OPENSKY EXERCISES
Solution by: Ayush Chaturvedi

Exercise 1.

Abstract:
In the given architecture I was able to identify a few flaws in terms of scale and a few foundational aspects of the overall system with respect to event driven design principles for ETL pipelines. Firstly, the way audio records are processed and stored both at the feeder service path and transcripts retrieval at the API service path is not fault tolerant and has some logical errors. Secondly, the given design also does not take full advantage of the kafka as a messaging system and finally, the architecture fails to address the entirety of system design by overlooking the aspect of high availability and CAP theorem. Therefore, my solution targets all these aspects and provides an entire solution that encompasses all aspects of system design right from the Users' browser.
The solution of ex1. Is stated below and addressed as follows: In section 1, I describe the problem design and answer Q1 of the exercise and identify bottlenecks (in which I answer Q2). In section 2, I propose a new architecture with an HLD answering the Q3 and also provide code snippets and pseudo code for the Q4. In section 3, I first discuss my approach in evaluating the new vs. old design and also provide a monitoring solution or telemetry pipeline which enables us to compare two designs in a quantifiable manner .

Section 1: Problem statement and Disadvantages
The overall design is shown in Figure 1.

Following is the architecture described in the exercise where the process starts at Feeders:

1. ATC recordings are sent to a micro service (not necessarily REST based) through a java/kotlin based backend service that acts as an entry point for recordings data..
2. The backend service then interacts with the cassandra object storage and stores metadata and recording into it. This communication is done via the cassandra driver(API) for JVM based languages over TCP.
3. The transcription service (another backend service not necessarily REST based) then fetches recording upon arrival into the cassandra and then will process the transcription and metadata. This service (per my understanding) is an asynchronous batch job which gets triggered and processes "new" records in the object stores.
4. Post successful or unsuccessful transcription the transcription service will send them to kafka topic "atc.transcript". This communication takes place through kafka consumer/producer protocol over TCP via java/kotlin client for kafka.
5. Next, when any user comes in to visualise his/her dashboard through a javascript/Vue.js based application, the application makes an http based REST request to the API backend in Quarkus. The user provides the deviceid of the feeder. The sample service path made by javascript application is shown in Fig. 1
6. The API server then instantiates a kafka consumer within the topic "atc.transcript" and starts reading all the records in the topic, post successful check for user roles.

The read records are then filtered for the “device_id” given by the user in the “params” of the API GET request.

