test your application so read the submission section carefully so I can check your application. I may ask you for a personal demonstration, particularly if your report or video are poor or unconvincing, or looking at your code I decide the video can't be authentic as the code doesn't match the demo. I will interview you to determine your level of understanding and ask for specific tests to be demonstrated. If you fail to attend a session when required to do so, you will be awarded a fail grade.

Assessment marking approach

The University now uses a banding scheme. The following is from the academic regulations and will be used to assess this assignment. The requirements for each band are specified in the requirements section of each Architecture Solution Option. No grade values, other than those in the numerical equivalent column are acceptable so consider this in your self-assessment. For example, you can score a 65 or a 68 but there is no awardable value in between so if you are not convinced you satisfy 68 then the best mark available is 65.

Level 4, 5 and 6 assessments (e.g. Undergraduate programmes)

Band	**Numerical equivalent**
Exceptional 1st	100
Very High 1st	94
High	87
Mid 1st	80
Low 1st	74
High 2.1	68
Mid 2.1	65
Low 2.1	62
High 2.2	58
Mid 2.2	55
Low 2.2	52
High 3rd	48
Mid 3rd	45
Low 3rd	42
Marginal Fail	35*
Mid Fail	30*
Low Fail	25
Fail	10
Non-submission/Penalty/No Academic Merit	0

*can be compensated

(Minimum Pass/Capped Mark)	40**

** The use of grade band marking is intended to encourage markers to assign grades on the basis of the band/classification of the work submitted. The use of the minimum pass mark is reserved for assessments passed at resubmission or passed for a capped mark. A marginal fail would receive a mark of 35 and a marginal pass would receive 42.

What you need to submit

Video presentation

A **mandatory** 10 minute video presentation (+/- 1 minute) is required to demonstrate your understanding of the subject. This should cover the operation of your solution and demonstrate that you have satisfied the functional and non-functional requirements provided in this document. Note: for option 4, the video should be no more than 15 minutes long.

I strongly suggest you write a script and follow it to ensure you cover all the areas clearly and concisely and don't waste time thinking of what to say next. You should speak quickly, but clearly, and interact with your application quickly; don't worry, I will keep up. Do not submit a video which is longer than 11 minutes as that is a failure to meet a specific requirement.

The video should be submitted as a **single** standard .mp4 format - i.e., **NOT** as part of the .zip file containing your code, not a link to YouTube, not a video of your screen using your phone or any other method – submit the professionally created video into Blackboard.

Your video should be clear and playable in Windows so **test it before submission** because if I can't hear what you are saying, or can't see what you are doing, I'll consider it a non-submission.

Use a screen recorder like teams. I use ActivePresenter; it is professional and comprehensive and the free version has minimal limitations.

Report

The report, no more than 2000 words, will mainly assess learning outcomes 1 and 2, as well as providing a clear justification for the grade you are proposing. It is also used to discuss more code detail you only had tie to discuss at a high level in the video.

Your report should have the assignment front sheet prepended, with your confirmation of authentication as the first part of the report.

The following key components of the report must be present

Self-assessment / critical evaluation

In the first paragraph, you should clearly state, with justification, how well you have satisfied the requirements and what areas could be improved. Using the assessment criteria, **propose an overall grade % justifying the mark you are proposing**. E.g.,

"My submission satisfies and demonstrates all the requirements for a **mid 2:1 grade of 65%** because …"

Or maybe: "My submission satisfies and demonstrates all the requirements for a mid 2:1 grade of 65% other than x & y where I had problems (describe). I explored (whatever approach) but was unsuccessful so reduced my assessment to a **low 2:1 at 62%**

accordingly".

This will help you to carefully review the requirements and critically assess your own work. Your mark will not count but will be used as a measure of your ability to critically self-assess. i.e., it should be close to my awarded mark as I will be following the same specification in this document. Note: in the banding scheme, there are no "between" marks; you need to select a mark from the provided banding table.

Critically evaluate patterns, technologies, and frameworks

Compare potential technologies for a distributed enterprise system

These are related learning outcomes so create two sections or do them both in one. This section should consist of a critique of the architectural design pattern used in this assignment and consider alternative architectural and messaging patterns and other technologies. This is not a discussion of how your code works, but a critique of the overall approach. As an engineer, would you have implemented a joke service the way you have been asked to? What other solution options should have been considered?

