# Performance Testing Strategy Document

**Executive Summary**

This document outlines a comprehensive performance testing strategy for a cloud-based e-commerce platform that recently migrated from a monolithic architecture to AWS microservices. The strategy is tailored to ensure scalability, reliability, and cost-efficiency under extreme traffic conditions, especially during flash sales and seasonal campaigns.

**Part 1: Design** – Strategy Definition

Testing Objectives

We aim to validate and enhance system reliability under varying loads by executing three major types of performance tests:

|  |  |
| --- | --- |
| **Test Type** | **Objective** |
| Load Testing | Validate system behavior under expected and peak traffic |
| Stress Testing | Identify breaking points, error recovery behavior, and bottleneck layers |
| Soak (Endurance) Testing | Detect memory leaks, connection exhaustion, and long-term stability |

**Metrics for Monitoring**

|  |  |
| --- | --- |
| **System Layer** | **Monitored Metrics** |
| Application Load Balancer (ALB) | Request count, Target response time, Error rate (5xx/4xx), Latency |
| ECS Services (Node.js) | CPU/Memory utilization, Container restarts, Network throughput |
| RDS PostgreSQL | Query execution time, Active connections, IOPS, Deadlocks |
| Global System View | Throughput (TPS), Response time percentiles, Concurrent sessions |

**Tools and Platforms**

|  |  |
| --- | --- |
| **Category** | **Tools** |
| Load Generation | Apache JMeter (distributed mode), AWS Distributed Load Testing (Fargate) |
| Monitoring & Tracing | AWS CloudWatch, AWS X-Ray, Grafana + Prometheus (optional integration) |
| Logging & Analysis | ALB Access Logs, RDS Performance Insights, JMeter listeners |
| CI/CD Integration | Jenkins, GitHub Actions, Slack/email notifications, Allure Reports |

**Realistic Test Scenarios**

1. High-Traffic Browsing: Concurrent users login, browse categories, filter products, view item details.

2. Checkout Flow: Users add items to cart, apply promo codes, and complete payment concurrently.

3. Flash Sale Spikes: Massive short-burst traffic searching for popular items and racing to purchase.

**Test Environment Provisioning**

- Infrastructure Duplication: Clone production architecture using Terraform or CloudFormation into isolated staging.

- Data Simulation: Seed database with production-like users, items, carts, and purchase records.

- Load Simulation Strategy:

- JMeter distributed agents on EC2 Spot Instances.

- Load orchestrated from multiple AWS Regions.

- Auto-scaling and rollback policies defined in testing pipeline.

**Part 2: Execution** – Test Implementation & Observability

**Sample Load Flow** (Pseudocode for Checkout Scenario)

Thread Group: +500 Virtual Users, Ramp-up: 5 minutes

* - POST /auth/login → Assert status 200/201, token exists
* -GET /products?category=xyz → Assert latency < 2s
* - POST /cart/add → Assert product count increased
* - POST /checkout → Assert 200/201 with order confirmation

**JMeter Execution Notes:**

* Add **HTTP Header Manager**: Content-Type: application/json
* Use **JSON Extractor** to capture the token from the login response
* Inject token into the Authorization header of all subsequent requests
* Add **Duration Assertion** to validate latency < 2s on /products
* Validate cart item addition using JSON path: $.cart.total\_items > 0

**Assertions on:**

* Response time thresholds (p90, p95)
* Functional correctness under load:
  + Validate HTTP status codes (e.g., 200 OK, 201 Created)
  + Check that required fields exist in the response JSON (e.g., "order\_id", "status")
  + Ensure response values are correct (e.g., "status" == "confirmed")
  + Validate that lists returned (e.g., items[]) are not empty
  + Detect unexpected nulls or schema violations
* Error handling:
  + Ensure retries occur on transient errors (e.g., 5xx, timeouts)
  + Validate proper response on invalid input (e.g., 400 Bad Request)
  + Track failover behavior under backend or DB failure conditions

**Monitoring Strategy**

|  |  |
| --- | --- |
| **Monitoring Layer** | **Metric Tools & Dashboards** |
| Application | CloudWatch Dashboards (ALB metrics), JMeter Backend Listeners |
| Backend Services | ECS Task Metrics, X-Ray Service Map, Prometheus Alerts |
| Database | RDS Insights, Query Profiling Logs, Connection Pool Monitoring |
| Real-time Alerts | CloudWatch Alarms, Slack Notifications, Grafana Threshold Warnings |

**Reporting Methodology**

- **Visuals:**

* TPS graphs, error rate trends, CPU/memory utilization
* Heatmaps for latency bottlenecks

**- Executive Summary**:

* Met vs. Unmet SLAs
* Top 3 latency contributors
* High-impact optimization opportunities

**- Artifacts**:

* Allure or HTML reports, Grafana/PrometheusGrafana dashboards , Logs, Test definitions (.jmx)

**Part 3: Optimization** – Recommendations and Cost Awareness

**Performance Recommendations** (Based on Hypothetical Results)

|  |  |
| --- | --- |
| **Issue Identified** | **Recommended Optimization** |
| ECS CPU spikes under checkout | Split microservices, increase CPU units, optimize Node.js thread pool |
| RDS query latency in flash sale | Add composite indexes, cache common queries, enable read replicas |
| ALB 5xx rate increase | Refine ECS health checks, adjust timeouts, improve error handling logic |

**Cost-Efficient Execution Strategy**

* Deploy EC2 agents as Spot Instances with fallback to On-Demand
* Schedule load tests during off-peak hours with automated shutdown
* Use AWS budgets and cost alerts tied to test duration
* Minimal test duration for stress/soak through parallelism

**Additional Bottlenecks & Remedies (Advanced RDS Troubleshooting)**

|  |  |  |
| --- | --- | --- |
| **Issue** | **How to Identify** | **Recommended Solution** |
| Replica Lag | High ReplicaLag in CloudWatch | Lag-aware routing, monitor replication health |
| Too Many Connections | Repeated connection errors | Use PgBouncer / HikariCP, limit pool size |
| Deadlocks on Tables | Deadlock alerts in RDS Insights | Avoid nested locks, use optimistic concurrency |

**Advanced Optimization for Serverless & Distributed Architectures**

|  |  |  |
| --- | --- | --- |
| Challenge | How to Identify | Recommended Solution |
| Cold Start in Lambda | High initial latency for first call | Use Provisioned Concurrency to keep functions warm |
| Rate Limiting/API Overload | Throttling errors or 429 responses | Apply throttling policies in API Gateway or NGINX ingress controller |
| Scaling Underutilized | Slow response under sudden load | Enable HPA (Kubernetes), Auto Scaling Groups (EC2/ECS), CloudWatch alarms |
| Caching Inefficiencies | High load on API/DB for static or repeated data | Apply Redis, TTL, CloudFront, AppSync caching |
| Query Caching in DB | Same queries executed repeatedly | Use PostgreSQL/MySQL query cache, or Redis layer |
| Partitioning/Archiving in DB | Growing DB with historical data | Use table partitioning and automated archiving strategies |
| Resource Overcommit in K8s | Pods evicted or starved for CPU/Memory | Tune Kubernetes Requests & Limits per container |

**Conclusion**

The above approach ensures your e-commerce system can scale predictably, recover gracefully, and operate efficiently under high-pressure campaigns. It combines modern observability tools with a modular test strategy to deliver actionable insights.

Prepared by Saber Wishahi | Performance Lead