What I've Found Valuable in CS 6650 So Far

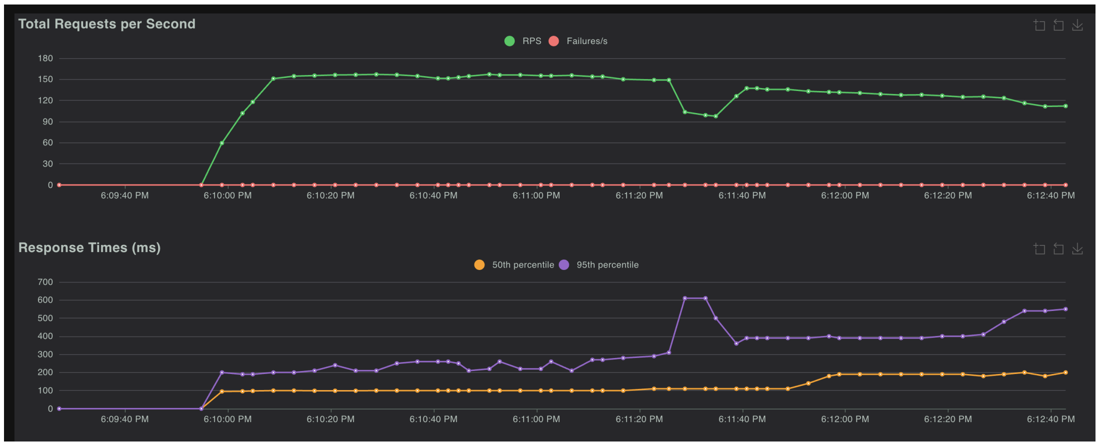
1. Thinking in Trade-offs: CAP, Latency, and Availability

Chapter 1 of *Distributed Systems for Fun and Profit* opened the course with a deceptively simple observation: *if you can't affect performance, you can't improve it; if you can't measure it, you can't manage it.* That framing, distributed systems as a sequence of deliberate trade-offs between latency, availability, and partition tolerance, became the lens through which I read everything else. Chapter 2 sharpened this further with the CAP theorem and the FLP impossibility result. FLP in particular was striking: it proved that in an asynchronous network with even a single faulty process, no deterministic consensus algorithm can guarantee termination. This isn't a limitation of engineering skill; it is a mathematical boundary on what distributed systems can do. The Google infrastructure overview reinforced how these theoretical constraints map to practice: Google's architecture separated storage, compute, and coordination precisely because each dimension has a different failure mode and a different scaling strategy.

These ideas crystallised in HW6 during the resilience test. Manually stopping an ECS task mid-load-test and watching the ALB continue routing traffic is a hands-on demonstration of the availability side of CAP: the system chose to remain available (near-zero failures in Locust) and accepted a brief window of reduced capacity while ECS recovered. A CP system would have stalled until the new task passed its health check.

2. Concurrency, Race Conditions, and the Cost of Shared State (HW3)

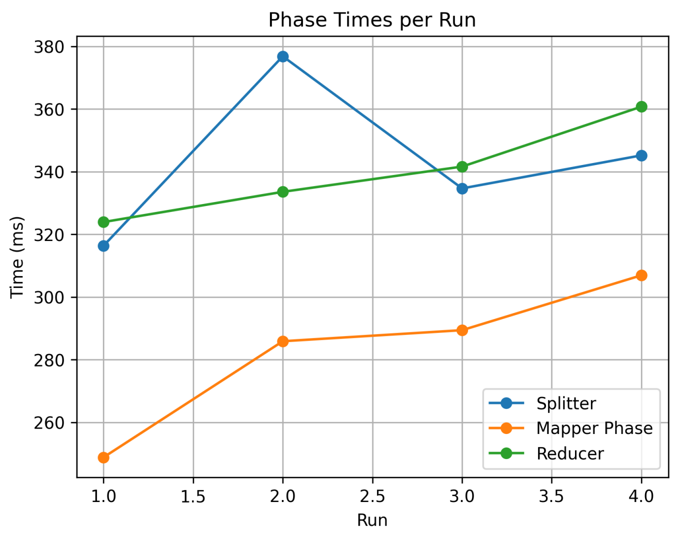
HW3 was where abstract concurrency theory became viscerally real. Running 50 goroutines against a plain **int64** counter consistently produced wrong answers, 29,259 instead of 50,000, because the read-modify-write sequence is not atomic. Go's -race flag identified the exact memory address and the exact goroutines involved, making the invisible visible. The progression from atomic operations to Mutex, RWMutex, and sync.Map taught me that *correctness costs something*, and the right primitive depends on access pattern: sync.Map halved latency versus a plain Mutex in my write-heavy test, but RWMutex offered no benefit because there were almost no reads. The Locust chart below from that homework shows the consequence at the API level: as concurrency increased, response time percentiles diverged, p95 rising faster than p50, because contention on shared state created serialisation points, exactly as Amdahl's Law predicts.



HW3: Locust RPS and response-time percentiles (1-worker vs 3-worker). p95 diverges from p50 under higher concurrency due to shared-state contention.

