AWS Re-architecture POC: WebSphere to AWS Native Services

Executive Summary

This Proof of Concept outlines the migration strategy from a traditional WebSphere-based architecture to a modern, scalable AWS-native solution. The proposed architecture leverages containerization, serverless computing, and managed services to improve scalability, reduce operational overhead, and enhance developer productivity.

Current Architecture Analysis

Existing System

- **Hosting**: WebSphere (IBM Server)
- **Tech Stack**: Spring MVC, Java 8, FreeMarker, JavaScript/jQuery, CSS
- **Databases**: Oracle, Teradata
- **Services**:
  - `dynamic-content-service`: FreeMarker templates, Java 8, Spring Framework, Gradle
  - `static-content-web`: Spring Boot app with frontend assets

Current Challenges

- Legacy infrastructure maintenance overhead
- Limited scalability and elasticity
- High operational costs
- Monolithic architecture constraints
- Dependency on proprietary IBM middleware

Proposed AWS-Native Architecture

1. Containerization Strategy

Container Platform: Amazon ECS with Fargate

yaml

```
# ECS Task Definition Example
apiVersion: v1
kind: TaskDefinition
metadata:
  name: dynamic-content-service
spec:
  family: dynamic-content-service
  cpu: 512
  memory: 1024
  networkMode: awsvpc
  requiresCompatibilities:
    - FARGATE
  containers:
    - name: app
      image: your-account.dkr.ecr.region.amazonaws.com/dynamic-content-service:latest
      portMappings:
        - containerPort: 8080
          protocol: tcp
```

Alternative: Amazon EKS

- For teams preferring Kubernetes
- Better suited for complex microservices orchestration
- Native integration with AWS services via controllers

2. Application Modernization

Dynamic Content Service Migration

Current: Java 8 + Spring Framework + FreeMarker + Gradle **Target**: Java 17 + Spring Boot 3.x + Thymeleaf/FreeMarker + AWS SDK v2 java

```
// Modern Spring Boot Configuration
@SpringBootApplication
@EnableWebMvc
public class DynamicContentApplication {

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("classpath:/templates/");
        configurer.setDefaultEncoding("UTF-8");
        return configurer;
    }

    @Bean
    public AmazonS3 s3Client() {
        return AmazonS3ClientBuilder.standard()
            .withRegion(Regions.US_EAST_1)
            .build();
    }
}
```

Static Content Web Migration

Current: Spring Boot + JavaScript/jQuery + CSS **Target**:

- **Option A**: Modernized SPA (React/Vue.js) + AWS Amplify
- **Option B**: Enhanced Spring Boot with modern frontend served from S3/CloudFront

3. Database Migration Strategy

Oracle Database → Amazon RDS for PostgreSQL

```sql
-- Migration approach using AWS Database Migration Service (DMS)
-- Schema conversion using AWS Schema Conversion Tool (SCT)

-- Example PostgreSQL equivalent
CREATE TABLE content_templates (
    id SERIAL PRIMARY KEY,
    template_name VARCHAR(255) NOT NULL,
    template_content TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Teradata → Amazon Redshift

```sql
-- Data warehouse migration
-- Redshift-optimized schema design
CREATE TABLE analytics_data (
    event_id BIGINT IDENTITY(1,1),
    user_id INTEGER,
    event_type VARCHAR(50),
    event_timestamp TIMESTAMP,
    event_data JSON
)
DISTSTYLE KEY
DISTKEY (user_id)
SORTKEY (event_timestamp);
```

4. AWS Services Mapping

Current Component	AWS Native Service	Benefits
WebSphere Server	ECS Fargate / EKS	Serverless containers, auto-scaling
Load Balancing	Application Load Balancer	Advanced routing, health checks
Static Assets	S3 + CloudFront	Global CDN, cost-effective storage
Oracle Database	RDS PostgreSQL	Managed service, automated backups
Teradata	Amazon Redshift	Serverless analytics, cost optimization
File Storage	Amazon EFS/S3	Scalable, durable storage
Monitoring	CloudWatch + X-Ray	Comprehensive observability
CI/CD	CodePipeline + CodeBuild	Native AWS DevOps tools

Implementation Phases

Phase 1: Foundation Setup (Weeks 1-2)

- □ AWS Account setup and IAM configuration
- □ VPC and networking infrastructure
- □ ECR repository creation
- □ RDS PostgreSQL instance provisioning

Phase 2: Application Containerization (Weeks 3-4)

- □ Dockerfile creation for both services
- □ Local Docker testing
- □ ECR image push and version management
- □ ECS cluster and task definition setup

Phase 3: Database Migration (Weeks 5-6)

- □ Schema conversion using AWS SCT
- □ DMS replication instance setup
- □ Data migration and validation
- □ Application connection string updates

Phase 4: Service Deployment (Weeks 7-8)

- □ ECS service deployment
- □ Application Load Balancer configuration
- □ Auto Scaling group setup
- □ Health check implementation

Phase 5: Static Content Migration (Week 9)

- □ S3 bucket creation and configuration
- □ CloudFront distribution setup
- □ Asset upload and testing
- □ DNS configuration

Phase 6: Testing and Optimization (Weeks 10-11)

- □ Performance testing
- □ Security assessment
- □ Cost optimization review
- □ Documentation completion

Sample Infrastructure as Code

Terraform Configuration

hcl

