Homework 4: ECR/ECS/Fargate Deployment and MapReduce Implementation

Part I: reading

comments on piazza

Part II: Infrastructure Setup with ECR/ECS/Fargate

Overview

In Part II, I successfully deployed a containerized web service to AWS using ECR, ECS, and Fargate. This assignment demonstrated the advantages of managed container orchestration over manual deployment methods.

Implementation Steps

1. **ECR Setup**: Created a private container registry and pushed a Docker image containing a simple Go web service (album API)
2. **ECS Cluster**: Created a Fargate-based cluster for serverless container execution
3. **Task Definition**: Defined container specifications including CPU (0.25 vCPU), memory (0.5 GB), and IAM roles (LabRole)
4. **Deployment**: Successfully ran the task and accessed the service via its public IP address
5. **Testing**: Verified functionality by querying `/albums` and `/albums/1` endpoints

Key Learnings

- **Scalability**: Unlike manual deployment, ECS makes it trivial to scale from 1 to N instances
- **Automation**: The ECR/ECS workflow eliminates repetitive manual setup tasks
- **Infrastructure as Code**: Task definitions serve as reproducible deployment blueprints
- **Networking**: Understanding VPCs, subnets, and security groups is essential for container networking

---

Part III: MapReduce Implementation with ECS and S3

Overview

I implemented a simplified MapReduce system using three microservices (Splitter, Mapper, Reducer) deployed on ECS with S3 for intermediate data storage. The system successfully processed Shakespeare's Hamlet (159KB text file) to perform word frequency analysis.

Architecture

```
 Input File (S3)
    ↓
Splitter Service (ECS Task)
    ↓
3 Chunks (S3)
    ↓
3 Mapper Services (Parallel ECS Tasks)
    ↓
3 Result JSONs (S3)
    ↓
Reducer Service (ECS Task)
    ↓
Final Result (S3)
```

Implementation Details

1. Splitter Service

- Input: Large text file from S3
- Processing: Splits file into 3 equal chunks by line count
- Output: 3 chunk files uploaded to S3
- Result: Hamlet.txt (159KB) → chunk1.txt (51.7KB), chunk2.txt (55.8KB), chunk3.txt (51.6KB)

2. Mapper Service

- Input: Individual chunk file from S3
- Processing: Counts word frequency using Go map data structure
- Output: JSON file with word counts

- Deployed: 3 parallel instances, each processing one chunk

3. Reducer Service

- Input: 3 result JSON files from mappers
- Processing: Merges word counts by summing values for duplicate keys
- Output: Final consolidated word frequency
- Result: Successfully identified "the" (993 occurrences) as the most frequent word

Deployment Process

All services were containerized using Docker, pushed to separate ECR repositories, and deployed as ECS task definitions. Tasks were triggered manually via AWS CLI with command-line arguments specifying S3 URLs for input/output files.

Reflections on Distributed Systems Questions

1. What happens if one of the mappers fails? How would you recover?

Current State: In my implementation, if one mapper fails, the entire MapReduce job becomes incomplete because the reducer expects exactly 3 result files.

Recovery Strategies:

- **Retry Logic**: ECS task definitions support automatic restart policies. Configure `maximumRetryAttempts` to automatically retry failed tasks
- **Dead Letter Queue**: Monitor failed tasks and trigger alerts for manual intervention
- **Idempotency**: Design mappers to be idempotent so retrying the same chunk produces identical results
- **Checkpointing**: Store task status in DynamoDB; a coordinator service could detect failures and relaunch specific mappers
- **Redundancy**: Launch 2 mappers per chunk and use the first one to complete

2. How can you scale this system to 10 or 100 mappers?

Horizontal Scaling Approach:

For 10 Mappers:

- Splitter divides file into 10 equal chunks
- Launch 10 mapper tasks simultaneously
- Reducer processes 10 result files
- ECS Fargate can handle this easily within service quotas

For 100 Mappers:

- **Challenge 1**: ECS task launch rate limits (~100-500 per minute per region)
  - **Solution**: Batch task launches or use ECS Service with desired count
- **Challenge 2**: S3 request rate limits
  - **Solution**: Use S3 prefixes to distribute load; S3 automatically scales to 3,500 PUT and 5,500 GET requests per second per prefix
- **Challenge 3**: Manual coordination becomes impractical
  - **Solution**: Implement an orchestration service (coordinator) that:
    1. Launches splitter
    2. Waits for completion
    3. Discovers chunk count dynamically
    4. Launches N mappers in parallel
    5. Monitors completion
    6. Launches reducer with all result URLs

Alternative: Use AWS Step Functions to orchestrate the workflow with built-in retry, error handling, and parallelization.