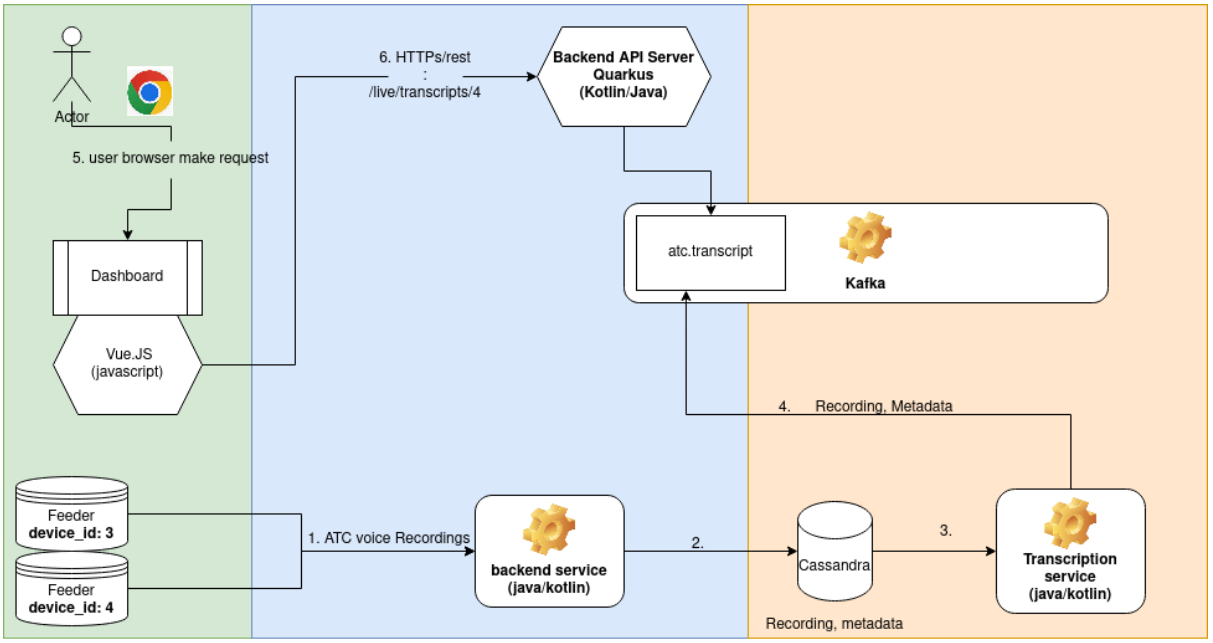

Figure 1: Overall Architecture per question statement

Problems with the current architecture

The current architecture is not scalable in its entirety and faces significant design flaws and logic flaws.
Specifically, in the backend API server :

1. The kafka consumer reads all the records present in the topic every time a REST call is made; this is simply redundant and not a feasible/robust design. For every user’s every request, reading all the records present in all the partitions of a kafka topic are simply not logical enough in terms of scale and throughput of the application.
2. There is no filter based on the device_id in the LiveStreamEndpoint.kt file.
3. Using an asynchronous (cron job based) service (batch job) for the transcription service is not resilient. As there might be a case that the job gets missed due to some server issue and on another tick it does get started. This will leave the job to process huge amounts of data and creates a lag for the next iterations/ticks of the job. This architecture also leaves this application itself vulnerable as scaling such an application will only deteriorate the condition since the checkpointing and partitioning of the data with current architecture needs to be handled by the app developer.