In this section you should address at least the following questions.

- Could you suggest alternative or improved architectural pattern with justification supporting your proposal as an alternative to the message-based microservice-architecture?
- Could you suggest and justify an alternative or improved architectural pattern?
- Would you recommend the assessment provided architecture as the best approach for this joke service? If not, why not? What do you suggest as an alternative?
- What alternatives might you propose and why?
- You used NodeJS and the Express framework for your application and web server. What alternatives could you have considered? Why would your suggestions be as good or better?
- Your application servers also served static content. Is that a good idea in an API-based solution?
- What alternative architectural approach to serving the user interface could you have used and why?
- You used RabbitMQ for asynchronous messaging and events, and http for synchronous messaging. Is this a good idea? Can you think of an alternative or potentially better approach?

The report must be submitted as a **single** Microsoft document, **NOT** part of a .zip file or any other format.

DO NOT use AI to write this report. By all means, use AI as **one** of your research tools but the report should be a synthesis of your research, in your own words and appropriately referenced.

Code discussion

You will have already demonstrated code and tests in your video presentation. However, use this section to explain any specific techniques you considered, which you

chose to solve various problems and why, what you found difficult and how you overcame the problems. i.e., an opportunity to provide more supporting information such as more detail on areas you discussed only at a high level in the video due to the time constraint.
For the higher level options, you should discuss in more detail here how you implemented key elements of your solution such as, continuous deployment, certificate creation and deployment, OIDC implementation etc.

Source Code

Source code and database export should be submitted as a **single .zip** file. The file must be decompressable in Windows. Test it before you submit it. **Do NOT** include the **node_modules** folders – just delete the node_modules folders from each project before you zip all the projects and database files. I'll recreate the application from your dsource files, .env, compose.yam, Dockerfile, database export file, package.json files – i.e. everything needed to recreate your solution. If you submit Terraform files, only submit the source files **NOT the .terraform directory**

You can use AI to help with code generation but don't rely on it as it is often wrong or unreliable and can often lead you down time-consuming rabbit holes. Use it a research tool or boiler plate code generator but make sure you fully understand any code you submit because if I'm not convinced by your video or report or a skim of your code, I'll interview you. Any code you copy directly from an AI tool or elsewhere should be referenced.

Database content

You should export your MySQL, Mongo or both databases so I can reconstruct your solution and database. MySQL should be exported as a single SQL file including schema. MongoDB should be exported as a single JSON file. The file(s) should be included in the zip file with the rest of your source code.

Again, do **NOT** include **node_modules** or **.terraform folders**. These can take a very long time to downloadas the file will be huge, so don't put me in a bad mood when I'm about to assess your work 😊

Solution options

You will develop, test, deploy, document and demonstrate an online joke service by implementing a software solution to **one** of the four provided solution architectures. The following sections consist of an overview of the requirements for each option, a breakdown of what is required for each grade and a set of detailed functional and non-functional requirements. A brief discussion of what is required in the video and report specific to each option is provided to be considered in conjunction with the more general discussion of the video earlier in this document.

Option 1 – Third Class

This is a minimal distributed system consisting of a joke application and a submit application. If either the joke or submit application fails, the other should still work. Although this is not a microservice, it offers very basic distribution and a basic level of service resilience. You can use MySQL which uses port 3306 or MongoDB which uses port 27017, as your database server. Each component should be deployed as a Docker container listening on the port numbers illustrated on the architectural diagram in red, and provide OpenAPI documentation as described in the detailed requirements to gain maximum marks. This option is based on a lab exercise in week 6 where you created an application and database. There are demos on Blackboard with code relevant to this option; you need to understand the demo code enough to be able to adapt it.

Any communication between containers within your solution, should use a Docker network and Docker DNS service names. i.e., access to the database is via a private docker network (shown in green on the diagram) – i.e. use a single Docker compose file for all three containers.

Your testing and video demo should focus on how your code works and demonstrate any functional and non-functional requirements. You have a choice of solution options within option1 depending on the grade you are aiming for.

Joke application overview

Users can request a specified number of random jokes of a specified type, e.g. general, dad, programming etc., The joke is retrieved from the database via an API call based on the selected joke type but randomly selected from jokes of that type. The joke types are returned from the database via a web API.

The user is presented with a web page which will display the requested joke. Setup is displayed, then after 3 seconds the punchline is revealed.

