Scalable Web Application Architecture — Traditional vs. Serverless

Project Deliverable  
IslandCart (E-commerce Platform)

# Project Overview

This project demonstrates two approaches to building a scalable web application (IslandCart, an e-commerce platform):  
  
1. Traditional Server Scaling  
 - Uses load balancers, EC2 Auto Scaling Groups, RDS, ElastiCache, S3, and SQS + workers.  
 - Vertical scaling at the database layer.  
 - Horizontal scaling at the web/application tiers.  
  
2. Serverless Architecture  
 - Uses CloudFront, S3, API Gateway, Lambda, DynamoDB, Cognito, and EventBridge.  
 - Fully managed services with automatic scaling.  
 - Pay-per-request cost model.  
  
The goal is to compare the two designs in terms of performance, cost, scaling, operational overhead, and use cases.

# Architecture Diagrams

Traditional Design: Route 53 → WAF → CloudFront → ALB → EC2 Web ASG → EC2 App ASG → {RDS, ElastiCache, S3, SQS → Worker ASG}

Serverless Design: Route 53 → WAF (on CloudFront) → CloudFront (origins: S3 static, API Gateway) → API Gateway (JWT authorizer via Cognito) → Lambda → {DynamoDB, S3 (pre-signed), EventBridge → Lambda/StepFn}

# Scaling Strategies

## Traditional

- Vertical Scaling (DB): Instance upgrades (r7g.large → r7g.8xlarge), read replicas, RDS Proxy.  
- Horizontal Scaling (Web/App/Workers): Auto Scaling Groups tied to ALB metrics; Redis for sessions; Workers scale on SQS backlog.

## Serverless

- Lambda Scaling: Automatic per-request; use Provisioned Concurrency for critical paths.  
- DynamoDB Scaling: On-Demand or Autoscaling for RCU/WCU.  
- API Gateway & CloudFront: Scale automatically with traffic.  
- Event-driven: EventBridge + Lambda/Step Functions handle async workloads.

# Comparison & Analysis

## Performance

Traditional: Predictable latency, ASG lag during spikes.  
Serverless: Near-instant scaling, occasional cold starts.

## Cost

Traditional: Fixed baseline (EC2, RDS, Redis, NAT). Good for steady load with Savings Plans.  
Serverless: Pay-per-use. Excellent for spiky/variable workloads.

## Operations

Traditional: Requires patching, scaling policies, DB tuning.  
Serverless: No servers to manage; focus on IAM, limits, and monitoring.

## Development & Deployment

Traditional: Full-stack servers, AMI/container deploys, blue-green/rolling.  
Serverless: Function-per-feature, CDK/SAM pipelines, smaller blast radius.

# Use Cases

Traditional best for:  
- Constant, predictable traffic.  
- Complex SQL workloads.  
- Environments with strong DevOps/SRE teams.  
- Compliance requiring server-level controls.  
  
Serverless best for:  
- Spiky or unpredictable traffic.  
- Event/HTTP-driven apps.  
- Lean teams that want to minimize ops.  
- Rapid iteration and global reach.

# Deliverables

- Architecture Diagrams: Traditional & Serverless.  
- Design Document: Detailed explanation of components, scaling strategies, decisions, and security considerations.  
- Comparison (Part 3): Scalability, cost, operational complexity, workflows, and use cases.  
- README.md: Summary for quick reference.

# Conclusion

Traditional scaling provides control, predictability, and suits steady traffic and SQL-heavy applications.  
Serverless provides agility, low idle cost, and elastic scaling for bursty or fast-evolving workloads.  
The right choice depends on traffic patterns, team expertise, compliance needs, and long-term scalability goals.