Section 2: Proposed Architecture
The Proposed architecture is shown in Fig. 2

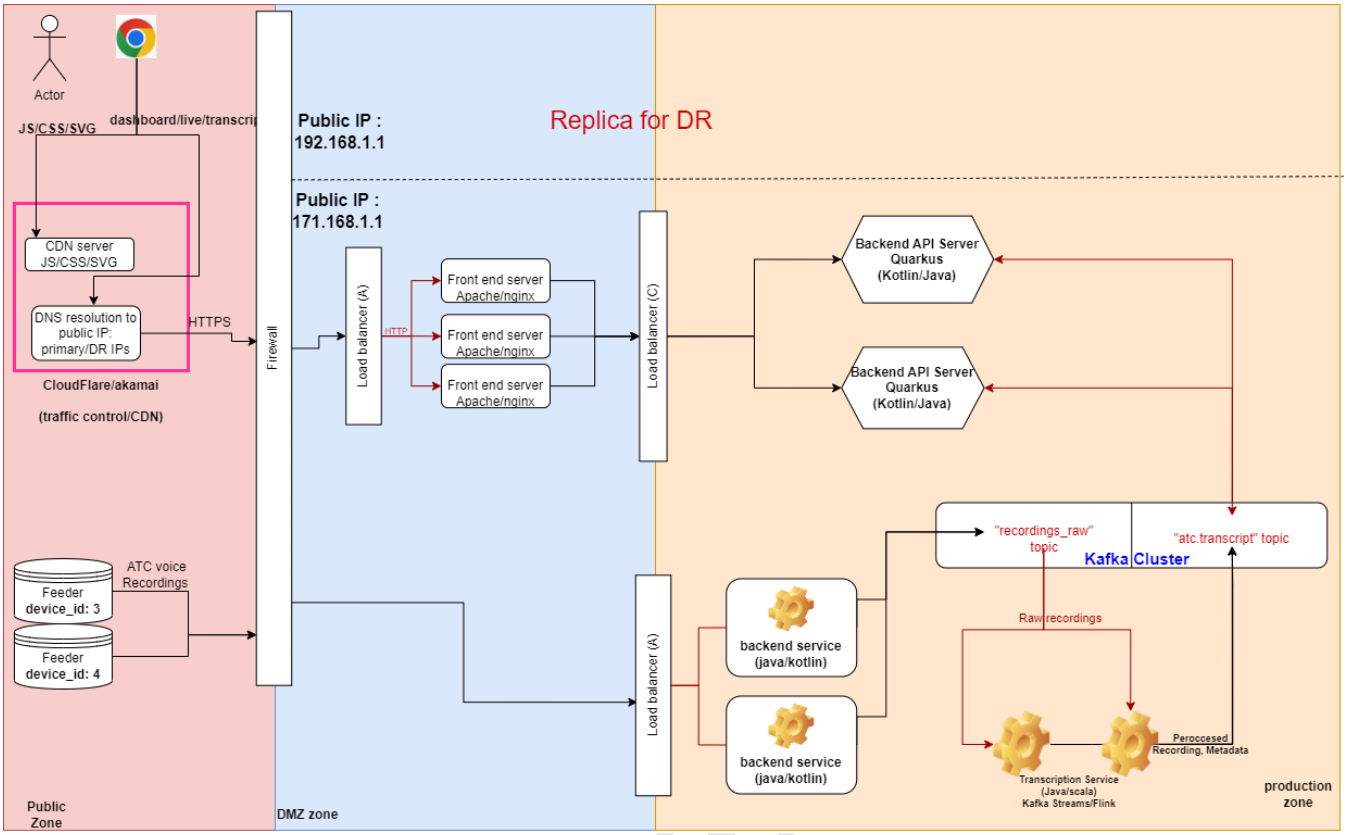

Figure 2: New proposed architecture

The significant changes are listed as below :

1. The frontend needs to be scaled and also respect the security therefore the entire infra needs to be broken down into zones for the purpose of security. This is labelled in the bottom section of each zone demarcated in colours red, green and yellow and described below:
    a. **Public Zone**: The red zone is marked as a public zone and is not the responsibility of the organisation, this implies that no production code, services, authentication can take place here.
    b. **The DMZ zone**: is where the HTTPs traffic get logged at the firewall and ssl is then offloaded. Any header based authentication is also done there (for eg. lua plugin for nginx header based authentication) . The sole purpose of this layer is to separate business logic and core app servers from the security layer. The DMZ zone caters for the firewall, load balancer A and C for the frontend and backend servers, respectively. The frontend servers are only present to server dynamic content for Web and also to redirect/route api requests to their respective servers/load balancers.
    c. **The production zone**: the production zone is where all the application infra lies, all the communication between the applications in this zone is non authenticated and done over TCP for reliability of packets.
    d. The entire infra is replicated to provide high availability through provisioning a cloud provider such as akamai or cloudflare and leveraging separate geo

location for DR site and primary site and separate public IPs (as shown in Fig.2). These separate public IP can then be used under traffic manager to serve an active-active or active-passive availability model.

2. The fronted server serving the static HTML and other static contents need to use a fast CDN server. This can be achieved by using any cloud CDN providers such as akamai, cloudflare, AWS etc.
3. By breaking down the requests made by frontend into static and dynamic requests we are able to scale the entire infra modularly. For instance, the dynamic requests can be scaled by leveraging a load balancer over multiple fronted servers which then make calls to a load balancer for backend API server.
4. Hiding frontend server behind load balancer and firewall has multiple advantages:
    a. Although the latency in the service path (for dynamic requests) increases, but user never experiences it, because the static content is served first (very fast through cached CDN) and dynamic content follows asynchronously in the time the static DOM is ready.
    b. Fault tolerance and high availability are guaranteed by using replication and load balancers.
    c. Choice of apache server or nginx can be discussed, the latter being more efficient because of non blocking i/o and extensibility.
5. The similar approach for high availability is followed for the service path followed by feeder recordings. The recordings pass through the firewall and end up into a backend service which acts as a kafka producer and pushes records to a new topic **“recordings_raw”.** By this approach we are getting rid of the cassandra/redis object storage for raw recordings, leveraging kafka as an ephemeral storage and messaging backbone. This has following benefits:

    a. Kafka guarantees constant time fetching of recording O(1) since it uses a log structured file system and hashing based internal data structures that stores data in byte format. This means that kafka performs the same for 5Gb of data and 5 Tb of data.
    b. Using kafka cluster ensures reliability and fault tolerance through zookeeper. (the newer version of kafka uses kraft protocol for consensus and gets rid of zookeeper as well, sweet :))
    c. Kafka also provides features such as timely truncation, compression and comes with out of box support for different data stores.
    d. Kafka takes away the headache of checkpointing the application from application developers as kafka itself stores the checkpoint of each consumer group within a separate topic in the kafka cluster itself.
    e. The business logic of this backend application is hence minimal as it is just an entry point for data.
6. As the record enters the “recordings_raw” topic, the transcription service acts as a kafka consumer and processes each record in an event driven stateless fashion and sends the processed records to the **“atc.transcript” topic** in the same cluster along with the recording metadata. This enforces an event driven design and by using kafka avro based schema with versioning. It allows developers to scale the applications independently which is not available if a database is used. (For eg. if a new field is added to the data from feeders, entire change has to be propagated from

datastore to API backend service, however with avro based schemas backward compatibility can be worked out).
Note: Rules can be applied to allow/disallow reading/writing to topics.