Submit application overview

A user can submit a new joke (setup and punchline) based on a joke type selected from a dropdown list, or add a new type if it doesn't exist in the list – e.g. knock-knock, sport etc. The new joke and its type are posted to an API endpoint to be added to the database.

The user is presented with a web page containing appropriate html elements to enter a setup and punchline. They should be able to select a joke type from a dropdown list or enter a new joke type to be added to the database.

Marking criteria:

Low 3rd: Report, video and all requirements must be complete. All functional and non-functional requirements should be clearly demonstrated.

The applications only need to run in Visual Studio Code or from the terminal and use a locally installed database.

The joke types' list in each application can be added directly into the HTML; you do not need to implement a new joke type in the submit application.

No containers are needed for a pass at this level.

As the joke types are fixed in the html, there is no need to implement the **/types** API endpoint.

Mid 3rd: As low 3rd, but with the addition of OpenAPI documentation for the submit application.

High 3rd: As Mid 3rd, but both applications and database are running in Docker containers. The dropdown lists holding the joke types for the joke and submit applications should be populated dynamically from the database using an API call. You need to implement an option for the user to create a new joke type in the submit application in addition to the option to select an existing joke type from the list.

Detailed functional and non-functional requirements for Option 1: 3rd class grade band

These requirements are for a high 3rd class band. Based on the marking criteria above, choose which of the detailed requirements you implement to achieve which marking band you are aiming for.

Joke application:

1. Create a basic web user interface (UI) for a user to request a randomly selected joke from a user specified joke type selected from a list. The returned joke setup and punchline should be displayed as follows: display the setup. Three seconds later and on a different line, display the punchline.
2. The UI should have a button which, each time it is pressed, will display another randomly selected joke of the user specified type.
3. The joke types in the selectable list, e.g. a dropdown list, should be read from the database using an API call to ensure the list reflects the joke types held in the database, including any new ones that may have been added by the submit application. i.e., when you interact with the list (open the list), it requests an update of its content by calling the appropriate API endpoint
4. The joke application should run on a NodeJS express server
5. Any web UI components such as HTML, CSS and JavaScript client files should be served as static content from the node server.
6. The joke application should have two API endpoints:
    a. **GET /joke/:type?count** This returns one or more randomly selected jokes from the database.
    ?count The joke endpoint takes between zero and one query parameter:
    If no query is provided, the API returns **one** joke of the specified type
    If **'count'** is provided, **'count'** jokes of the specified type are returned
    If there are less jokes available than requested, return what there is. You should accommodate an **"any"** type to select a random joke of any type – i.e. randomly selected from all jokes in the database regardless of type.

    Although the UI requires **only one** joke, your API should be capable of returning more than one as others wishing to use your API should be able to get as many jokes and their types as they want as their application may not be to simply display jokes but maybe print them on Christmas crackers for example. Don't display more than one joke on your web page but **demonstrate the API returning multiple jokes** in the video using Postman or a similar tool

    /:type is a path parameter specifying which type of joke(s) the user wants

    b. **GET /types** This endpoint returns all the joke types held in the database to populate the front-end list. This should be called each time the dropdown list is interacted with to ensure the list is always up to date.
    There should be **no duplicated** types in the list

7. The joke application should be deployed into a Docker container.
8. The database should be either MySQL server or a MongoDB server (your choice) and deployed locally for a low or mid 3rd or in a Docker container for a high 3rd as specified in the marking criteria. This is the master data store for the whole joke service.
9. If using MySQL, you should create two tables: **jokes** and **types** and establish an appropriate relationship between them. The types' table should not allow duplicate entries.
10. If deploying into containers, a persistent volume should be created to ensure the database data survives a VM restart or a container re-creation. You need to demonstrate this in your video.
11. If the Submit application is down, joke should still work.

Submit application:

1. Create a basic web user interface to submit a new joke based on a joke type selected from a list, or a new joke type entered by the user that reflects the new joke if an appropriate one isn't available in the list. Note: if you are hard-coding your list in the html (low or middle 3rd class), this new type won't be visible in the UI list so you need to show it added to the database using MySQL Workbench.

