PART A: TECHNICAL DESIGN DOCUMENT

ML Production System for App Similarity & Performance Prediction

MobUpps Home Assignment

Author: Ortal Lasry | ML Engineer

Date: October 2025

Version: 1.0

1. SYSTEM ARCHITECTURE OVERVIEW

1.1 High-Level Architecture

The system is designed as a microservices architecture on AWS, capable of scaling from 100 to 1M requests/day with minimal latency (<200ms p99).

Key Components:

• API Gateway: Entry point for all client requests
• ECS Fargate: Containerized application hosting (auto-scaling)
• ElastiCache Redis: Embedding cache + session storage
• S3: Data lake for embeddings, metadata, and performance logs
• DynamoDB: Real-time metrics and A/B test assignments
• CloudWatch: Monitoring, logging, and alerting
• SageMaker: Model retraining and versioning
• Lambda: Event-driven processing (feedback loop)
• Step Functions: Orchestration for model deployment pipeline

1.2 Data Flow

[Client Request]
```
-> API Gateway (Rate limiting, auth)
-> Application Load Balancer
-> ECS Fargate (Similarity Service)
-> ElastiCache (Check embedding cache)
-> If miss: S3 (Load embeddings)
-> DynamoDB (Log A/B assignment + metrics)
-> Return similar apps + predictions
<- Response to Client
```

[Feedback Loop]

```
-> CloudWatch Logs (Capture predictions + outcomes)
-> Kinesis Data Streams (Real-time event processing)
-> Lambda (Aggregate performance metrics)
-> S3 (Store training data)
-> SageMaker (Model retraining trigger)
-> Model Registry (Version new embeddings)
-> Step Functions (Orchestrate deployment)
```

1.3 Architecture Diagram Description

Layer 1 - Client Layer:

```
- Mobile Apps / Web Clients
- SDK/API Integration
```

Layer 2 - Edge Layer (AWS CloudFront + API Gateway):

```
- CloudFront: CDN for static assets + edge caching
- API Gateway: REST API with request validation
- WAF: DDoS protection + rate limiting
- Cognito: Authentication (if needed)
```

Layer 3 - Application Layer (ECS Fargate):

```
- Cluster: Auto-scaling (2-50 tasks)
- Service 1: Similarity API (FastAPI containers)
- Service 2: Prediction API (FastAPI containers)
- Load Balancer: ALB with health checks
- Target Groups: Blue/Green deployment support
```

Layer 4 - Data Layer:

```
- ElastiCache Redis:
* Embedding cache (TTL: 1 hour)
* A/B test assignments (sticky sessions)
* Rate limiting counters
```

```
- DynamoDB:
* Metrics table (real-time aggregations)
* A/B test results (partition by experiment_id)
* Request logs (time-series data)
- S3:
* Embeddings bucket (v1, v2, v3...)
* Historical data bucket (training datasets)
* Logs bucket (CloudWatch exports)
```

Layer 5 - ML Pipeline (SageMaker + Step Functions):

```
- SageMaker Training: Model retraining jobs
- SageMaker Endpoint: Real-time inference (if needed)
- Model Registry: Versioned embeddings
- Step Functions: Deploy workflow automation
```

Layer 6 - Monitoring & Observability:

```
- CloudWatch Logs: Structured logging
- CloudWatch Metrics: Custom metrics (latency, accuracy)
- CloudWatch Alarms: Threshold-based alerts
- X-Ray: Distributed tracing
- SNS: Alert notifications (email, Slack, PagerDuty)
```

2. AWS SERVICES SELECTION & JUSTIFICATION

2.1 Core Services

• Amazon ECS Fargate** (Application Hosting)

Why:

• Serverless containers (no EC2 management)
• Auto-scaling based on CPU/memory/custom metrics
• Pay per vCPU-second (cost-efficient for variable load)
• Blue/Green deployments with ALB
• Integrates with CloudWatch for observability
Alternative Considered: EKS (more complex), Lambda (cold start issues)

