aws

AWS Journey

Performance Testing on AWS

Performance Testing on AWS: AWS Journey

Table of Contents

Overview **1**
Journey Overview 1
How This Journey Works 1
Journey Stages 2
Why Performance Fails in Production **4**
The "Early, After, Always" Philosophy 4
The Discipline Pyramid 5
When and Why to Performance Test **7**
The Seven Use Cases 7
What Each Use Case Looks Like in Practice 8
Mapping Use Cases to This Journey 9
Why Every Stage Matters 9
Defining Success: SLIs, SLOs, and Performance Budgets **11**
SLI, SLO, SLA: The Measurement Stack 11
Understanding Percentiles (p50, p90, p95, p99) 11
Choosing the Right SLIs for Load Testing 12
Error Budgets and Performance Budgets 13
Understanding Test Types **14**
The Eight Performance Test Types 14
Smoke Tests: Fail Fast Before You Invest 15
Baseline Tests: Know Your Sweet Spot 15
Load Tests: Validate Your Peak Capacity 16
Stress Tests: Find Your Breaking Point 16
Endurance (Soak) Tests: Catch Time-Dependent Degradation 17
Scalability Tests: Verify Linear Scaling 17
Spike Tests: Validate Elasticity Under Sudden Bursts 18
Volume Tests: Measure Data-Scale Impact 18
Choosing the Right Test for Your Situation 19
Common Mistakes in Test Type Selection 19
Combining Test Types into a Testing Calendar 20
Choosing Your Testing Tool **21**
Running Scripts at Scale with DLT 22
Architecture Overview 24
Deployment Options 24

Deployment Checklist 25
Multi-Region Testing 26
DLT CLI for CI/CD Integration 26
Put It Into Practice 26
Troubleshooting Common Deployment Issues 26
Designing Realistic Test Scenarios 28
Principles of Realistic Workload Design 28
Test Data Strategy 28
Environment Considerations 29
Common Pitfalls 29
Script Structure Best Practices 30
Determining the Right Concurrency Target 31
Put It Into Practice 31
Interpreting Results and Taking Action 32
The Results Hierarchy 32
Observability During Load Tests 33
Service-Specific Insights 34
Building a Performance Analysis Workflow 36
CloudWatch Logs Insights Queries for Post-Test Analysis 37
Put It Into Practice 37
Automated Root Cause Analysis with AWS DevOps Agent 37
Automating Performance in CI/CD 40
Where Performance Tests Fit in the Pipeline 40
Integration Approach: Tool Agnostic 40
Pipeline Integration Pattern 41
Concrete Example: GitHub Actions with DLT CLI 41
Baseline Management 42
Scheduling Recurring Tests 43
Handling Performance Test Failures in the Pipeline 43
Dealing with Flaky Performance Tests 44
Put It Into Practice 44
Chaos Engineering Meets Load Testing 46
The Combined Pattern: DLT + AWS Fault Injection Service 46
Common DLT + FIS Experiment Combinations 47
Designing Your First Chaos + Load Experiment 47
Starter FIS Experiment: AZ Failure Under Load 48

Interpreting Chaos Results 49
Put It Into Practice 49
Scaling to Production-Grade Testing 51
Multi-Region Load Distribution 51
Cost Management for Load Testing 51
AWS Countdown Premium for Critical Events 52
Governance and Organizational Patterns 53
Pre-Event Testing Playbook 53
Service Quota Planning for Large Tests 54
Put It Into Practice 55
Investigation at Scale with AWS DevOps Agent 55
What Comes Next 58
Additional Resources 59
Related AWS Services 59
Other Related Assets 60
Feedback 62
Share your feedback 62

Overview

Journey Overview

Who should read this journey: Platform Engineers, DevOps Engineers, Cloud Architects, SRE, Application Developers, Technical Leaders

This journey takes you from understanding why performance failures happen in production through running continuous, production-scale load tests integrated into your CI/CD pipeline. Your application works in development, but will it survive 10,000 concurrent users on launch day? Performance testing is the discipline that answers this question before your customers do, and this journey embeds that discipline into every stage of your development lifecycle.