7. Post processing from the transcription service the records lay inside the atc.transcript topic.
8. The design of this topic has to be carefully done as this topic acts as our database to fetch records for a given device and user. Therefore , we will partition the data in this topic based on key: device_id. A custom code had to be written in the kafka streams application in the transcription service to achieve this. A java based code example of which is attached in **custom_partitioner** file in the same repository (Note: the example is not runnable and only for demonstration).
9. As soon as a user comes in and makes an HTTP request for a fetching transcriptions of a particular device id the user's request will end up into the backed API server within which lies a kafka consumer which reads data from this topic.
   The kafka consumer will start reading all the records of the topic with a condition for device id, an example is listed below:

```
KStream<String,TranscriptQueueItem> inputStream= builder.stream(inputTopic);
KStream<String,TranscriptQueueItem> completedFlowStream = inputStream.filter(value-> value.get("device").equals(device_id))
```

This filtering is **missing** from the provided code in the LiveStreamEndpoint.kt file.
A corrected version of it is also added in the github repository.

Section 3: Evaluation

First we need to define criteria for comparison which can be Accuracy and Latency.
For this purpose we will integrate TICK (telegraf , influx, chronograf and grafana) stack in our architecture and collect telemetry data to influxDb.

We will install telegraf at each api server and server's of kafka. After this we will also enable access logging in the spring boot/quarkus application servers, frontend apache/nginx servers and kafka cluster.
Telegraf will provide us with all the server telemetry and also pass the access logs to influxDB via its file reader plugin.

Leveraging the TICk stack will give us telemetry for API endpoints and by comparing the access log timing for API server with new and old solution we will be able to figure out the E2E latency (from access logs of the frontend servers). Not only this, we will also be able to compare our solution in terms of scale and server health. Moreover, using the mentioned monitoring pipeline with time series also gives us p99 , p95 metrics of API service path which are very crucial. Furthermore, we can easily integrate any alerting system such as Zabbix to notify us whenever there is a health alert for application servers.
In order to measure the accuracy we will use kafka telemetry to validate the message count in the both topics of kafka post and pre transcription. This can be done by leveraging kafka telemetry plugin in telegraf wich pushes many data point regarding broker health including consumer group lag and record counts.

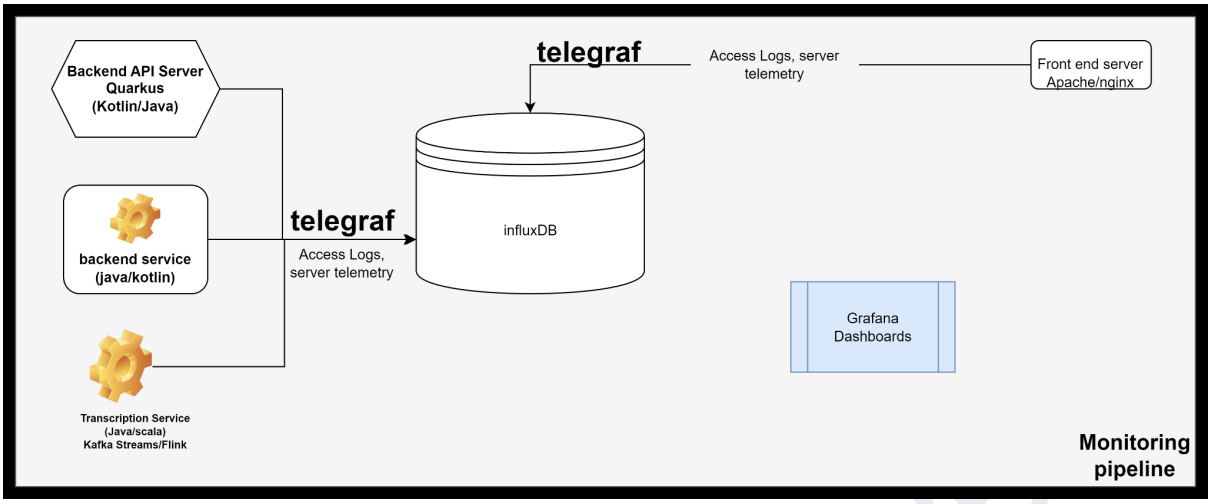

Figure 3. Monitoring pipeline example