   The user should be able to:

   a. Write a setup and punchline
   b. Select a joke type from a dropdown list
   c. Enter a new joke type if required to better reflect the current joke's theme
   d. Submit all their data at once by clicking a button
   e. Prevent the submission of data unless all required fields have been completed
2. The submit application should run on a NodeJS express server
3. Any Web page, CSS and JavaScript client files should be served as static content from the submit component's node server
4. The submit component should have three endpoints:
   a. **POST /submit** This is used to post the new joke and type to the submit application. The application writes the joke and possibly type, to the database
   b. **GET /types** retrieves the list of all joke types to populate the UI types' list
   c. **GET /docs** returns Open API (Swagger) compliant documentation for all three of the Submit endpoints. **Testing the API endpoints using the Swagger documentation should be demonstrated in the video**

5. If the joke application is down, submit should still work
6. If using MySQL as your database, ensure the submit service does not write duplicate types into the types' table
7. Demonstrate system resilience and functional operation of your application in the video
8. Show the database content and structure using Workbench, Atlas or similar tool

Video specifics for this option

As option 1 is the most basic option, your video discussion should focus on the code, how it works, what could be improved and how. Demonstrate in the video that you have satisfied all the requirements for the required grade.

Clearly demonstrate in your video and report if required, that **all** the functional and non-functional requirements have been satisfied to justify your self-assessed grade. For example, you should clearly show the database structure and relationship between tables if you chose to use a relational database. You should show the tables or collection content changing as you add jokes and types.

If you find your video is less than nine minutes, then you are unlikely to be explaining it in enough detail or not covering all the points listed such as demonstrating all the functional and non-functional requirements; this will affect your grade.

Note: do not simply read out the code line by line. I need to be convinced you understand the principles and chosen approach. Eg., discussion of the benefit of database connection pool if you have used one, or maybe you had problems connecting to the database – how did you overcome the problems? How have you populated the joke types' dropdown etc. **Not**: "this is a for loop; it executes the code within it 5 times".

Option 2 – Lower Second Class (2:2)

Functional and non-functional requirements for Option 2: 2-2 grade band

This option should satisfy all the functional and non-functional requirements of option 1
The joke and submit applications are deployed into Azure cloud services and distributed across different virtual machines as microservices. They communicate with each other across the Azure private network using synchronous messaging for the joke types retrieval and asynchronous messaging via RabbitMQ for the new joke and any new joke type submitted to the ETL application which will add the data to the database.

This approach is resilient to whole microservice failure. The loose coupling is achieved with the message queue and the limitation of the synchronous http messaging is managed to some extent by providing a local cache of the joke types in a file within a Docker volume. There are various code demos on Blackboard to help you with this.

Marking criteria:

All components of the previous option must be fully implemented

Low 2:2: Submit microservice should populate the UI dropdown with data retrieved from the joke microservice using an http request. You should cache joke types in a file and clearly demonstrate joke types being read from the file if the joke service is down. You only need to demonstrate your solution in containers running on a local machine, not in the cloud. New joke and type should be written to the message queue. There is no need to implement the ETL application. Simply demonstrate that you can write to the message queue by showing the data in the queue manager. Read and remove the message using the queue manager. To demonstrate the dynamic loading of new types in the UI, add a new type directly into the database using Workbench; show it being added to your file cache in a Docker volume.

Mid 2:2: Implement the ETL application. This should receive jokes and type from the queue and write the data into the database. Jokes and joke types should be clearly shown being written into the joke database via the ETL container.

High 2:2: All containers are deployed and shown to be running on Azure virtual machines: one VM for joke, database and ETL containers, and another VM hosting Submit and RabbitMQ.

Joke microservice

The joke application has the same detailed requirements as those stated in option1, including your choice of database: MySQL server or MongoDB database server. However, new jokes are not submitted directly into the database but via an ETL component to decouple the joke and submission functionality.

The ETL component will **E**xtract the messages from the submit message queue, **T**ransform the data, format into that required to write into the database, then **L**oad the data into the database. This enables new jokes to be submitted even if the joke application has failed or been taken offline so provides a substantial level of resilience.

Note: You should use a larger virtual machine for this microservice as it is running three containers. I suggest using a B2s as opposed to the usual B1s. Unlike the B1s which is completely free for up to 750 hours per month per CPU, the B2s will consume some of your $100 credit. Don't forget to stop all VMs when not in use. Disks connected to stopped VMs still consume from your $100 credit as do public IP addresses. It's worth setting up a cost alert to monitor the rate at which your $100 is being used. Unlike the free Azure account which will charge you real money from your credit card if you go beyond these limits, the student account will simply stop working. If you are worried about any of these limitations then ask; there should be absolutely no issue with resource costs for this assignment if you manage your resources sensibly.

