**Aim: To Implement a Message Queueing System**

Introduction:

A message queueing system is a software architecture pattern that enables communication between different parts of a distributed system by allowing them to exchange messages. In this system, messages are stored in a queue and are retrieved by consumers when they are ready to process them. This decouples the producers and consumers of messages, allowing them to operate independently and asynchronously.

Message queueing systems can be implemented using various technologies, including open- source solutions like Apache Kafka, RabbitMQ, and ActiveMQ. In this practical we are going to implement a simple message queueing system using Python.

RabbitMQ Model:

RabbitMQ is one of the most widely used message brokers, it acts as the message broker,

“the mailman”, a microservice architecture needs.

RabbitMQ consists of:

1. producer — the client that creates a message

2. consumer — receives a message

3. queue — stores messages

4. exchange — enables to route messages and send them to queues

The system functions in the following way:

1. producer creates a message and sends it to an exchange

2. exchange receives a message and routes it to queues subscribed to it

3. consumer receives messages from those queues he/she is subscribed to

Implementation:

We are going to implement a job manager as described in the below figure.

Components of our message queueing system are:

o Publisher – produces jobs/messages into the queue

o Consumers – consumes the jobs

o RabbitMQ broker – contains the exchange and queue

o Connections – denoted by double-sided arrows

o Channels – denoted by colourful bands within the connections

Technologies Used:

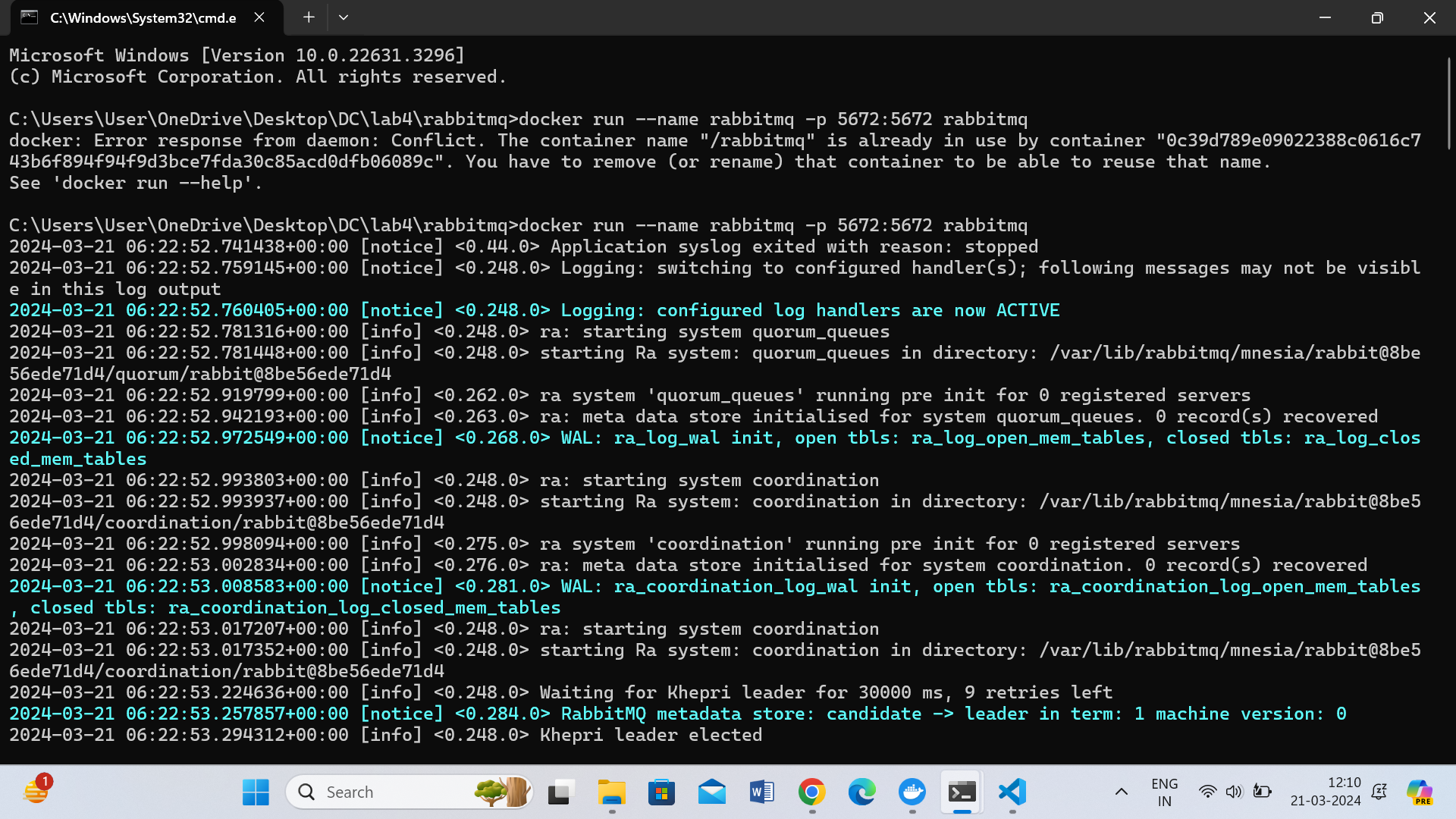
• Docker

• RabbitMQ Image

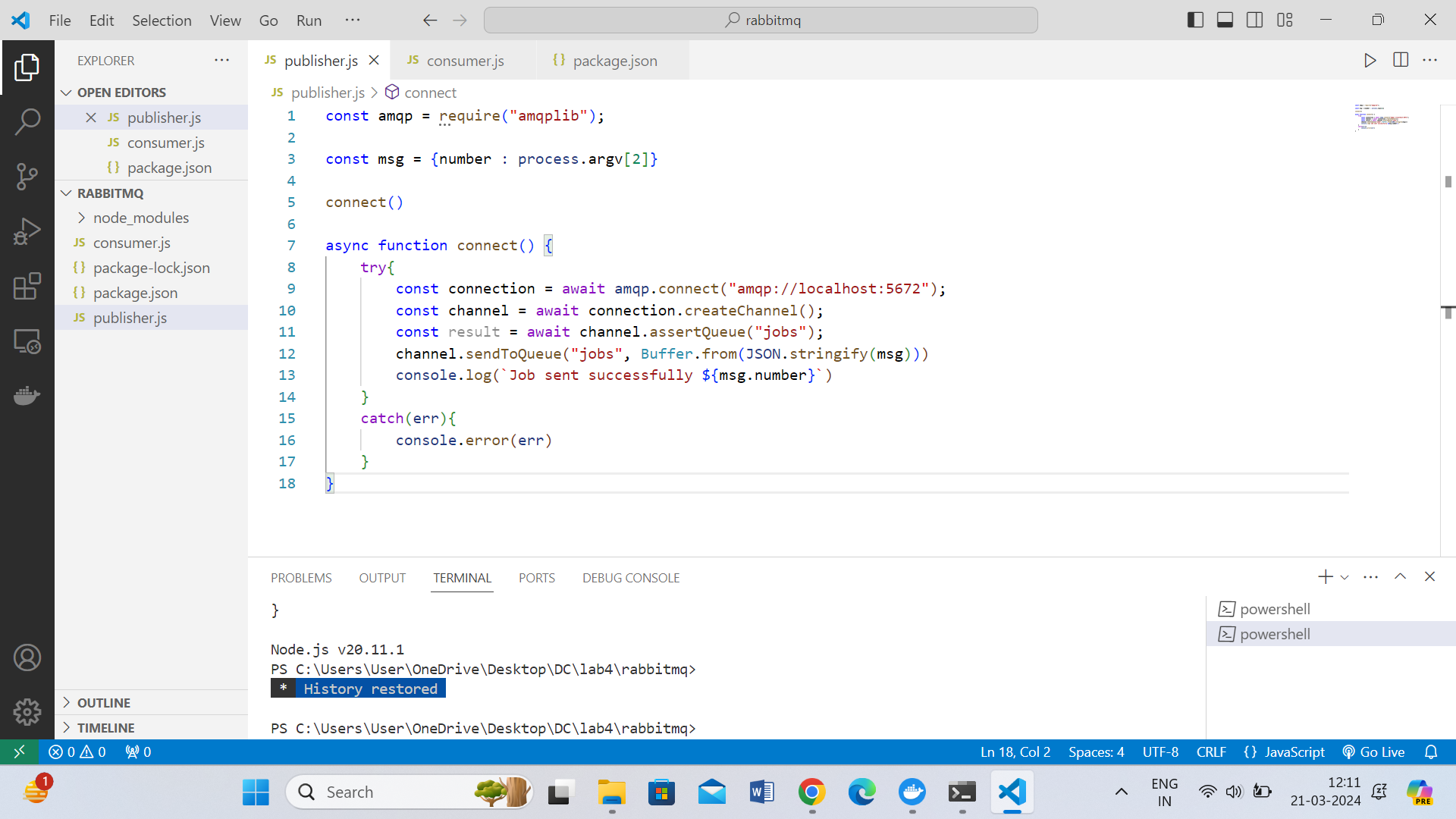
• Node.js

• amqplib Library

Step 1: Run RabbitMQ’s Docker Image



Step 2: Write a Producer Program - publisher.js



o A Node Library named “amqplib” is used to implement AMQP (Advanced Message

Queueing Protocol)

o We then create a connection with the RabbitMQ server.

o Then a channel is created using connection’s createChannel() function

o This channel is used to create a new queue named “jobs” which resides within our

