CS6650

Midterm Reflection

Huining Yu

1. MapReduce

* Idea: Split a big dataset across workers. Each worker does a Map (emit key/value pairs), framework does a Shuffle (group by key), then workers do a Reduce (aggregate).
* Why it’s important: Teaches data parallelism, data locality, and coordination at scale. Most modern big-data engines (Spark, Flink, Beam) generalize this model.

2) Terraform

* What it is: Declaratively describe cloud resources. Terraform computes a plan (diff) and applies it to converge reality to your code.
* Core concepts: Providers, resources, data sources, modules, state, workspaces.
* Why it clicked: Reproducibility. Instead of clicking in consoles, you version your infra—great for reviews and rollbacks.

3) Docker

* What it is: Containerize apps so “works on my machine” becomes “works anywhere.”
* Core concepts: Image (layers), Container (running instance), Registry (ECR/Docker Hub), Networks, Volumes, docker-compose for local multi-service.
* Why it’s especially useful: You get reproducible dev envs, fast CI/CD, and a frictionless handoff to orchestration (ECS/Kubernetes).

4) AWS: ECR + ECS (New & Practical)

* ECR (Elastic Container Registry): Private image store. docker login with AWS creds, docker tag, docker push.
* ECS (Elastic Container Service): Runs containers via:
  + Fargate (serverless—no EC2 to manage) or EC2 (you manage instances).
  + Task Definition (container spec), Service (desired count + rolling deploy), Cluster (capacity).
* Why it’s fun/practical: Deploy the same container you run locally; add Auto Scaling, ALB for traffic, and CloudWatch for logs/metrics.
* First-deploy checklist:
  + docker build → push to ECR.
  + Create Task Definition (CPU/RAM, port, health check).
  + Create Service on Fargate, attach ALB target group.
  + Set Auto Scaling on CPU or ReqCount.

5) Threading & Load Testing (Locust)

* Threading vs. async: Threads are simple but can waste CPU when I/O-bound; async scales better for I/O heavy microservices. Use semaphores to cap concurrency.
* Locust: useful tool for load test

6) Microservices

* Why microservices: Independent deployability and scaling; technology choice per service; fault isolation.
* Costs: Distributed tracing, eventual consistency, more failure modes (network, timeouts), harder debugging.
* Resilience patterns:
  + Fail-Fast (backpressure via queue caps; return 503 early)
  + Bulkhead (cap concurrency to slow dependencies)
  + Timeouts + Retries (with jitter/backoff; only when safe)
  + Circuit Breaker (temporarily stop calling a sick downstream)