ETL container

1. An Extract Transform and Load (ETL) application should be implemented to extract jokes and types from the queue submitted by the user. Use a simple queue with a single producer and consumer. Transform the data as required, then load them into the database tables (if MySql or collection if Mongo). The ETL application should be implemented as a RabbitMQ message **consumer** which registers a callback with the submit microservice message broker. When a message is added to the queue by the submit **provider**, the message broker calls the registered ETL callback and sends the message to the ETL component. ETL will write the message content into the database and acknowledges receipt of the message to enable the queue manager to delete it.
2. The ETL application should run on its own NodeJS server deployed into a Docker container as shown on the architectural diagram. There are no endpoints in this application, but you may want to add at least an "alive" endpoint to check it is up.
3. If using MySql server, ensure that no duplicate joke types are written to the types' table.

Submit microservice

The three endpoints have the same detailed requirements as those documented in option 1. Additional requirements are:

1. The submit application retrieves the joke types requested by the UI by making an http request to the joke microservice. To ensure these types are available to the UI if the joke service is down, they should be cached in a file as shown on the diagram. If the request for types fails, those held in the cache should be returned to the UI. The cache should be refreshed on every **/types** API call from the UI to ensure it is as in sync with the master data store as possible.
2. The types' cache should be implemented as a Docker volume for resilience
3. Rather than the submit application writing new jokes directly into the database, as was the case in option 1, it will write them to the message queue.

4. The message queue is implemented using a RabbitMQ message broker and should be deployed as a container and the queue should be persistent to survive a RabbitMQ container failure and recreation.

Video specifics for this option

This section discusses at least what I expect you to demonstrate for a high 2:2. If you have submitted a low or mid 2:2 then clearly you only need to cover areas relevant to that level.

Focus your presentation on demonstrating the functional and non-functional requirements you have implemented, but you also need to discuss the code. At this level however, you don't need a line-by-line discussion, focus more on the more challenging areas covering the requirements for your proposed grade. For example, creating and accessing the queue by showing queue activity in the RabbitMQ management console. Show how you have modularised your code and made it maintainable. Clearly show the containers running by accessing them locally using the container port numbers, or the IP address if you implemented Azure VMs showing all services available via the APIs. Show the data being written to the database from the ETL application.

Show the dropdown menu being populated dynamically. Use the RabbitMQ management console to show messages on the queue waiting to be consumed when you have stopped the joke service to illustrate the service resilience NFR. Show the data consumed from the queue when you restart the joke service. Show the structure of your database and show data being written to it. Show the joke types' cache working when jokes is down, and it being synchronised when jokes comes back up.

Option 3 – Upper Second Class (2:1)

Functional and non-functional requirements for Option 3: 2-1 grade band

This option has the same requirements as options 2 but introduces an API gateway. An API gateway is a typical component used in industry to provide a single endpoint to access all the microservices. This enables the microservices to look like a single service, decouples the microservices from the internet by operating as a reverse proxy, and can provide common functionality across microservices in a single place.

Marking criteria:

All components of the previous option must be fully implemented
Low 2:1: Access Microservices using a single http origin. E.g. internet url http://20.30.40.50:80 and rate limit the joke API requests

Mid 2:1: Implement **https**. E.g., https://20.30.40.50:443/path…

High 2:1: Create the Kong virtual machine using Terraform infrastructure as code. You do not need to deploy your containers using Terraform. Be careful to ensure your VM is in the same VNet as the others

API gateway

1. Implement the Kong API gateway as a container running on its own virtual machine
2. Client requests to the gateway should be forwarded to the appropriate microservice
3. Implement rate limiting on the joke microservice to ensure users calling the API cannot abuse it by continuously calling it. Set the limit to a low value so you can demonstrate it working
4. Implement SSL (TLS) on the gateway. You can use a service such as let's encrypt or mkcert to create a certificate. The certificate should be deployed to the Kong VM not added to the Kong image
5. Kong can be run with or without a database. You are only required to run Kong without a database, but you can use database mode if you feel you have a requirement to so
6. Create the Kong virtual machine using Terraform. You will probably find that the benefits of creating one VM with Terraform will convince you to create the others using it too, but you only need one to satisfy this requirement

Video specifics for this option

This section discusses at least what I expect you to demonstrate for a high 2:1. If you have submitted a low or mid 2:1 then clearly you only need to cover areas relevant to that level.

