Part1 — Topics Enjoyed Learning

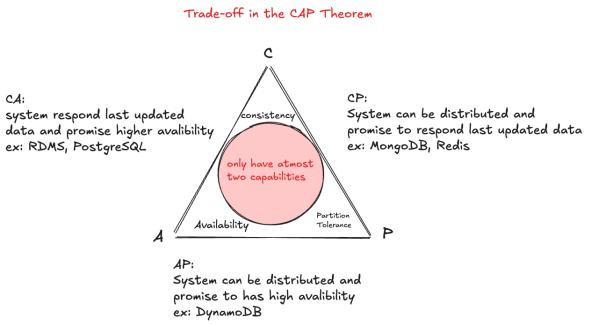
1. Understanding the CAP Theorem

One of the most eye-opening moments in this course was learning the **CAP Theorem**—the simple but powerful idea that no distributed system can simultaneously provide **Consistency**, **Availability**, and **Partition tolerance**. This concept, first introduced in *Day 2 Intro and Concurrency*, fundamentally reshaped how I think about system design.

In class we used a triangle diagram showing the three properties. Before this course I used to assume that "good systems" should just guarantee everything. But the CAP discussion helped me see why engineers must *choose which two matter most* for a given use case. For instance, a banking application must favor **Consistency + Partition tolerance (CP)**—it cannot lose a transaction even if some replicas are slow—while an e-commerce catalog or social feed might prefer **Availability + Partition tolerance (AP)** so that users can keep browsing even if some nodes are out of sync.

My mental model (see Figure 1) is a simple **CAP triangle**, with arrows showing real-world examples at each edge. Thinking through these trade-offs made me realize that architecture is not only about technology but about *business priorities*—whether we value speed or accuracy when the network misbehaves. I now read every system diagram through that lens.

(Figure 1: CAP triangle showing trade-offs and example systems – CP databases vs AP caches)



---

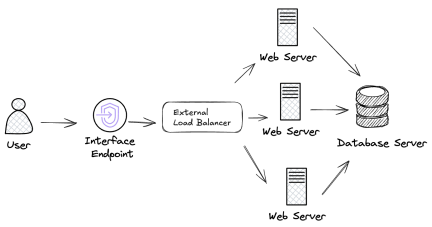
2. Load Balancing and Scaling in Distributed Systems

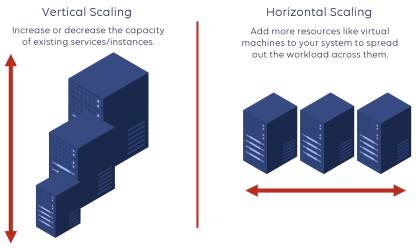
Another topic I deeply enjoyed was learning how systems actually scale to millions of users through **load balancing and elastic scaling**. The lectures on *Day 5 APIs and Load Balancers* and *Scaling Applications by Peter Smith* made the abstract idea of "scalability" feel concrete.

We explored several **load-balancing algorithms**—*Round Robin (RR)*, *Weighted Round Robin (WRR)*, and *Smooth Weighted Round Robin (SWRR)*—each distributing traffic a little more intelligently than the last. I found it fascinating that such small algorithmic differences can significantly improve throughput and fairness. The concept of **health checks** and **stateless services** (where any node can handle any request) made me appreciate the elegance of decoupling computation from state. It prevents single points of failure and makes horizontal scaling possible.

From there we moved into **horizontal vs vertical scaling**. *Horizontal scaling (Scale Out)* adds more nodes; *vertical scaling (Scale Up)* makes one node bigger. My "aha" moment came when I realized that vertical scaling always hits a physical limit, while horizontal scaling demands smart **data partitioning** and **replication** to avoid bottlenecks. In my notebook I drew the model shown in Figure 2—adapted from my own mind-map—showing a load balancer distributing requests across multiple stateless services, with replicas maintaining redundancy and partitions storing different key ranges.

(Figure 2: My mental model of load balancing and scaling – LB → N nodes with horizontal scaling and data partitions)





Working through the AWS Labs and Terraform + Fargate assignments made these ideas real. Seeing the **Application Load Balancer (ALB)** in action, routing traffic based on HTTP headers, and watching **auto scaling** trigger when CPU usage passed 70% gave me tangible proof that the concepts truly work. I enjoyed plotting CloudWatch metrics and realizing that every data point represented decisions made by those algorithms we studied.

This topic changed my mindset: scalability is not just "adding more servers," it is a carefully orchestrated balance of traffic routing, state management, and elasticity.

---

3. Circuit Breaker Pattern for System Resilience

The topic that tied everything together for me was **system resilience**, especially the **Circuit Breaker Pattern** from *Day 6 Sneaking Up on Microservices* and my own *Part II experiment*. After building microservices for an e-commerce example, I saw firsthand how one small failure could cascade through the system—exactly as shown in my project: the *Product Service* crashed, the *Shopping Cart* began throwing errors, and eventually even the *Recommendation Service* failed.

Implementing a **circuit breaker** transformed that chaos into controlled behavior. Acting as a proxy, it monitored success and failure rates. When too many errors occurred, it *"tripped"* into the **open state**, immediately rejecting new requests instead of repeatedly hitting the dead service. After a timeout, it moved into **half-open**, allowing limited test requests, and if they succeeded, it closed again.

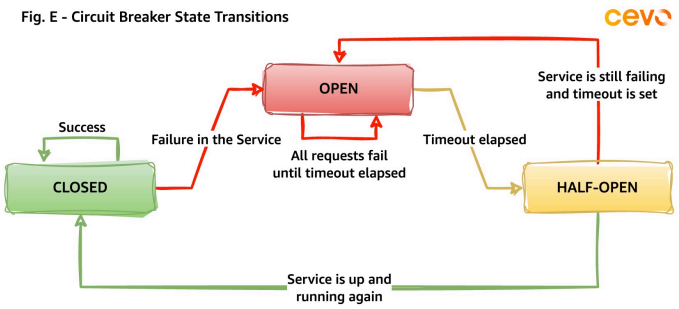
The lecture compared it to an electrical circuit: when a power surge happens, the breaker opens to protect appliances. The same principle applies to software. I even implemented a simplified version where the *Shopping Cart* service returned *"Product information temporarily unavailable"* instead of crashing outright. Watching the service recover automatically after the timeout felt magical—the system was now **self-healing**.

My **mental model** (Figure 3) depicts this clearly:

- **Closed state** → normal traffic
- **Open state** → fail-fast, no calls
- **Half-open state** → test recovery

This topic helped me connect many dots: the CAP Theorem showed the limits of perfect consistency; load balancing and scaling showed how we design for availability; and circuit breakers showed how we recover gracefully when things inevitably fail.

(Figure 3: Circuit Breaker State Machine – Closed → Open → Half-Open → Closed)



Reflection

Across these topics, what I enjoyed most was realizing that distributed systems are not just about code—they are about **trade-offs** and **resilience**. CAP taught me why perfection is impossible. Load balancing taught me how to engineer around physical limits. Circuit breakers taught me how to accept failure without collapse.

In the beginning of the semester, I tended to think of "system crashes" as disasters caused by bad code. Now I see them as normal, expected events in complex distributed environments. The goal is not to prevent every crash, but to ensure that the system *recovers quickly and gracefully*. That shift in mindset is the biggest lesson I have taken from this course so far.