```
# ECS Cluster
resource "aws_ecs_cluster" "main" {
  name = "content-management-cluster"

  setting {
    name  = "containerInsights"
    value = "enabled"
  }
}

# Application Load Balancer
resource "aws_lb" "main" {
  name               = "content-management-alb"
  internal           = false
  load_balancer_type = "application"
  security_groups    = [aws_security_group.alb.id]
  subnets            = aws_subnet.public[*].id

  enable_deletion_protection = false
}

# RDS Instance
resource "aws_db_instance" "main" {
  identifier        = "content-management-db"
  engine            = "postgres"
  engine_version    = "14.9"
  instance_class    = "db.t3.medium"
  allocated_storage = 20
  storage_encrypted = true

  db_name  = "contentdb"
  username = "dbadmin"
  password = var.db_password

  vpc_security_group_ids = [aws_security_group.rds.id]
  db_subnet_group_name   = aws_db_subnet_group.main.name

  backup_retention_period = 7
  backup_window           = "03:00-04:00"
  maintenance_window      = "sun:04:00-sun:05:00"
```

```
    skip_final_snapshot = true
  }
```

Cost Analysis

Current vs. Proposed Monthly Costs (Estimated)

Component	Current (WebSphere)	Proposed (AWS)	Savings
Compute	$2,500	$800 (ECS Fargate)	$1,700
Database	$1,800	$400 (RDS t3.medium)	$1,400
Storage	$300	$150 (S3 + EFS)	$150
Networking	$200	$100 (ALB + CloudFront)	$100
Total	**$4,800**	**$1,450**	**$3,350**

Note: Costs may vary based on actual usage patterns and reserved instance purchases

Risk Mitigation

Technical Risks

1. **Database Compatibility**: Use AWS SCT for schema analysis before migration
2. **Application Dependencies**: Thorough dependency audit and testing
3. **Performance Degradation**: Load testing in staging environment

Business Risks

1. **Downtime**: Blue-green deployment strategy
2. **Data Loss**: Comprehensive backup and rollback procedures
3. **Skills Gap**: Team training and AWS certification programs

Success Metrics

Performance Metrics

- Application response time: < 200ms (current: 500ms)
- Database query performance: 50% improvement
- System availability: 99.9% uptime

Operational Metrics

- Deployment frequency: From monthly to weekly

- Mean time to recovery: < 30 minutes
- Infrastructure provisioning: From days to hours

Cost Metrics

- 70% reduction in infrastructure costs
- 50% reduction in operational overhead
- ROI achievement within 12 months

Next Steps

1. **Stakeholder Approval**: Present POC to leadership team
2. **Team Training**: AWS fundamentals and containerization
3. **Environment Setup**: Development and staging environments
4. **Pilot Migration**: Start with dynamic-content-service
5. **Full Migration**: Complete system migration following phases

Conclusion

This AWS-native re-architecture provides a modern, scalable, and cost-effective solution that addresses current limitations while positioning the organization for future growth. The containerized approach with managed services reduces operational complexity while improving system reliability and developer productivity.

The phased approach ensures minimal business disruption while providing early wins and learning opportunities throughout the migration process.