Focus your presentation on demonstrating the functional and non-functional requirements you have implemented, but you also need to discuss the code. At this level however, you don’t need a line by line discussion, focus more on the more challenging areas covering the requirements for your proposed grade such as database pooling, creating and accessing the queue by showing queue activity in the RabbitMQ management console. How you have modularised your code and made it maintainable?

Demonstrate all the functional requirements are satisfied by accessing a single IP address. Discuss your kong.yaml file. Show the data being written to the database from the ETL application. Show the dropdown menu being populated dynamically. Use the RabbitMQ management console to show messages on the queue waiting to be consumed when you have stopped the joke service to illustrate the service resilience NFR. Show the data consumed from the queue when you restart the joke service. Show the structure of your database and show data being written to it. Show the joke types cache working when jokes is down, and it being synchronised when jokes comes back up.

Discuss how you created and deployed your TLS certificates and demonstrate https access to your services with no browser security warnings.

Discuss the key sections of your Terraform files.

Option 4 – First Class

Detailed requirements for Option 4:
1st class grade band

The architecture is now more event-based than message-based by implementing a Event-Carried State Transfer (ECST) pattern discussed in the notes and illustrated in a provided code demo called "rabbitPubSub" on Blackboard. It also introduces another microservice called moderate.

The moderate microservice enables an authenticated human moderator to review the newly submitted joke and type by reading the submitted joke and type, if available, from the submit microservice via the queue and presenting it to the moderator in a UI which enables the moderator to review and if required, edit any of the components – setup, punchline and type. If there are no new jokes available to moderate, an appropriate message should be displayed, and the UI should poll for a new joke – e.g. every second. On submission by the moderator or deletion by the moderator if the joke is wholly inappropriate, the next one in the queue, if available, should be loaded into the UI. Note: deletion in this case is simply a non-submission and a call is made for the next joke. Typically, however, we would generate a deleted event which could be subscribed to by an analytics microservice for example, but that's not necessary in this case.

This modified architecture requires the broker to be available to all microservices and be configured to manage events. The broker works in the same way as before, but the messaging is a little different.

The submit microservice (producer) puts the submitted new joke onto a queue (submit) which is received by the moderate service (consumer). When the moderator has reviewed the joke and potentially edited it, it is sent by the moderate microservice as a producer to a queue (moderated) and received by the joke microservice ETL application (consumer) to be written into the database as shown in the architectural diagram. This is only slightly different to the previous option with the addition of an additionalqueue.

Rather than continually call the joke type API to try to keep the types' cache in sync with the master data store (SoR), the ETL application will write the type to the types' database table if it is not already present and if a write is successful, ETL will create a type_update event and send it to the broker. All subscribers (moderate and submit) will receive the event and update their types' file cache. This enables the types' API call to simply read the file content because the synchronous API call to the joke service used in previous options is no longer needed. The code demo rabbitPubSub in conjunction with the slides in the notes illustrate exactly how to implement this pattern. You can call the queues what you like as the names on the diagram are illustrative. As the type_update events need to come from uniquely named queues, I suggest simply appending mod and sub as namespaces to create queue names mod_type_update and sub_type_update but it's up to you.

Additionally, you need to implement a MySQL server **AND** a MongoDB database server to demonstrate your knowledge of both technologies. These do not need to run at the same time; the administrator can choose which to run by a simple config change.

You should build all your infrastructure using Terraform. You should automate the build, push and any file deployment to the remote servers – i.e. continuous deployment. You can do this using Terraform local and remote executioners, GitHub actions, a combination of both or any another suitable approach as long as it is fully automated. Discuss your approach in the video at a high level and in the report at a more detailed level.

Marking criteria:

All components of the previous option must be fully implemented
Low 1st: Implement the moderator microservice including types' cache and event handling

Mid 1st: Implement a second, different database in your joke microservice which is configurable by changing an environment variable from MYSQL to MONGO or visa versa.