• Amazon API Gateway** (API Management)

Why:

• Built-in rate limiting (10,000 req/sec burst)

• Request/response validation

• API versioning support

• AWS WAF integration for security

• Usage plans for different client tiers

Alternative Considered: ALB only (lacks API management features)

• Amazon ElastiCache Redis** (Caching + Session Storage)

Why:

• Sub-millisecond latency for embedding lookups

• Cluster mode for horizontal scaling

• Automatic failover (Multi-AZ)

• Supports complex data structures (hashes for embeddings)

• TTL for cache invalidation

Alternative Considered: DynamoDB DAX (higher cost), Memcached (no persistence)

• Amazon DynamoDB** (Real-time Metrics + A/B Assignments)

Why:

• Single-digit millisecond latency at scale

• Auto-scaling (pay per request)

• Global Secondary Indexes for querying by experiment_id

• TTL for automatic data cleanup

• DynamoDB Streams for change data capture

Alternative Considered: RDS Aurora (more expensive, over-engineered)

• Amazon S3** (Data Lake)

Why:

• Unlimited storage capacity

• 99.999999999% durability

• Lifecycle policies (move to Glacier after 90 days)

• S3 Select for querying CSV/Parquet

• Event notifications (trigger Lambda on new embeddings)

Alternative Considered: EFS (higher cost for large datasets)

• Amazon SageMaker** (Model Management)

Why:

• Model Registry for versioning

• Training jobs with spot instances (70% cost savings)

• Batch transform for bulk predictions

• Feature Store for metadata management

• Pipelines for end-to-end ML workflows

Alternative Considered: Self-managed MLflow (operational overhead)

• AWS Step Functions** (Orchestration)

Why:

• Visual workflow designer

• Error handling and retries

• Integration with all AWS services

• Audit trail for deployments

• State machine versioning

Alternative Considered: Airflow on EC2 (requires maintenance)

2.2 Monitoring & Observability

• Amazon CloudWatch** (Logging + Metrics + Alarms)

Why:

• Native AWS integration (no agents needed)

• Log Insights for querying structured logs

• Custom metrics (business KPIs)

• Anomaly detection using ML

• Dashboard for real-time monitoring

Cost: ~$0.50/GB ingested + $0.03/GB analyzed

• AWS X-Ray** (Distributed Tracing)

Why:

• Trace requests across microservices

• Identify bottlenecks (slow S3 calls, Redis misses)

• Service map visualization

• Integration with CloudWatch ServiceLens

Cost: ~$5 per million traces

• Amazon SNS** (Alerting)

Why:

• Fan-out to multiple channels (email, Slack, PagerDuty)
• Message filtering
• Mobile push notifications
• Dead-letter queue for failed deliveries

Cost: ~$0.50 per million notifications

3. A/B TESTING STRATEGY

3.1 Traffic Routing Strategy

• Approach: Hash-Based Sticky Sessions**

Implementation:

1. Hash partner_id + app_id using MD5
2. Map hash to [0, 100] range
3. Route based on configured split (e.g., 50/50)
4. Store assignment in Redis (TTL: 24 hours)

Benefits:

• Deterministic: Same user always gets same variant
• Stateless: No database lookup for every request
• Configurable: Adjust split via environment variables
• Fast: O(1) assignment decision

Code Example:

def pick_arm(partner_id, app_id, v1_weight=0.5):

```
key = f"{partner_id}:{app_id}"
hash_val = int(hashlib.md5(key.encode()).hexdigest(), 16) % 100
return "v1" if hash_val < v1_weight * 100 else "v2"
```

3.2 A/B Test Configuration

DynamoDB Table: ab_experiments

• experiment_id (PK): "embedding_v1_vs_v2_2025_10"

• status: "active" | "completed" | "stopped"
• v1_weight: 0.5 (50% traffic)
• v2_weight: 0.5 (50% traffic)
• start_date: "2025-10-15"
• end_date: "2025-10-30"
• winning_criteria: "ctr_improvement > 5%"