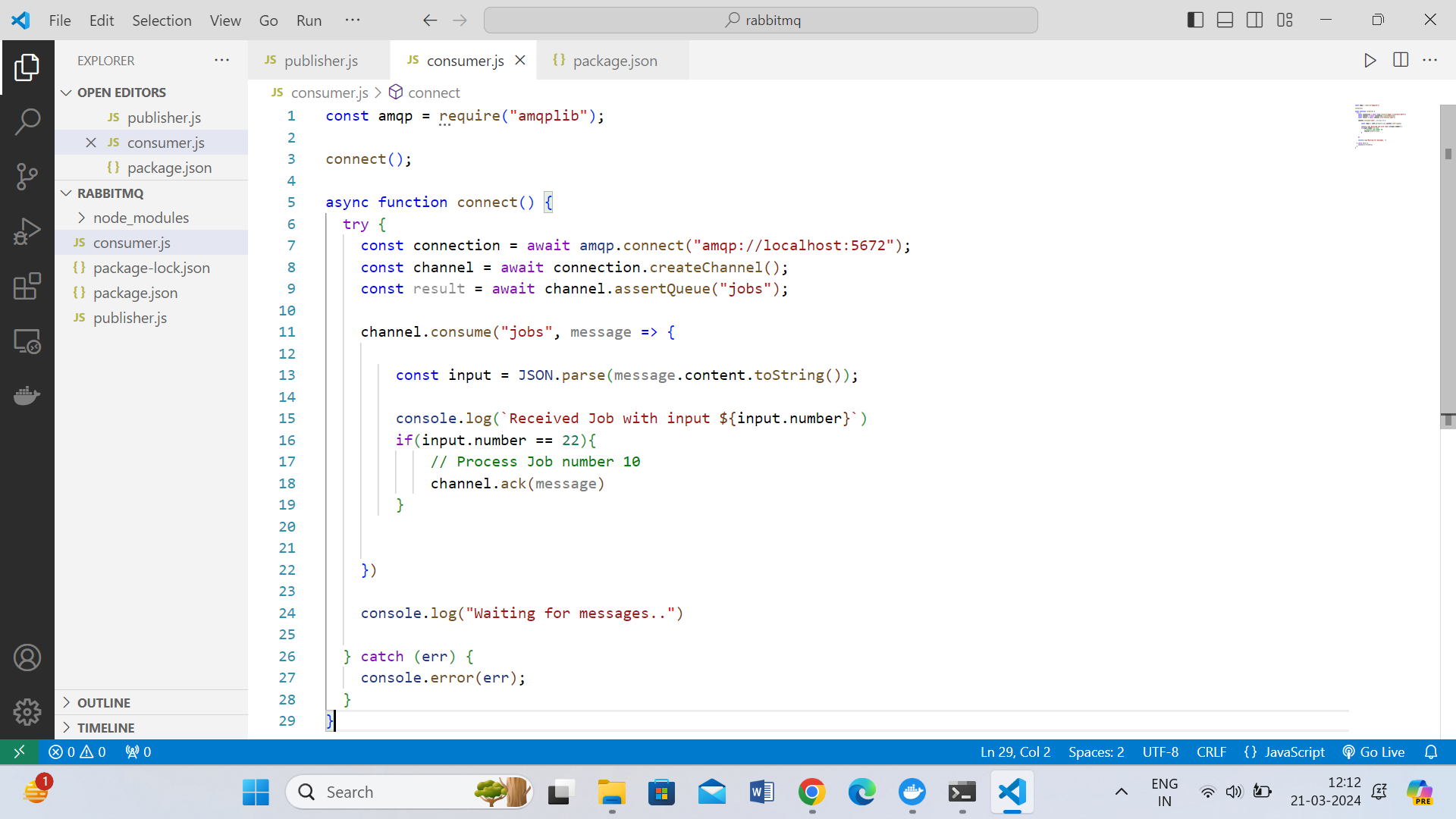
RabbitMQ broker

o A new message is enqueued within the queue. In other words, a new job is

produced. The content of this message is provided as a command line argument

when we run our producer program

Step 3: Write a Consumer Program – consumer.js



Here too, we create connection and channel the same way as in our publisher.js