Performance failures share a common root cause: the application was never tested under realistic conditions at realistic scale. Teams assume that passing unit tests and integration tests means the system is production-ready, but performance is an emergent property that only manifests when components interact under load. The cost of discovering these problems in production includes lost revenue, emergency war rooms, and weeks of post-incident remediation.

By the end of this journey, you will have deployed Distributed Load Testing on AWS to simulate realistic traffic patterns, defined success metrics through SLIs and SLOs, established baseline comparisons that catch regressions automatically, and combined load testing with chaos engineering to build truly resilient systems. Performance testing will be a continuous signal in your pipeline rather than a one-time gate before launch.

How This Journey Works

This journey follows a maturity progression from Explorer through Practitioner to Expert. Early stages build foundational understanding of why performance fails in production and establish the "Early, After, Always" testing philosophy. Middle stages provide hands-on implementation guidance for deploying load testing infrastructure, defining success metrics through SLIs and SLOs, and integrating tests into CI/CD pipelines. Later stages address chaos engineering integration, production-scale validation, and continuous performance governance. Each stage builds on what came before, transforming performance testing from a pre-launch gate into a continuous engineering discipline.

Journey Stages

Stage	Title	Maturity Level	What You'll Accomplish
1	Why Performance Fails in Production	Explorer	Every engineering team has experienced it. The application sailed through functional testing,...
2	Defining Success: SLIs, SLOs, and Performance Budgets	Explorer	Before running a single test, you need to define what "good" looks like. Without clear success...
3	Understanding Test Types	Practitioner	The term "load test" is often used as a catch-all, but there are eight distinct test types, each...
4	Choosing Your Testing Tool	Practitioner	Distributed Load Testing on AWS (DLT) is the recommended solution for performance testing on...
5	Designing Realistic Test Scenarios	Practitioner	A load test is only as useful as the traffic pattern it simulates. Testing with uniform request...
6	Interpreting Results and Taking Action	Practitioner	Raw load test data is noise without

Stage	Title	Maturity Level	What You'll Accomplish
			a framework for interpretation. The goal is not to generate...
7	Automating Performance in CI/CD	Expert	Performance testing delivers the most value when it runs automatically on every change, not as a...
8	Chaos Engineering Meets Load Testing	Expert	Load testing tells you how the system performs under expected conditions. Chaos engineering...
9	Scaling to Productio n-Grade Testing	Expert	Moving from "my first load test" to "production-grade performance engineering" requires...

Why Performance Fails in Production

Every engineering team has experienced it. The application sailed through functional testing, passed code review, deployed cleanly, and then buckled under real user traffic. Response times crept from 200ms to 2 seconds. Databases hit connection pool limits. Auto Scaling kicked in too late. Customers left.

Performance failures in production share a common root cause: the application was never tested under realistic conditions at realistic scale. Teams assume that passing unit tests and integration tests means the system is production ready. It does not. Performance is an emergent property that only manifests when components interact under load, and the only way to validate it is to simulate that load deliberately.

The cost of discovering performance problems in production is staggering. Industry research consistently shows that each additional second of page load time can reduce conversions by 5-10%. For high-traffic applications, degraded performance during peak events translates directly to lost revenue. Beyond revenue, recovery requires emergency scaling, war rooms, and post-incident remediation that consumes engineering cycles for weeks.

This journey exists because performance testing should not be an afterthought bolted onto the release process. It should be a discipline woven into every stage of development, applied early, applied after every change, and applied always.