DynamoDB Table: ab_metrics
• experiment_id (PK): "embedding_v1_vs_v2_2025_10"
• timestamp (SK): 1729360000
• arm: "v1" | "v2"
• requests_count: 10000
• avg_latency_ms: 25.3
• p99_latency_ms: 89.1
• ctr: 0.0235
• prediction_error: 0.0123

3.3 Metrics to Track

• Performance Metrics:**
• Latency (p50, p95, p99)
• Throughput (requests/second)
• Error rate (4xx, 5xx)
• Cache hit rate

• Business Metrics:**
• Click-Through Rate (CTR)
• Conversion Rate
• Prediction accuracy (MAE, RMSE)
• User engagement

• Statistical Significance:**
• Use Bayesian A/B testing (Beta distribution)
• Require minimum sample size: 10,000 requests per arm
• Confidence level: 95%
• Minimum detectable effect: 5% improvement

3.4 Winner Selection Logic

Automated Decision (Lambda triggered daily):

1. Check if experiment has run for minimum duration (7 days)
2. Verify minimum sample size reached (10K per arm)
3. Calculate statistical significance (Bayesian credible interval)

4. If winner detected:

```
- ctr_improvement > 5% AND p_value < 0.05
- Gradually shift traffic to winner (10% increments)
- Monitor for regressions
- After 3 days of stability, promote to 100%
```

5. Notify team via SNS (Slack channel)

4. CI/CD PIPELINE OVERVIEW

4.1 Pipeline Stages

Stage 1: Source (GitHub)
• Push to main branch triggers pipeline
• GitHub Actions webhook to AWS CodePipeline
• Or use AWS CodeCommit for native integration

Stage 2: Build (AWS CodeBuild)
• Pull Docker base image from ECR
• Run unit tests (pytest)
• Run integration tests (testcontainers)
• Run performance tests (k6 or locust)
• Build Docker image
• Tag with commit SHA + timestamp
• Push to Amazon ECR

Stage 3: Deploy to Staging (ECS Fargate)
• Update ECS task definition (new image tag)
• Deploy to staging cluster
• Run smoke tests
• Run end-to-end tests
• Performance benchmarking

Stage 4: Manual Approval (Optional)

• Notify team via SNS

• Review test results

• Approve/Reject via AWS Console or CLI

Stage 5: Deploy to Production (Blue/Green)

• Create new ECS task set (Green)

• Route 10% traffic to Green (canary)

• Monitor metrics for 10 minutes

• If healthy: shift remaining 90%

• If unhealthy: automatic rollback to Blue

• Terminate old task set after 1 hour

Stage 6: Post-Deployment Validation

• Run synthetic tests (CloudWatch Synthetics)

• Monitor error rate, latency

• Alert if anomalies detected

4.2 CodePipeline Configuration

buildspec.yml (CodeBuild):

```
version: 0.2 phases: pre_build: commands: - echo Logging in to Amazon ECR... - aws ecr
get-login-password | docker login --username AWS ... - pip install -r requirements.txt build:
commands: - echo Running tests... - pytest tests/ --cov=app --cov-report=xml - echo Building
Docker image... - docker build -t similarity-service:$CODEBUILD_RESOLVED_SOURCE_VERSION . -
docker tag similarity-service:$CODEBUILD_RESOLVED_SOURCE_VERSION \
$ECR_REPO_URI:$CODEBUILD_RESOLVED_SOURCE_VERSION post_build: commands: - echo Pushing Docker
image... - docker push $ECR_REPO_URI:$CODEBUILD_RESOLVED_SOURCE_VERSION - echo Writing image
definitions file... - printf '[{"name":"similarity-api","imageUri":"%s"}]' \
$ECR_REPO_URI:$CODEBUILD_RESOLVED_SOURCE_VERSION > imagedefinitions.json artifacts: files:
imagedefinitions.json
```

4.3 Deployment Strategies

• Blue/Green Deployment** (Production)

