Midterm Mastery - Learnings Report

Before taking this course, I thought scaling meant "just add more servers." But through lectures, assignments, and my hands-on experiment, I learned that building reliable distributed systems is much more complex. The most valuable lessons came from understanding **horizontal scaling with load balancing** and **resilience patterns** that prevent system failures. In my experiment, I focused on horizontal scaling, but learning about the three resilience patterns gave me a much deeper understanding of how production systems handle failures.

1. Horizontal Scaling and Stateless Services

- Why Stateless Matters

One of my biggest "aha moments" was understanding stateless services. In a stateful service, each server remembers information about users (like shopping cart data). This creates problems:

- If the server crashes, user data is lost
- Load balancers must send the same user to the same server (sticky sessions)
- Scaling becomes complicated

With stateless services, servers don't remember anything. All user data is stored in a database. This means:

- Any request can go to any server
- If a server fails, just send the request to another server
- Easy to add or remove servers

My Experiment Experience

In my experiment, I deployed a product search service on AWS ECS with different configurations:

- Configuration 1: 2 tasks, 100% minimum healthy
- Configuration 2: 3 tasks, 66% minimum healthy

When I manually stopped tasks to simulate failures, Configuration 1 had 35 seconds of downtime when both tasks failed simultaneously. But Configuration 2 had **zero downtime when tasks failed one by one.** This proved that horizontal scaling with proper redundancy works!

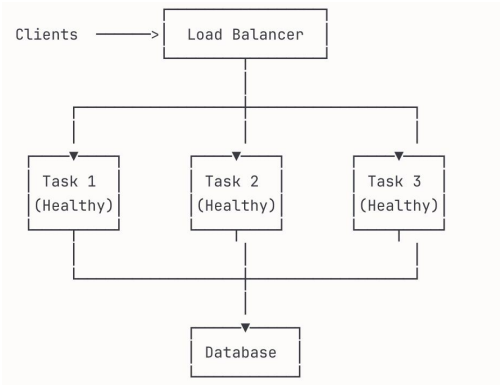
The key insight: The **60-90 second vulnerability window**. After one task fails, it takes time for a replacement to start. If another task fails during this window, the whole system goes down. Having 3 tasks instead of 2 eliminates this timing problem.

- Load Balancer's Role

The **Application Load Balancer (ALB)** is critical for horizontal scaling:

- **Health checks:** Detects when a task is unhealthy
- **Traffic distribution:** Sends requests evenly to all healthy tasks
- **Automatic failover:** Stops sending traffic to failed tasks

Diagram: Horizontal Scaling Architecture



If Task 2 fails:
- Load balancer detects failure through health checks
- Traffic automatically goes to Task 1 and Task 3
- New Task 4 starts automatically (if min healthy % requires it)
- No downtime for users!

- What My Experiment Taught Me

Before the experiment, I thought:
- More servers = better reliability (partially true)
- Redundancy just means "run 2 copies" (not enough!)

After the experiment, I learned:
- **Timing matters:** The **60-90 second window** between failure and recovery is dangerous
- **Over-provisioning is necessary:** N+1 redundancy (3 tasks instead of 2) prevents timing-dependent failures
- **Proactive vs reactive:** The 66% minimum healthy threshold is proactive – it prevents dangerous states. Auto-scaling alone is reactive – it fixes problems after they happen.
- **Cost-benefit trade-off:** Adding one task costs 50% more (~$15/month) but eliminates downtime for realistic failure scenarios

2. Three Resilience Patterns for Microservices

While my experiment focused on horizontal scaling, I learned three important patterns from class that explain how production systems handle more complex failure scenarios. These patterns are like "safety features" that prevent small problems from becoming big disasters.

Pattern 1: Timeout / Fail Fast

The Problem: Imagine calling a service that usually responds in 100ms, but suddenly it becomes very slow and takes 30 seconds. Without timeouts, your application threads wait for 30 seconds. If many requests come in, all your threads get stuck waiting. Eventually, you run out of threads and your whole application stops working.

The Solution: Set a **maximum wait time**. If a request takes too long, give up and return an error immediately.

Why it matters: Slow services are more dangerous than dead services. With a dead service, you get an error immediately. With a slow service, you waste time and threads. **Timeouts solve this problem by treating "too slow" the same as "dead."**

Pattern 2: Circuit Breaker

The Problem: If a downstream service keeps failing, why keep calling it? Each failed request wastes threads and time. It's like calling a friend whose phone is dead – after 10 tries, you should stop and try again later.

The Solution: Track failures. After too many failures, "open the circuit" – stop calling the service temporarily. After some time, try again to see if it recovered.

Three states:
1. CLOSED (normal): Requests go through normally
2. OPEN (broken): Block all requests immediately, return error or cached data
3. HALF-OPEN (testing): Try a few requests to see if service recovered

Real-world analogy: Your favorite coffee shop. If it's closed 5 times in a row, you stop going there for a week instead of checking every single day.

Pattern 3: Bulkhead

The Problem: Imagine your application calls three services: Products, Customers, and Orders. All requests share the same 100-thread pool. If the Products service becomes very slow, it uses all 100 threads. Now Customer and Order requests also fail, even though those services are working fine! One slow service kills everything.

The Solution: Create separate thread pools for each service. Like bulkheads in a ship – if one compartment floods, others stay safe.

Why it matters: This pattern prevents cascading failures. One slow or broken service can't bring down your entire application.

3. Connecting the Concepts

What I found interesting is how these patterns work together:

Horizontal Scaling gives you redundancy – multiple copies of your service. If one fails, others keep working. This is what I tested in my experiment.

The Three Patterns protect against different types of failures:
- Timeouts: Protect against slow services
- Circuit Breakers: Protect against repeatedly calling broken services
- Bulkheads: Protect against one service affecting others

In my experiment, the **ALB health checks** were actually using timeouts behind the scenes – if a task didn't respond quickly enough, it was marked unhealthy. The minimum healthy percentage (66%) acted like a bulkhead at the infrastructure level – **it kept some capacity reserved so the system could handle failures.**

4. Real-World Implications

In production systems, failures are complex:
- Memory leaks cause slow, cascading failures (need timeouts + circuit breakers)
- Network issues happen randomly (need horizontal scaling)
- One microservice going down shouldn't kill everything (need bulkheads)
- Deployments and updates need rolling restarts (need minimum healthy thresholds)

The lessons I learned show that **reliability is not about preventing failures** – it's about designing systems that **work DESPITE failures**. You need multiple layers of protection:

- Horizontal scaling for redundancy
- Stateless services for easy failover
- Resilience patterns for graceful degradation

5. Conclusion

This course transformed my understanding of distributed systems. The most valuable learning came from combining theory with practice. I read about these concepts in slides, implemented horizontal scaling in my experiment, and now understand how all the pieces fit together.

My biggest takeaway: Building reliable systems requires thinking about failures at every level – from infrastructure (how many tasks?) to application code (timeouts, circuit breakers, bulkheads). My experiment proved that proper redundancy (3 tasks instead of 2) eliminates downtime for realistic failures. The resilience patterns I learned show how to handle even more complex failure scenarios in production systems.

This mindset shift – from "preventing failures" to "working despite failures" – will guide how I design systems in my future career.