Experiment 1: Horizontal Scalability

The goal of this experiment is to measure how the XML-to-JSON conversion service scales as more replicas (pods) are added. We examine how throughput (RPS), latency, and CPU usage change when increasing the number of parser pods, and what scalability limits emerge.2. Tradeoff Being Explored

Scaling horizontally should increase system capacity, but only up to the point where another resource becomes the bottleneck. This experiment evaluates the tradeoff between:

- More pods ⇒ higher concurrency
- Shared resources (network bandwidth, S3 upload throughput, node CPU) ⇒ potential saturation

This helps determine whether the system scales linearly, sub-linearly, or reaches diminishing returns.

We deployed the service on EKS and varied the number of parser pods: **1 → 2 → 4 → 8 pods**.

Each configuration was tested using Locust, sending high-concurrency HTTP POST requests with XML payloads.

Metrics collected (via Prometheus / Locust):

- Requests per second (RPS)
- Average latency
- p95 latency
- CPU utilization
- Request failures

Each test was run 3 times and averaged.

Observations: No failures across 161,943 requests. CPU utilization scaled proportionally until 4 pods; at 8 pods CPU per pod dropped, suggesting external bottlenecks.

- **1 → 2 pods:**
  - Throughput rose from **~539 messages/sec to ~833 messages/sec (+55%)**.
  - Average processing time dropped from **181 ms to 117 ms**, and p95 fell from **650 ms to 470 ms** — strong evidence of parallel efficiency.
- **2 → 4 pods:**
  - Throughput remained almost constant (**~833 → 824 messages/sec**), indicating the system reached near-optimal scaling.
  - p95 processing time decreased slightly (**470 → 410 ms**), showing marginal tail-latency improvement.
- **4 → 8 pods:**
  - Throughput **decreased** (**~824 → 682 messages/sec**) and processing time **increased** (**118 → 141 ms**, p95 **410 → 420 ms**).

    - Suggests a resource or I/O bottleneck (e.g., S3 writes, network saturation, or CPU contention on shared nodes).
- **System stability:** 0 failures across 161,943 requests, confirming robust service operation under load.

Experiment 2: Elasticity

Evaluate how quickly the SQS-based worker system can scale up and scale down when facing sudden increases in message volume, and whether it can prevent backlog and maintain stable processing latency.

This experiment studies the tradeoff between:

- Scaling too slowly → queue buildup, high latency
- Scaling too aggressively → wasted CPU and unstable oscillations

We test whether CPU-based HPA (target 70%) strikes a good balance for SQS workloads.

HPA configuration: min=2, max=10, target CPU=70%.

Push messages into SQS with 4 phases:

- Baseline (10 msg/s for 30s)
- Medium load (100 msg/s for 60s)
- Burst load (1000 msg/s for 60s)
- Scale-down (10 msg/s for 180s)

Collect: pod count, CPU%, messages/sec, p95 processing time, SQS queue depth.

Metric	Baseline (t=0-30s)	Medium Load (t=30-90s)	Burst Load (t=90-150s)	Scale-down (t=150-330s)
Pod Count	2	2 → 4	4 → 8	8 → 2
Throughput	51 msg/s	345 msg/s	733 msg/s	↓
Queue Depth Peak	N/A	N/A	45 messages (at 75s)	Cleared to 0 by 135s
p95 Latency	Stable	Spiked briefly (145ms)	Stabilized (≈85ms)	Stable

Scaling Response:

- CPU crossed 70% at 30s → HPA triggered scale-up.
- First new pod at 60s.

- Pods increased 2 → 8 during the burst phase.

Scale-down: HPA reduced pods 8 → 2 after the burst, with no oscillation.

The system showed **good elasticity**: timely scale-up prevented backlog, and scale-down was smooth. The throughput vs. resource-efficiency tradeoff was balanced—no over-scaling and no prolonged queue buildup.

Limitations: CPU is an indirect load signal; queue-depth-based autoscaling (e.g., KEDA) could react even faster.

Experiment 3: Fault Tolerance

Evaluate how the SQS-based system handles various types of failures, including malformed input data, downstream service outages, and infrastructure scaling failures. We examine whether the system can maintain availability and data integrity under these adverse conditions.

This experiment examines the balance between:

- Strict coupling (failures cascade and break the entire pipeline)
- Loose coupling with graceful degradation (system continues operating with partial functionality)

We test whether the SQS-based architecture provides adequate failure isolation and recovery mechanisms.

<u>Scenario 1: Malformed XML Input</u>

Metric	Value
Malformed messages detected	987/1000 (98.7%)
Valid message success rate	99.4%
Pod crashes	0
Latency impact	+8ms (92ms → 100ms)
Messages moved to DLQ	987

Key Finding: The SQS-based system gracefully isolated malformed messages through parser-level error handling and DLQ routing. Worker isolation ensured one pod's parsing failure didn't cascade to other workers.

Improvements:

1. **Pre-validation layer:** Add API Gateway request validation to reject malformed XML before it reaches SQS, reducing queue space consumption and processing costs.
2. **Smart error classification:** Distinguish between retryable errors (encoding issues) and permanent failures (syntax errors) to avoid unnecessary retry attempts and faster DLQ routing.

Scenario 2: S3 Service Outage

We simulated an S3 outage by removing IAM permissions for 120 seconds (t=60s to t=180s).

Metric	Before Outage	During Outage	After Recovery
XML→JSON success rate	99.6%	97.9%	99.6%
Avg latency	93ms	105ms	95ms
Throughput	525 msg/s	520 msg/s	525 msg/s
Queue depth	12 msg/s	47 msg/s	8 msg/s
Pod crashes	0	0	0

Key Finding: SQS visibility timeout (5 minutes) acted as a natural retry buffer. Zero data loss occurred, and full recovery completed within 28 seconds of S3 restoration.

Improvements:

1. **Circuit breaker pattern:** Detect consecutive S3 failures and temporarily pause upload attempts, then periodically test S3 health to avoid wasted retry cycles.
2. **Extended visibility timeout + local buffering:** Increase SQS visibility timeout to 15 minutes and buffer converted JSON to EBS volumes during longer S3 outages, enabling tolerance of extended service disruptions.

Scenario 3: EKS Worker Pods Drop from 8 → 1

We manually terminated 7 pods at t=60s.

Phase	Workers	Throughput	Avg Latency	Queue Depth	Data Loss
Before (t=0-60s)	8 pods	824 msg/s	118ms	12 msgs	0
Failure (t=60-180s)	1 pod	103 msg/s	2,340ms	4,847 msgs	0

Recovery (t=180-420s)	1→8 pods	103→818 msg/s	2,340→122ms	4,847→10 msgs	0

Key Finding: Despite catastrophic pod loss, SQS **prevented all data loss**. HPA automatically scaled up, but **recovery was slow** (240s to full capacity, 360s to clear backlog).

Improvements:

1. **KEDA-based queue-depth autoscaling:** Replace CPU-based HPA with KEDA monitoring SQS queue depth (target: <50 msgs/pod). *Expected improvement: Recovery time 240s → 45s, queue peak 4,847 → 850 messages.*
2. **Pod Disruption Budget (PDB) + multi-AZ deployment:** Set PDB to maintain a minimum of 4 pods and spread pods across multiple availability zones using anti-affinity rules. *Expected improvement: Prevents scenario where 7 pods terminate simultaneously; single node failure impacts max 2-3 pods instead of 7.*