• Two environments: Blue (current), Green (new)

• Instant rollback capability

• Zero downtime

• Cost: 2x resources during deployment (~10 minutes)

• Canary Deployment** (High-Risk Changes)

• Route 5% -> 10% -> 25% -> 50% -> 100%

• Automated rollback if metrics degrade

• Slower but safer

• Rolling Update** (Low-Risk Changes)
• Update tasks one-by-one
• No extra resources needed
• Longer deployment time

5. MONITORING & ALERTING

5.1 Metrics to Monitor

• Application Metrics (CloudWatch Custom Metrics):**
• similarity_search_latency (ms) [p50, p95, p99]
• prediction_latency (ms)
• embedding_cache_hit_rate (%)
• ab_test_assignment_latency (ms)
• requests_per_second (count)
• error_rate_4xx (%)
• error_rate_5xx (%)

• Infrastructure Metrics (CloudWatch Default):**
• ECS CPU Utilization (%)
• ECS Memory Utilization (%)
• ALB Target Response Time (seconds)
• ALB Healthy Host Count
• Redis CPU Utilization (%)
• Redis Evictions (count)
• DynamoDB Read/Write Capacity (units)
• API Gateway 4XXError, 5XXError (count)

• Business Metrics (Custom):**
• ctr_by_arm (v1, v2)
• prediction_accuracy_mae
• revenue_per_request ($)
• user_satisfaction_score

5.2 Alerting Strategy

• Critical Alerts (PagerDuty 24/7):**
• Error rate > 5% for 5 minutes

• p99 latency > 500ms for 10 minutes

• All ECS tasks unhealthy

• DynamoDB throttling (>100 events/min)

• Zero traffic for 5 minutes

• Warning Alerts (Slack #monitoring):**

• Error rate > 2% for 10 minutes

• p99 latency > 300ms for 15 minutes

• Cache hit rate < 80%

• Redis memory > 80%

• Cost anomaly detected (>20% spike)

• Info Alerts (Email):**

• Daily metrics summary

• Weekly A/B test report

• Monthly cost report

5.3 CloudWatch Alarms Configuration

Alarm: HighErrorRate

```
Metric: error_rate_5xx
Threshold: > 5%
Period: 5 minutes
Evaluation Periods: 2
Action: SNS -> PagerDuty
Auto-Scaling Action: Scale out by 2 tasks
```

Alarm: HighLatency

```
Metric: similarity_search_latency_p99
Threshold: > 500ms
Period: 10 minutes
Evaluation Periods: 2
Action: SNS -> Slack
Auto-Scaling Action: Scale out by 1 task
```

Alarm: LowCacheHitRate

```
Metric: embedding_cache_hit_rate
Threshold: < 80%
Period: 15 minutes
```

```
Evaluation Periods: 3
Action: SNS -> Email (investigate cache config)
```

5.4 Dashboard Layout (CloudWatch)

Dashboard: Similarity Service Production

Panel 1: Traffic

```
- Requests/second (line chart)
- Error rate (stacked area: 4xx, 5xx)
- Cache hit rate (line chart)
```

Panel 2: Latency

```
- p50, p95, p99 latency (line chart)
- Latency heatmap (hour x percentile)
```

Panel 3: Infrastructure

```
- ECS CPU/Memory utilization (gauge)
- Healthy host count (number)
- Redis connections (line chart)
```

Panel 4: A/B Testing

```
- Traffic split (pie chart: v1 vs v2)
- CTR by arm (bar chart)
- Prediction accuracy by arm (line chart)
```

Panel 5: Business Metrics

```
- Revenue/hour (line chart)
- Cost/1M requests (number)
```

6. SCALING CONSIDERATIONS

6.1 Current State (100 requests/day)

Architecture:

• ECS Fargate: 1 task (0.25 vCPU, 0.5 GB RAM)

• ElastiCache: cache.t3.micro (0.5 GB)

