Assumptions:

1. Building a demo legacy system could have been a good option but it will be difficult to mimic “unpredictable response times” and “multiple attempts”. So, I have used WireMock.
2. Employee IDs are validated by an upstream service before hitting this API.
3. Didn’t made the feature where, if an employee misses his/her clock-out, they can do it from somewhere else (out of the scope).

Q1) Describe how you would build the system so that user actions (check-in/check-out) are separated from third-party API calls, for example by using queues and events.

1. To separate user actions from **3rd party** API calls, I have used Transactional Outbox and Kafka (message queues). This guarantees that no event is lost because inside a single block, we are saving the data in two databases (Attendance & Event).
2. The kafka asynchronously handles the messages and takes care of lags and spikes.
3. Used two topics with two separate consumers such that even if legacy API fails, it won’t affect the email workflow. Everything remains stable (clock-in API too).

Q2) Explain how you would ensure consistency, handle errors and retries and what you would do with messages that keep failing

I implemented the following:

1. To ensure consistency, I used @*Transactional* annotation to save Attendance record and Outbox event within same block.
2. To properly handle retries and errors in legacy and email API, I used springboot’s *DefaultErrorHandler,* which gives enormous power to the consumers to try 4 times with a 30s delay between each.
3. If the 3rd party APIs keeps on failing and the consumer is unable to acknowledge to kafka, once the 4 retry attempts exhaust, kafka will deliver it to the **Dead Letter Queue**. This initiative will prevent **Poison Pill** problem. Kind of same logic was used during publishing to kafka, it’s just a scheduler will try to publish and once the retry attempt exhausts, it will mark as failed in the DB (Event).

Q3) Suggest ways to deal with rate limits and outages of the recording system.

1. Implementing a library like *Resilience4j,* which can pause or slow down the kafka consumer, if the rate limit exceeds. We can also use *Exponential Back-off* instead of fixed retries.
2. To deal with outages, we can implement a circuit breaker or a *DefaultErrorHandler.* Otherwise for major ones we can implement a DLQ or configure retention settings in the kafka to hold data for longer periods.

Q4) Briefly outline how you would monitor and trace what happens to each message and how you would handle changes to the message format over time.

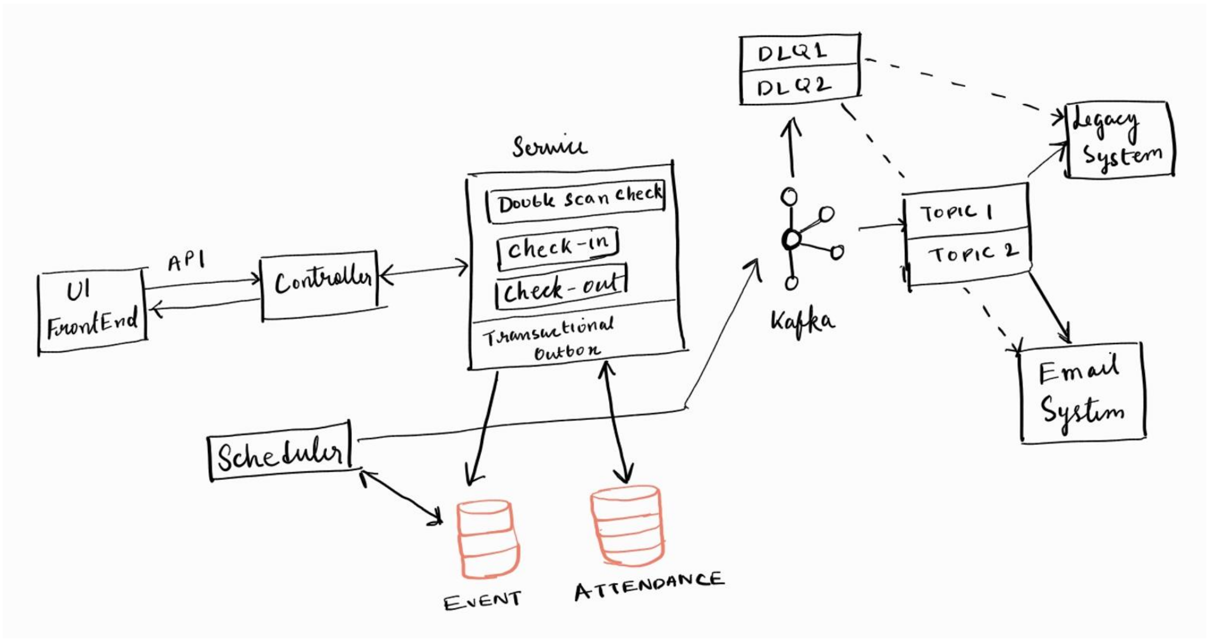
Monitoring and Tracing:

1. We can use ids and use them in every log, database and kafka headers.
2. Spring Cloud Sleuth can be also used to give us a visual timeline.
3. *Kafka UI, Prometheus* and *Grafana* are some of the very important choices.

Message Evolution:

1. We can use Apache Avro which provides schema evolution capabilities.
2. We can also use versioned DTOs with correct ObjectMappers.

Architecture Diagram:



Once the user's check-in by their card, it will fire an API call having *employeeId.* Then the control goes from controller to service layer. Here before processing, we are going to check for double scan (both for check in/out) with a buffer of 60s each and once it passes, it will check the db for existing record. If not present then it will create a new record in Attendance db otherwise it will perform a check-out operation where the check-out time will be updated and *totalWorkingHr* will be calculated and saved in Attendance and Event db under same ***Transactional*** block.

Then a schedular will pick up the data from Event db and finally pass it to the kafka producer. In case of failure, the scheduler will retry (4 times, 30s each), if still fails then it will mark the status for that data as **FAILED.**

We have two consumers waiting for the message (legacy & email). Each consumer will acknowledge only when the respective API call succeed. In case of failure, it will retry 4 times with a gap of 30s. If it still fails then, they will be moved to DLQ.

Also to mimic the original behaviour, WireMoch and Mailhog has been used.

LLM usage on this project:

- I used Gemini 3 Flash to:
    - Know more about DLQ in consumers and also helped me to implement them but I twigged them in the end.
    - Generate commands in Docker-compose.yml file, specially KRaft commands.
    - Took help to setup WireMock only.
    - Generate a basic structure of README.md
    - Suggest me some articles on @Transactional, Kafka and docker