program

o Then we write functionality to consume the messages already present in the queue

o Let us say that our consumer only consumes message number 22. Hence, if the

queue has a message number 22, it will be consumed by the consumer and an

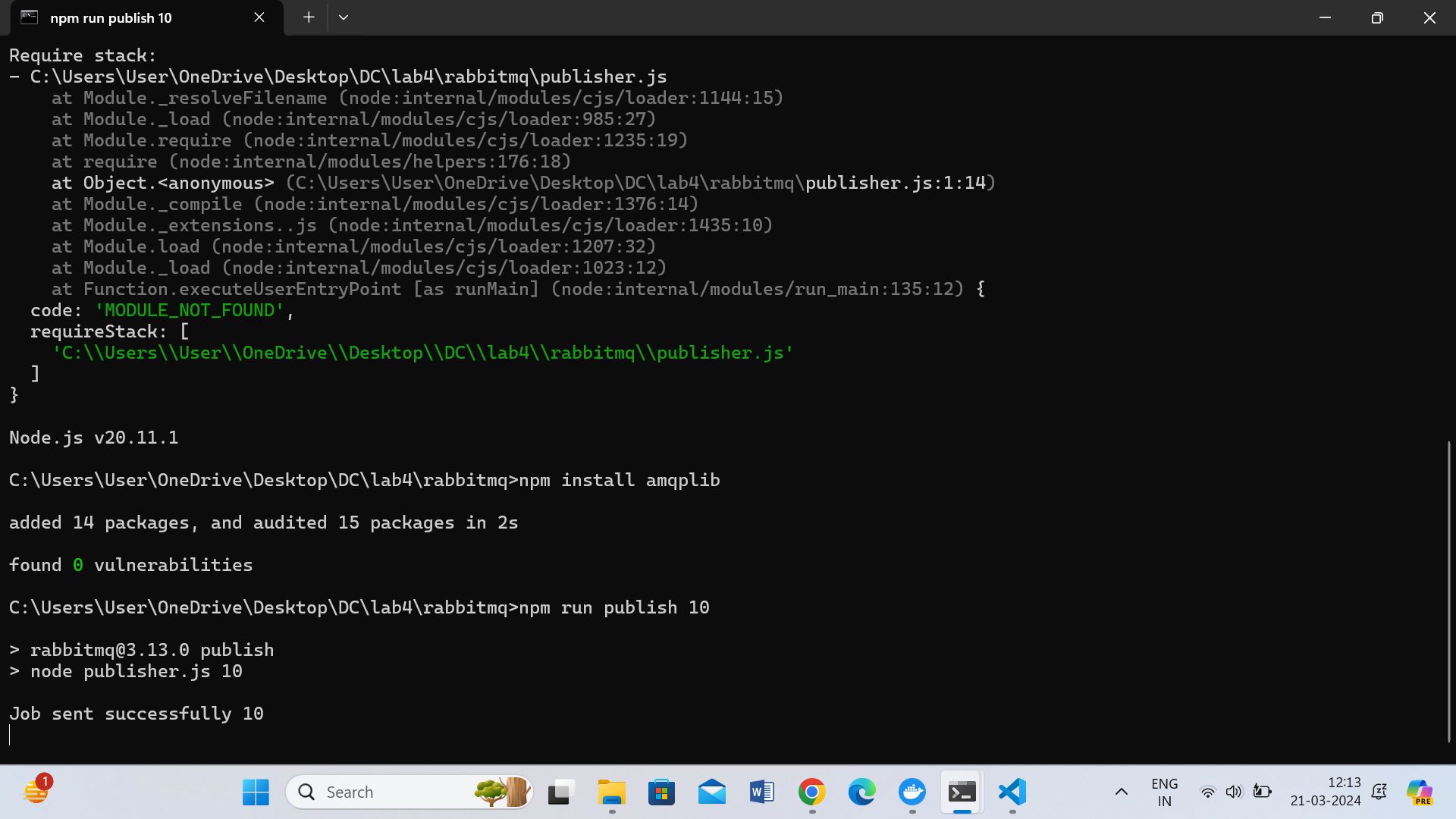
acknowledgement will be passed to the RabbitMQ server. Subsequently the message