High 1st: Implement VM build, image push, pull, copy, config and container start for the moderator microservice as a Continuous Deployment pipeline using Terraform or any approach of your choice as long as the whole process is automated. There is no need for automated testing.

Very High 1st: Implement moderator authentication using Open ID Connect. This can be at the application level or using a Kong API Gateway plugin

Exceptional 1st: Implement all requirements. Provide professional looking and usable user interfaces (there is no need to use react or anything fancy – html, css and js is adequate). Provide a high-quality report and test strategy. Demonstrate the success of all functional and non-functional requirements through the report and a concise, well-articulated video

Moderate microservice functional requirements

1. Create a basic web user interface to display a submitted joke setup, punchline and type read from the submit event payload on the queue
   a. Display the setup and punchline in editable web UI elements
   b. The moderator can edit or submit the joke and type or reject it by not submitting it and polling for the next one
   c. The moderator UI should display a new joke if one is available or display an appropriate message then enter a polling cycle to check for an available joke at a timed interval – e.g., 1 second. Note: don't worry about losing jokes by refreshing the page - that's just further front-end implementation
   d. The UI should populate a list, e.g., a dropdown list with available joke types so the moderator can see what types are already available in the joke database. The moderator can override the submitted type by choosing one from the list if it is more appropriate, or edit the proposed type for a more appropriately named type if there isn't a suitable one in the list
   e. The list of types should be updated using the type_update event as shown on the diagram
2. The moderate microservice should run on a NodeJS express server
3. Any Web pages, CSS and JavaScript client files should be served as static content from the moderator component's node server
4. The moderate application should run in a Docker container
5. The moderate component should have three API endpoints:
   a. **GET /moderate** This **gets** a joke and type from the queue if available or the UI displays a response indicating there are none currently available then starts the polling cycle
   b. **POST /moderated** This is used to **post** the moderated joke and type to the moderator's queue
   c. **GET /types** This returns the types from the docker volume holding the file cache of joke types
6. If the joke service and / or submit service is down, the moderator should still be able to submit new joke and type as a business continuity workaround
7. RabbitMQ should run in a Docker container on its own VM
8. Add an alternative database to your joke microservice. Your solution should be fully modular such that on setting an environment variable, your joke microservice will be deployed using a MongoDB **OR** MySQL server in a container, **NOT** both. i.e., when you run Docker compose up, either a MongoDB container is started OR MySQL server is started and the application can connect and operate from whichever is running.
9. The moderator is required to authenticate to gain access to the moderator microservice as this is a privileged role. Access to the microservice also requires authorisation for the **POST /moderated** endpoint. You should research and use an OpenID connect (OIDC) identity provider. You can authenticate to the moderator service or authenticate to Kong using a Kong plugin. There are free services available to support OIDC third party authentication and authorisation.
10. Build and automate deployment of the Moderator microservice using Terraform or other continuous deployment approach.

Video specifics for this option

This section discusses at least what I expect you to demonstrate for a very high 1st. If you have submitted a low, mid or high 1st then clearly you only need to cover areas relevant to that level.

For option 4, there will be much more to discuss so focus on the system operation as opposed to code detail other than areas specifically included for this option or those requiring personal research or innovative solutions.

Consider the presentation options specified for option 3, and clearly illustrate all the functional and non-functional requirements, such as operating both databases and switching services off to demonstrate others continue and resynchronise when failed services are back up. But for this option I also need to see the events processing illustrated, a demo of continuous deployment of at least one of your microservices, and a demonstration of the authentication you have implemented. You can pause the recording whilst lengthy processes take place such as VM creation and Docker installation.

Given the additional complexity and content of option 4, your video can be up to but no more than **15 minutes**.

You should briefly discuss the code for these key areas but given the time constraint, explain in detail in your report how you have implemented the key requirements for this option where you have not had time to discuss them in detail in the video.

Summary of submission artefacts

You should submit **three** artefacts:

1. A **word** document containing the report. It should have the assignment front sheet with your confirmation of authentication as the first part of the report.

2. An **.mp4** video demonstration of your solution. No more than 11 minutes for options 1-3 and no more then 15 minutes for option 4

3. A **.zip** file containing all source files, including .env, package.json and database exports and anything else needed to rebuild your solution– do not include the **node_modules or .terraform folders**

If you resubmit before the deadline because you have improved something, make sure you submit **all three artefacts** again as I will only see the most recently submitted artefacts as a group.