3. MapReduce: Pipelines, Bottlenecks, and the Parnas Principle (HW4)

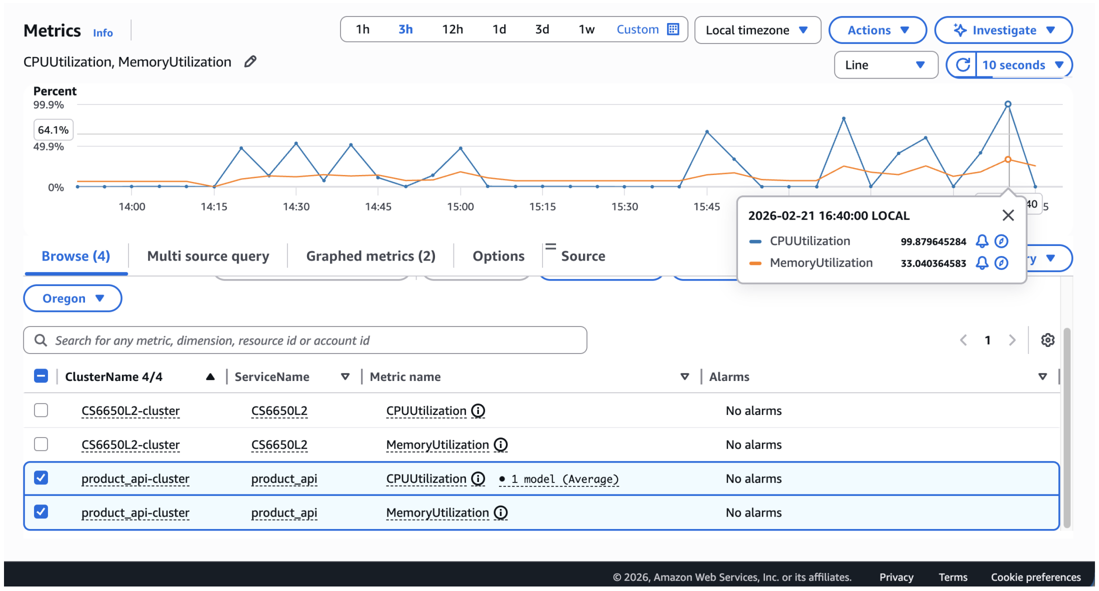
The Dean & Ghemawat MapReduce paper was the first research paper that made me re-examine how I decompose problems. The key insight is not just parallelism, it is that a rigid **Split -> Map -> Reduce** interface hides all internal implementation decisions from the caller. This connects directly to Parnas's 1972 paper *"On the Criteria to be Used in Decomposing Systems into Modules,"* which argues that the right decomposition boundary is one that hides a design decision, a changeable assumption, from the rest of the system. In MapReduce, the Reduce phase hides the aggregation strategy; you can swap the algorithm without touching the Mapper. HW4 made this concrete: I measured each phase independently and found that the Reducer grew as a bottleneck across runs (green line climbing in the chart below) while the Mapper remained relatively flat. This is classic Amdahl: the Reduce phase is a sequential aggregation point that no amount of Mapper parallelism can eliminate.



HW4: Phase times per run. Reducer latency grows monotonically, revealing it as the pipeline bottleneck, the serial fraction Amdahl's Law predicts.

4. Measurement-Driven Infrastructure Decisions (HW5 & HW6)

The most practically useful skill this course has built is the habit of *measuring before concluding*. In HW6 I ran six progressive stress tests (500 -> 3,000 concurrent users). The CloudWatch screenshot below captures the key moment: CPU hit 99.87% while memory sat at 33%. That single dashboard view, two lines, one near the ceiling and one near the floor, answered the question of what to scale. Without it, over-provisioning memory would have been an easy and expensive mistake. This is the operational version of what Chapter 1 describes as understanding your system's bottleneck before optimising.



CPUUtilization, MemoryUtilization

2026-02-21 16:40:00 LOCAL

CPUUtilization	99.879645284
MemoryUtilization	33.040364583

ClusterName 4/4	ServiceName	Metric name	Alarms
CS6650L2-cluster	CS6650L2	CPUUtilization	No alarms
CS6650L2-cluster	CS6650L2	MemoryUtilization	No alarms
product_api-cluster	product_api	CPUUtilization • 1 model (Average)	No alarms
product_api-cluster	product_api	MemoryUtilization	No alarms

© 2026, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

HW6 Test 6 (3,000 users, 200 req/s): CPU peaks at 99.87%, memory stays at 33%. Throughput had already plateaued at ~2,200–2,300 RPS, a textbook CPU saturation signature.

Adding an Application Load Balancer and a target-tracking auto-scaling policy (trigger: 70% CPU, min 2 / max 4 tasks) resolved the bottleneck. Under the same test, ECS scaled from 2 to 3 tasks, CPU per task dropped, and throughput recovered. What I appreciated about this experiment is that it closed a loop that starts in the readings: the Google infra blog describes horizontal scaling across commodity machines as the foundational Google strategy precisely because vertical scaling has a hard ceiling and a linear cost curve. Running that strategy on ECS at student scale made the principle feel real rather than abstract.

The common thread across all of this is that theory and measurement reinforce each other. CAP and FLP tell you what is impossible; CloudWatch and Locust tell you where you actually are. Parnas tells you how to draw module boundaries so that failure is contained; ECS and the ALB implement exactly that boundary at the infrastructure level. I came into this course thinking of distributed systems as a collection of techniques. I'm leaving the first half thinking of it as a discipline for reasoning under uncertainty.