number 22 will be dequeued

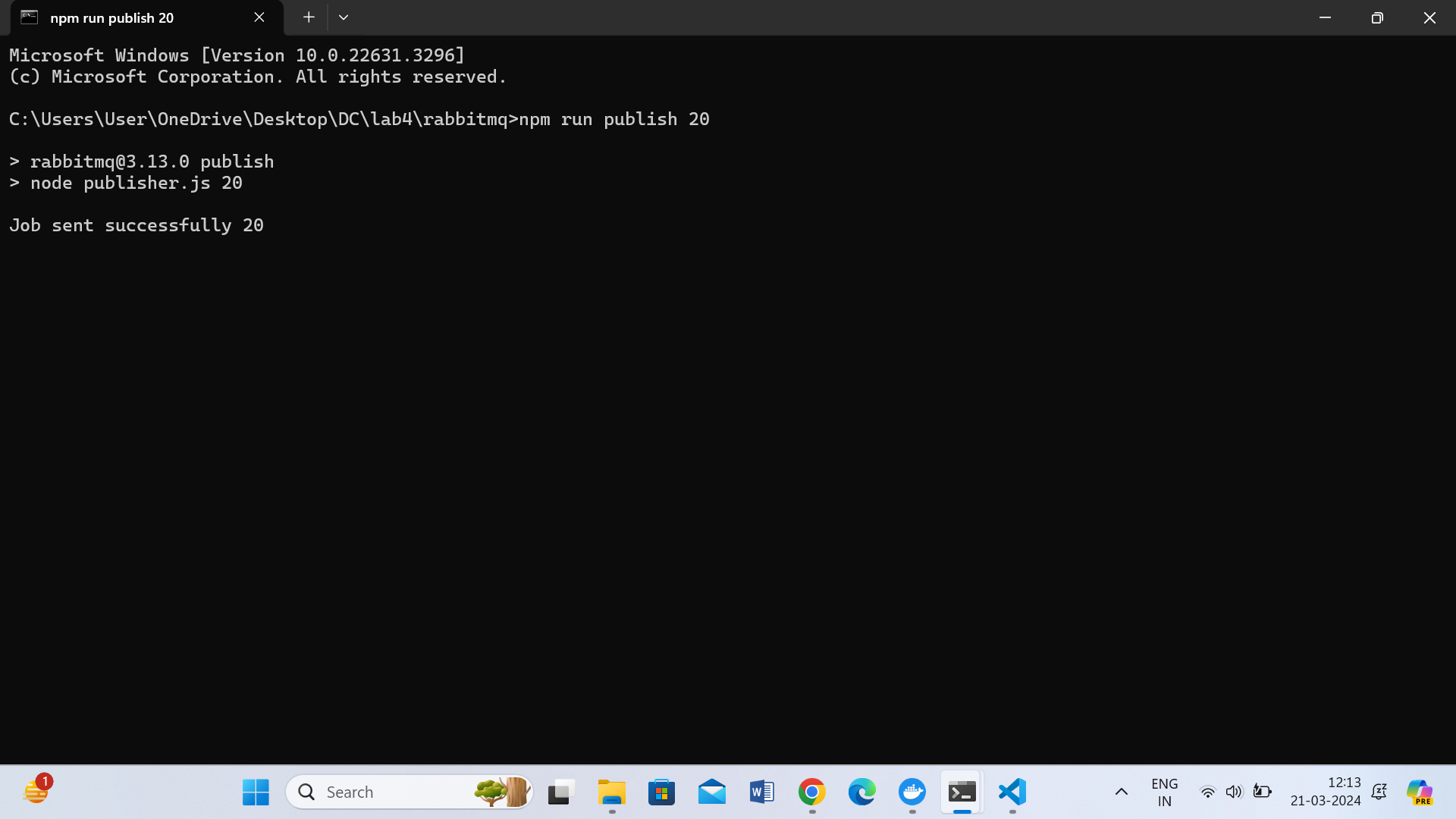
Step 4: Testing our system

Running Producer – publisher.js

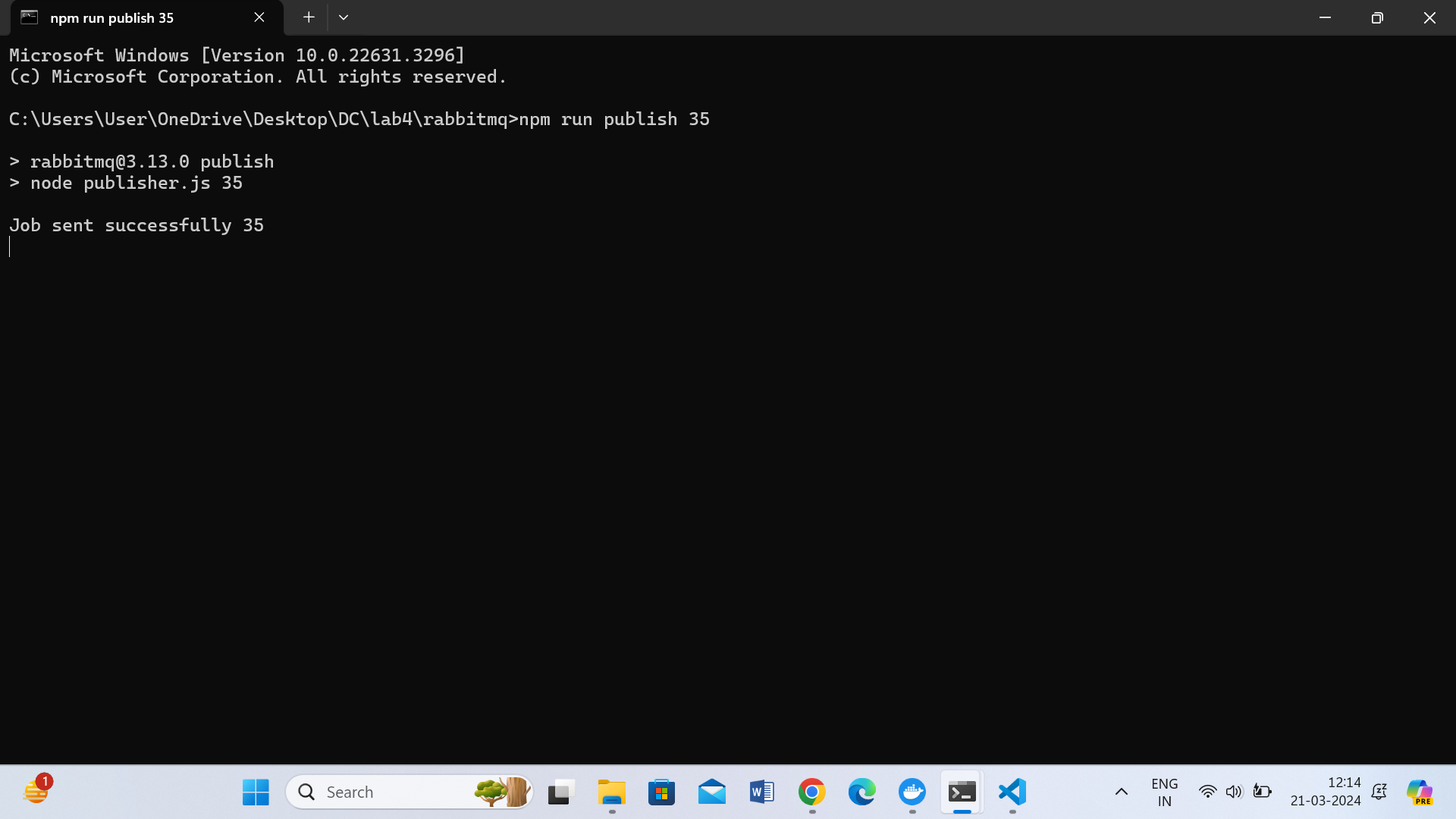
Publish job 10



Publish job 20

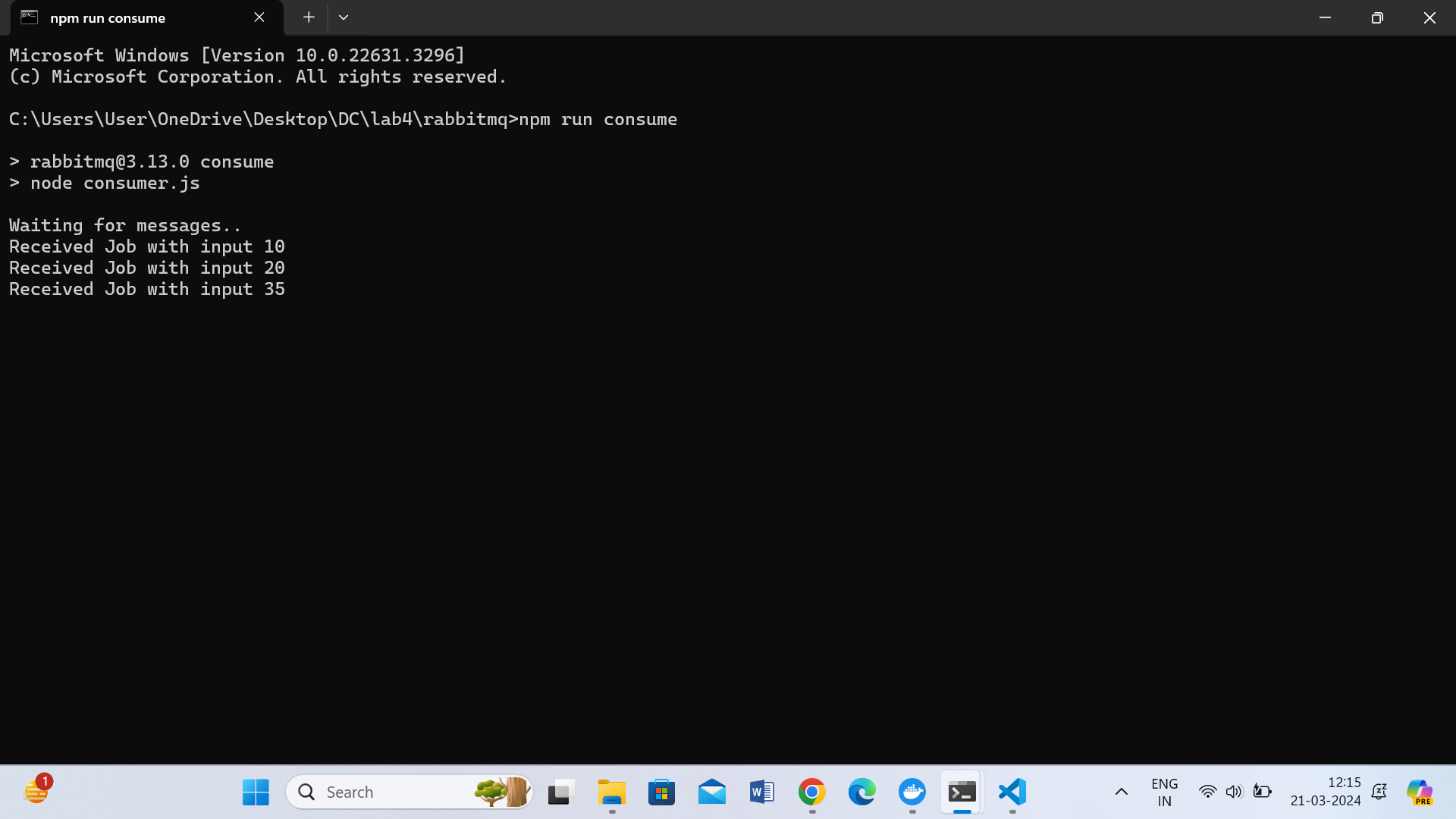


Publish job 35



Running Consumer – consumer.js

All the jobs displayed:



Conclusion:

o Message queueing systems, its need, architecture, and implementation were

understood

o A simple message queueing system was designed and executed using RabbitMQ

message broker.

Postlab Questions:

1.What is message Queueing?

1. **Asynchronous Communication**: Message queuing enables asynchronous communication between components. Senders and receivers do not need to interact directly or be active at the same time. Messages are placed in the queue by the sender and processed by the receiver whenever it's ready.
2. **Decoupling**: Message queuing decouples the producers (senders) and consumers (receivers) of messages. This means that components can send messages to a queue without needing to know the details of who will consume them or when they will be processed.
3. **Reliability**: Message queues often provide mechanisms for ensuring reliable message delivery, such as persistence and acknowledgment mechanisms. Messages are typically stored persistently in the queue until they are successfully processed by a consumer.
4. **Scalability**: Message queuing systems can handle large volumes of messages and scale horizontally to accommodate increased loads. They can distribute messages across multiple queues and consumers to balance the workload.