• DynamoDB: On-demand pricing

• API Gateway: Pay-per-request
• S3: Standard storage

Cost Estimate: ~$50/month

Performance:

• Avg latency: ~30ms
• p99 latency: ~100ms
• Throughput: 100 req/day (~0.001 req/sec)

6.2 Medium Scale (10,000 requests/day)

Architecture:

• ECS Fargate: 2-4 tasks (auto-scaling)
• ElastiCache: cache.t3.small (1.5 GB)
• DynamoDB: On-demand (auto-scaling)
• API Gateway: Pay-per-request
• S3: Standard storage

Auto-Scaling Policy:

• Target CPU: 70%
• Target Memory: 80%
• Scale-out cooldown: 60 seconds
• Scale-in cooldown: 300 seconds

Cost Estimate: ~$200/month

Performance:

• Avg latency: ~30ms
• p99 latency: ~150ms
• Throughput: 10K req/day (~0.12 req/sec)

6.3 High Scale (1M requests/day)

Architecture:

• ECS Fargate: 10-50 tasks (auto-scaling + scheduled scaling)
• ElastiCache: cache.r6g.xlarge (26 GB) - Cluster mode

• DynamoDB: Provisioned capacity with auto-scaling

• API Gateway: Regional with CloudFront CDN

• S3: Intelligent-Tiering

Optimizations:

1. CloudFront edge caching for repeated queries
2. Redis cluster mode (3 shards) for horizontal scaling
3. DynamoDB Global Tables (multi-region if needed)
4. Batch processing for non-realtime predictions
5. Connection pooling (max 1000 connections per task)

Auto-Scaling Policy:

• Target CPU: 60%

• Target Custom Metric: requests_per_task < 100/sec

• Scheduled Scaling: Scale up 2 hours before peak traffic

• Minimum tasks: 10 (always warm)

• Maximum tasks: 50 (cost limit)

Cost Estimate: ~$2,000-3,000/month

Performance:

• Avg latency: ~30ms

• p99 latency: ~200ms

• Throughput: 1M req/day (~12 req/sec avg, 50 req/sec peak)

6.4 Extreme Scale (10M+ requests/day)

Architecture Changes:

• Multi-region deployment (us-east-1, eu-west-1)

• Global Accelerator for optimal routing

• Aurora Global Database for cross-region replication

• Lambda@Edge for edge computing

• Kinesis for event streaming (instead of direct DynamoDB writes)

• SQS for asynchronous predictions

Cost Estimate: ~$10,000-15,000/month

Performance:

• Avg latency: ~30ms

• p99 latency: ~200ms

• Throughput: 10M req/day (~120 req/sec avg, 500 req/sec peak)

6.5 Bottleneck Analysis & Solutions

Bottleneck 1: Embedding Cache Misses

Problem: Loading embeddings from S3 takes 100-500ms

Solution:

• Pre-warm cache on deployment

• Increase cache size (cache.r6g.4xlarge - 104 GB)

• Use Redis read replicas for read scaling

Bottleneck 2: Cosine Similarity Computation

Problem: Brute-force similarity search doesn't scale to 100K+ apps

Solution:

• Migrate to FAISS or Annoy for ANN search

• Use SageMaker endpoint for GPU-accelerated search

• Pre-compute top-1000 neighbors for popular apps

Bottleneck 3: DynamoDB Throttling

Problem: Burst traffic exceeds provisioned capacity

Solution:

• Use on-demand pricing (auto-scales instantly)

• Or: Set auto-scaling target to 70% utilization

• Use DynamoDB Streams to offload aggregations

Bottleneck 4: Cold Start Latency

Problem: New ECS tasks take 30-60s to start

Solution:

• Keep minimum 5 tasks always running

• Use Fargate Spot for cost savings (70% discount)

• Implement health check warm-up period

7. COST OPTIMIZATION STRATEGIES

7.1 Compute Optimization

• ECS Fargate Savings:**

• Use Fargate Spot for non-critical tasks (70% savings)

• Right-size tasks (start with 0.25 vCPU, monitor utilization)