3. What was the challenging part of coordinating tasks manually?

Key Challenges Experienced:

1. Asynchronous Task Monitoring

- ECS tasks don't block the terminal; you must poll for status
- Had to manually check S3 for output files to know when tasks completed
- No built-in notification when a task finishes

2. Parameter Passing

- Constructing JSON for `--overrides` with nested command arrays was error-prone
- Easy to make typos in S3 URLs, leading to silent failures
- Had to remember correct S3 paths for each stage

3. Sequential Dependencies

- Must wait for splitter to complete before knowing chunk filenames
- Must wait for all 3 mappers to complete before starting reducer
- Manual timing estimation required (wait 30-60 seconds, then check)

4. Error Handling

- When a task failed, had to check CloudWatch Logs manually
- No automatic rollback or cleanup

- Debugging required navigating multiple AWS Console pages

5. Resource Management

- Easy to forget running tasks, potentially incurring costs
- Manual cleanup of S3 files, ECR images, ECS tasks, and clusters
- VPC/Subnet variables lost between terminal sessions

What Would Improve This:

- **Orchestration Layer**: AWS Step Functions or Apache Airflow
- **Infrastructure as Code**: Terraform or CloudFormation for reproducibility
- **CI/CD Pipeline**: Automate build → push → deploy workflow
- **Monitoring Dashboard**: Centralized view of all task statuses
- **Scripting**: Bash or Python wrapper scripts to automate the full pipeline

---

Performance Analysis

Distributed Processing Benefits

File Details:

- Input: Shakespeare's Hamlet (159KB, ~5,500 lines)
- Processing: Word frequency counting

Theoretical Speedup:

- Single-threaded: Process entire file sequentially
- Distributed (3 mappers): Process 3 chunks in parallel
- Expected speedup: ~3x (assuming mapper is CPU-bound and chunks are equal size)

Actual Observations:

- Chunk sizes: 51.7KB, 55.8KB, 51.6KB (well-balanced)
- All 3 mappers completed within approximately the same time window
- Overhead: Task startup time (~20-30 seconds per task) dominates for small files
- For larger files (multi-GB), the parallel speedup would be more significant

Bottlenecks Identified:

1. **Task Startup Time**: Fargate cold start takes 20-30 seconds
2. **Network I/O**: Downloading chunks from S3 and uploading results
3. **Sequential Stages**: Splitter and Reducer are not parallelizable

Optimization Opportunities:

- Use smaller container images to reduce pull time
- Keep tasks warm with periodic health checks
- Increase chunk count for larger files to maximize parallelism
- Stream processing instead of downloading entire chunks

Key Takeaways

Technical Skills Gained

1. **Container Orchestration**: Hands-on experience with ECS task definitions, clusters, and Fargate
2. **AWS Services Integration**: ECR, ECS, S3, VPC, IAM working together
3. **Microservices Architecture**: Building independent, single-purpose services
4. **Distributed Computing**: Understanding MapReduce paradigm and data parallelism
5. **Infrastructure Management**: Manual vs automated deployment tradeoffs

Distributed Systems Concepts

- **Data Locality**: S3 as shared storage eliminates data transfer between services
- **Fault Tolerance**: Current system lacks it; would need retry logic and redundancy
- **Scalability**: Easy horizontal scaling but coordination complexity increases
- **Trade-offs**: Simplicity vs robustness, manual control vs automation

What I Would Do Differently

1. **Orchestration**: Use AWS Step Functions to automate the workflow
2. **Monitoring**: Set up CloudWatch alarms and centralized logging
3. **Error Handling**: Implement retry logic and dead letter queues
4. **Testing**: Create smaller test files for faster iteration
5. **Documentation**: Script the entire process for reproducibility

---

Conclusion

This assignment provided valuable hands-on experience with cloud-native distributed computing. Part II demonstrated the power of managed container services for eliminating operational overhead. Part III illustrated both the potential and challenges of distributed data processing.

The MapReduce implementation, while simplified, successfully processed real-world data (Shakespeare's Hamlet) and achieved the core goal of parallel processing. The manual coordination highlighted the need for orchestration tools in production systems.

Most importantly, this exercise reinforced that distributed systems are fundamentally about trade-offs: between complexity and scalability, between automation and control, and between development time and operational robustness.

The ability to deploy services to ECS and coordinate distributed tasks is a valuable skill applicable to real-world cloud engineering scenarios.

Appendix: Final Results

Most Frequent Words in Hamlet:

```
 "the": 993 occurrences
"and": 862 occurrences
"to": 683 occurrences
"you": 522 occurrences
"of": 361 occurrences
"that": 361 occurrences
"a": 497 occurrences
"i": 338 occurrences
"my": 319 occurrences
"in": 270 occurrences
```

Total Unique Words: 4,218 **Total Word Count**: ~32,000 words

The system successfully demonstrated distributed processing by splitting the workload across 3 parallel mappers and consolidating results into a single comprehensive word frequency analysis.