5. **Fault Tolerance**: Message queuing systems are designed to be fault-tolerant. They can handle failures in producers, consumers, or the messaging infrastructure itself by providing features like message replication, failover mechanisms, and message retry policies.
6. **Integration**: Message queuing is commonly used for integrating disparate systems, allowing them to communicate and exchange data in a standardized and reliable manner. It's often used in microservices architectures, where different services need to communicate asynchronously.

2.What are the benefits of message Queueing?

1. **Asynchronous Communication**: Message queuing allows for asynchronous communication between different components of a system. This means that the sender and receiver do not need to interact with each other in real-time. Asynchronous communication improves system responsiveness and overall performance by reducing blocking and waiting times.
2. **Decoupling of Components**: Message queuing decouples the producers of messages from the consumers. This decoupling allows components to operate independently of each other, making the system more modular and flexible. Changes to one component can be made without affecting other components as long as the message format remains consistent.
3. **Reliability and Fault Tolerance**: Message queuing systems often provide built-in mechanisms for ensuring reliable message delivery, such as message persistence, acknowledgment mechanisms, and message retry policies. These features improve the reliability and fault tolerance of distributed systems by handling failures gracefully and ensuring that messages are not lost.
4. **Scalability**: Message queuing systems can scale horizontally to handle increasing workloads. By distributing messages across multiple queues and consumers, message queuing systems can accommodate growing volumes of messages and scale with the demand. This scalability is essential for applications that need to handle large amounts of data or fluctuating workloads.
5. **Load Balancing**: Message queuing systems can distribute messages evenly across multiple consumers, allowing for load balancing and efficient resource utilization. This helps prevent bottlenecks and ensures that no single consumer is overwhelmed with messages while others remain idle.
6. **Integration and Interoperability**: Message queuing is often used for integrating disparate systems and technologies, allowing them to communicate and exchange data in a standardized and reliable manner. Message queuing systems support various protocols and message formats, making them suitable for heterogeneous environments and enabling seamless interoperability between different systems.
7. **Buffering and Throttling**: Message queues act as buffers, allowing messages to be stored temporarily until they can be processed by the consumer. This buffering capability helps smooth out spikes in message traffic and prevents message loss during periods of high load. Additionally, message queues can be used for throttling message consumption, ensuring that consumers do not overwhelm the system with too many requests at once.