Dynamic workload Ex 2 Failure Modes

1. In the event of manager failures, the managed queues and their data may be lost as the queues are currently stored in the local memory of the EC2 instances.

   To address the issue of preserving queue data and ensuring its availability even in the event of manager failures, you can consider the following solution:
   a. Set up a database or managed queue server: Instead of relying on in-memory queues, store the queues and their data in a persistent database or a managed queue service such as Amazon Simple Queue Service (SQS) or Redis. This will ensure that the queue data is durably stored and can be accessed even if the managers fail.
   b. Modify the worker architecture: Update the worker architecture to directly interact with the queue service or the database. Workers can fetch jobs from the queue and process them independently, regardless of the manager's availability. This way, even if the managers fail, the workers can continue processing the jobs based on the data available in the queue.

2. If one or both of the managers go down, there is currently no mechanism in place to deploy a recovery EC2 instance to replace them.

   To resolve this problem, we recommend implementing a load balancer that can monitor the health of the managers and provide a recovery system. The load balancer would be responsible for detecting failed instances and automatically deploying replacement instances to maintain the system's functionality and availability.

3. In the scenario where the port on the EC2 instance used by the managers is already occupied, there is currently no mechanism in place to validate whether the Flask app can run on an alternative port.

   To address this issue, we suggest implementing a mechanism that can validate and dynamically assign an available port for the Flask app to run on.

4. In our implementation, we have incorporated a mechanism where the worker notifies the manager once it is up and running. Only after receiving this notification does the manager change a flag indicating that additional workers can be deployed if necessary.

However, if a worker fails to notify the manager, the flag remains unchanged. This prevents the manager from scaling up additional workers when required.

To tackle this problem, we have implemented a retry mechanism to minimize the chances of the notification process failing. Alternative approach we suggest to addressing this problem is by implementing a heartbeat mechanism between the worker and the manager.

5. In the event that a worker fails to notify the managers about completing the processing of a job, the data related to that particular job will be lost.

   To address this issue, we suggest retry mechanism: If a worker fails to notify the manager about job completion, the manager can initiate a retry mechanism to periodically check the status of the job. This allows for the detection and recovery of failed notifications.

6. To mitigate the risk of data loss during the scaling down of an EC2 instance responsible for data storage or processing, it is crucial to implement appropriate measures. This can be achieved through proper data replication, backup, or synchronization mechanisms.

   By implementing data replication, copies of the data are created and stored on multiple instances or storage systems. This redundancy ensures that even if an instance is removed during scaling down, the data remains accessible and intact on other instances.

By adopting these suggestions, we can enhance the reliability and resilience of the system, preventing data loss and enabling seamless recovery in case of manager failures.