• Use Graviton2 (ARM) for 20% cost savings

• Set aggressive scale-in policy (shutdown idle tasks faster)

Estimated Savings: 40-50%

• Lambda for Batch Processing:**

• Use Lambda for infrequent tasks (model evaluation, metrics aggregation)

• 1M free requests/month

• Pay per 100ms execution time

Estimated Savings: $200/month vs. dedicated ECS task

7.2 Storage Optimization

• S3 Lifecycle Policies:**

• Move embeddings to S3 Glacier after 90 days (80% savings)

• Move logs to S3 Glacier Deep Archive after 180 days (90% savings)

• Delete old training data after 1 year

• Use S3 Intelligent-Tiering for unpredictable access patterns

Estimated Savings: 60% on storage costs

• ElastiCache Reserved Instances:**

• Purchase 1-year reserved instances for baseline capacity (35% savings)

• Use on-demand for burst capacity

Estimated Savings: 30-35% on cache costs

7.3 Data Transfer Optimization

• Reduce Cross-AZ Transfer:**

• Use single-AZ deployment for dev/staging

• Use VPC endpoints for S3/DynamoDB (no internet gateway fees)

• Enable CloudFront compression (reduce bandwidth by 50%)

Estimated Savings: $100-500/month at scale

7.4 Monitoring Optimization

• CloudWatch Logs Optimization:**
• Filter logs before ingestion (reduce noise)
• Use log sampling for high-volume logs (1% sample)
• Export to S3 after 7 days (90% savings)
• Use CloudWatch Logs Insights instead of Athena queries

Estimated Savings: 70% on log costs

7.5 Cost Monitoring

• AWS Cost Explorer:**
• Set budgets for each service
• Enable anomaly detection (detect cost spikes)
• Tag resources by environment (prod, staging, dev)

• Cost Allocation Tags:**
• Team: data-science, ml-engineering
• Environment: prod, staging, dev
• Project: similarity-service
• CostCenter: 12345

• Monthly Cost Review:**
• Identify top 10 cost drivers
• Right-size over-provisioned resources
• Delete unused resources (old ECS task definitions, AMIs)

7.6 Total Cost Breakdown (1M requests/day)

ECS Fargate (20 tasks avg): $800/month
ElastiCache (r6g.xlarge): $400/month
DynamoDB (provisioned): $300/month
S3 (1 TB storage): $23/month
API Gateway (1M requests): $3.50/month
CloudWatch (logs + metrics): $200/month
Data Transfer: $100/month

Load Balancer: $20/month

Route53: $1/month

SNS: $1/month

Total: $1,848/month

• With Optimizations:**

• Fargate Spot: $800 -> $400 (-50%)

• ElastiCache RI: $400 -> $260 (-35%)

• S3 Lifecycle: $23 -> $10 (-57%)

• CloudWatch sampling: $200 -> $60 (-70%)

Optimized Total: $1,055/month

• Annual Savings: $9,516 (43% reduction)**

8. FEEDBACK LOOP FOR MODEL IMPROVEMENT

8.1 Data Collection

Real-Time Events (Kinesis Data Streams):

• Prediction event: {app_id, query, neighbors, predicted_score, timestamp}

• Click event: {app_id, campaign_id, clicked, timestamp}

• Conversion event: {app_id, campaign_id, converted, revenue, timestamp}

Batch Exports (Daily):

• Export CloudWatch Logs to S3 (parquet format)

• Export DynamoDB metrics to S3 (time-series data)

8.2 Feature Store (SageMaker Feature Store)

Feature Group: app_features

• app_id (PK)

• category

• region

• pricing_model

• features_list

• historical_ctr (updated daily)

• avg_rating (updated weekly)

• last_updated (timestamp)

8.3 Model Retraining Pipeline

Trigger: Weekly (or on-demand)

Step 1: Data Preparation (Lambda)

- Query last 30 days of prediction + outcome data
- Join with feature store
- Filter outliers
- Save to S3 training bucket

