When I look back at the start of this project, I realize that I had a very simplified mental model of how distributed systems behave. I thought failures would be obvious, metrics would be straightforward to interpret, and debugging would be similar to working on a single machine. This project quickly showed me that almost none of those assumptions were correct, and that building a reliable distributed system means understanding how every component behaves under stress, not just how the code works in isolation.

The biggest lesson for me came from Experiment 3, when I completely misunderstood what was happening during the S3 outage. When we turned off S3 access, the system printed error logs repeatedly, the queue depth kept increasing, and the workers seemed to be “failing” hundreds of times. My first reaction was that our architecture had collapsed. I assumed messages were being dropped or that our workers weren’t stable enough to handle downstream failures.

But after investigating more carefully, reading AWS documentation, checking the visibility timeout, and tracing message lifecycles, I realized that my interpretation was completely wrong. The system was not failing. It was doing exactly what SQS is designed to do:
buffer messages, retry them, and prevent permanent data loss during temporary outages.

The “hundreds of failures” I thought I saw were not actual failures—they were S3 upload attempts during the outage, followed by retries after the visibility timeout expired. No messages were lost. No pods crashed. The queue simply held the workload until S3 came back online. Before this project, I did not appreciate how important the distinction is between an application error and a distributed system retry cycle.

This misunderstanding taught me several things. First, logs alone are not enough to understand system behavior. A line of red text may look like a catastrophic error when it's really part of a normal retry cycle. Second, metrics need context. Queue depth rising is not inherently bad, it may simply reflect a temporary bottleneck downstream. Finally, AWS managed services behave differently than local code, and that behavior must be understood before drawing conclusions from experiments.

Another area where I learned a lot was autoscaling. Before this course, I thought “autoscaling” just meant adding more pods when CPU gets high. Through Experiment 2, I saw that CPU-based scaling reacts slowly compared to queue depth, and that burst traffic can expose delays in HPA’s response time. It made concepts from the lectures—like backpressure, elasticity, and load spikes—much more concrete. If I were to redesign the autoscaling logic, I would probably consider queue-length–based scaling (e.g., KEDA), which reacts faster than CPU metrics.

There were also several things that went wrong technically. For example, early in the project, I assumed that pushing more goroutines inside a worker would automatically improve throughput. In reality, because each worker competes for the same network bandwidth to S3 and SQS, this only increased retries and made logs harder to interpret. It reminded me that concurrency inside a single pod is not a substitute for horizontal scaling, which is a core theme of the course.

I also made mistakes with Terraform at the beginning, especially around IAM roles and permissions. Some failures were caused not by code, but by misconfigured policies. At one point I was stuck because the EKS cluster didn't have the correct role to interact with SQS, and I assumed it was a bug in my code. This taught me how much cloud infrastructure influences distributed system behavior, even when the application logic is correct.

If I were to redo the project, I would be more systematic. I would trace message flow in detail before running tests, I would design better metrics dashboards to distinguish queue-level failures from worker-level failures, and I would set up alerts earlier instead of reading logs reactively. I also would spend more time upfront understanding the semantics of SQS and S3, especially visibility timeout, maxReceiveCount, and retry policies, because these concepts shaped the experiment results more than the application code did.

Overall, this project changed how I think about reliability. I learned that a distributed system is not defined by how it behaves when everything works, but by how it behaves when something breaks: a downstream dependency goes offline, a worker restarts, a message is malformed. The fact that our system continued running, even during failures,and recovered cleanly afterward gave me a much deeper appreciation for the real-world patterns we learned in class. I now understand why systems are designed with DLQs, retries, backpressure, autoscaling, and decoupled components.

It was not always smooth, and some of the debugging sessions were frustrating, but those were also the moments where I learned the most. I walked into the project thinking mainly about code, and I'm walking out thinking about architecture, failure modes, and system behavior over time. That's the biggest growth I experienced, and I think it will stay with me in future work.