Step 2: Model Training (SageMaker Training Job)

- Load training data from S3
- Train new embedding model (v3)
- Validate on holdout set (20%)
- Compute metrics: recall@10, precision@10, MAP
- Save model artifacts to S3

Step 3: Model Evaluation (Lambda)

- Compare v3 vs v2 on validation set
- If v3 improves recall@10 by >3%:

```
- Promote to staging
- Notify team via SNS
```

- Else:

```
- Archive model
- Log metrics to DynamoDB
```

Step 4: Staging Deployment (Step Functions)

- Deploy v3 to staging environment
- Run regression tests
- Run A/B test simulator (synthetic traffic)
- If all tests pass:

```
- Approve for production
```

- Else:

```
- Rollback
```

Step 5: Production Deployment (Gradual)

- Create new A/B experiment: v2 vs v3
- Route 10% traffic to v3

• Monitor for 48 hours

• If CTR improvement >5%:

```
- Gradually shift to 50/50
- Eventually 100% v3
```

• Deprecate v2 after 30 days

8.4 Monitoring Drift

Feature Drift Detection:

• Monitor distribution of input features (category, region)

• Alert if >20% shift in 7 days

• Use SageMaker Model Monitor

Prediction Drift Detection:

• Compare predicted_ctr vs actual_ctr

• Alert if MAE increases by >50%

• Trigger model retraining

9. SECURITY & COMPLIANCE

9.1 Security Best Practices

• Network Security:**

• VPC with private subnets for ECS tasks

• Security groups: Allow only ALB -> ECS on port 8000

• No public IPs for ECS tasks

• VPC endpoints for AWS services (no internet gateway)

• AWS WAF rules: Rate limiting, IP blocking, SQL injection protection

• Data Security:**

• S3 buckets: Block public access

• S3 server-side encryption (SSE-S3 or KMS)

• DynamoDB encryption at rest (KMS)

• ElastiCache encryption in transit (TLS)

• Secrets Manager for API keys, DB passwords

• Access Control:**

• IAM roles for ECS tasks (least privilege)
• IAM roles for Lambda (separate per function)
• MFA for AWS Console access
• CloudTrail for audit logs

• Compliance:**
• GDPR: PII data anonymization, data retention policies
• HIPAA: If handling health data, use HIPAA-eligible services
• SOC 2: Implement logging, monitoring, access controls

9.2 Disaster Recovery

• Backup Strategy:**
• S3: Versioning enabled + Cross-Region Replication
• DynamoDB: Point-in-time recovery (PITR) + On-demand backups
• ElastiCache: Daily snapshots (7-day retention)

• RTO/RPO:**
• RTO (Recovery Time Objective): < 1 hour
• RPO (Recovery Point Objective): < 5 minutes
• Multi-AZ deployment for high availability

• Failover Plan:**
1. Detect outage (CloudWatch alarms)
2. Notify on-call engineer (PagerDuty)
3. Failover to secondary region (Route53 health check)
4. Investigate root cause
5. Post-mortem document

10. CONCLUSION

This architecture is designed to be:

• Scalable: From 100 to 1M+ requests/day with auto-scaling
• Cost-Effective: $50/month at low scale, $1,000-3,000/month at high scale
• Reliable: 99.9% uptime with multi-AZ deployment
• Observable: Comprehensive monitoring and alerting
• Maintainable: CI/CD pipeline with automated testing
• Secure: VPC isolation, encryption at rest/transit, IAM least privilege

• ML-Ready: Feedback loop for continuous model improvement

The system leverages AWS managed services to minimize operational overhead and maximize developer productivity. The A/B testing framework enables data-driven decision making for model improvements.

Next Steps:

1. Implement proof-of-concept on AWS Free Tier
2. Load test with k6 (simulate 1M requests/day)
3. Optimize based on bottleneck analysis
4. Document runbooks for on-call engineers
5. Train team on AWS services and monitoring dashboards

END OF DOCUMENT