The "Early, After, Always" Philosophy

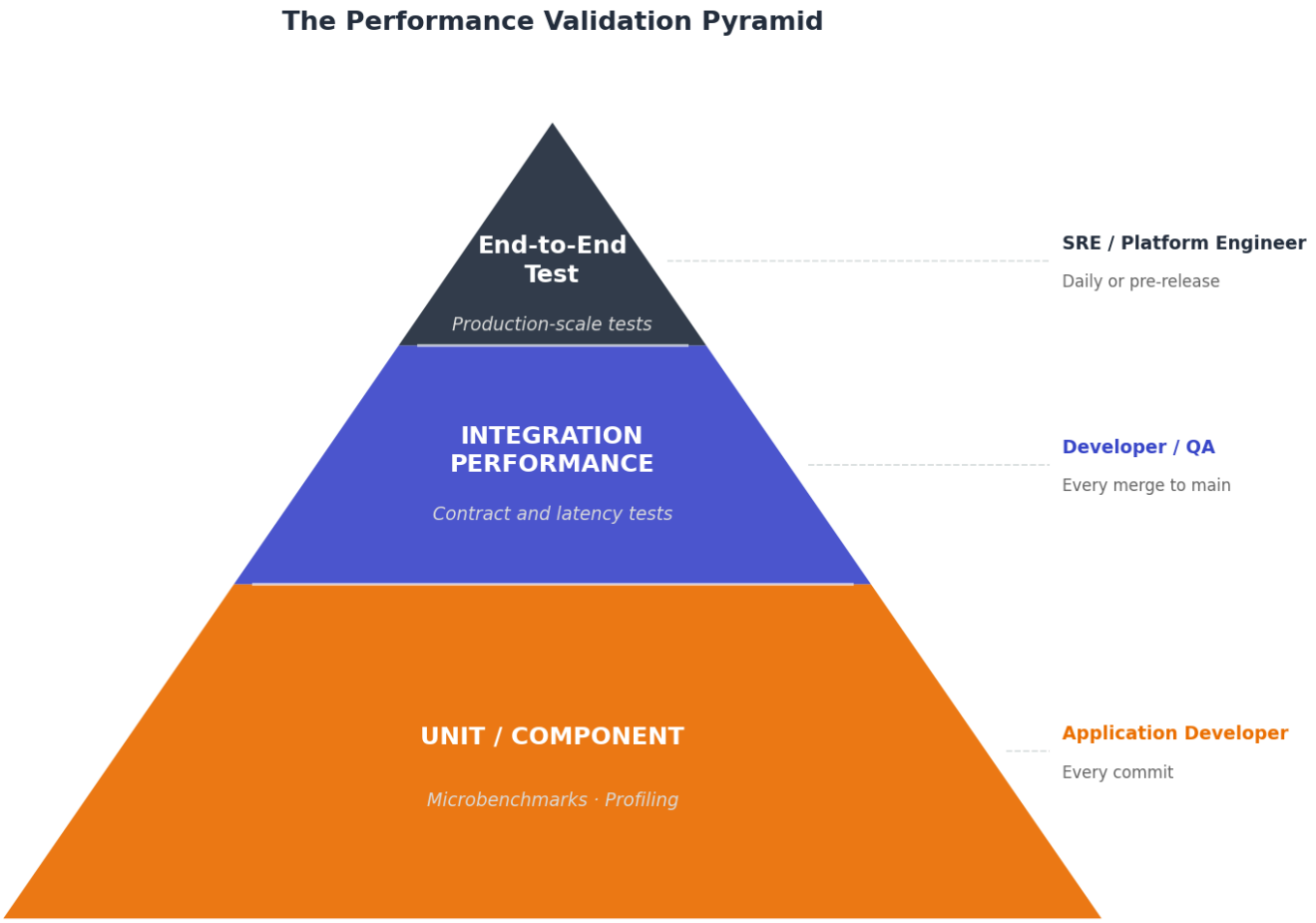
Performance testing as a discipline follows three principles:

Principle	What It Means	When to Apply
Test Early	Establish baselines before you build complexity	Sprint 1, after the first deployable service exists
Test After Every Change	Catch regressions immediate ly	Every merge to main, every infrastructure change
Test Always	Continuous validation against SLOs	Scheduled recurring tests, pre-event validation

These principles transform performance testing from a gate before launch into a continuous signal that your system meets its promises.

The Discipline Pyramid

Performance validation is not a single activity. It is a pyramid of complementary techniques, each owned by different roles:



Level	Owner	Purpose	Frequency
Unit / Component Benchmarks	Application Developer	Detect algorithmic regressions in isolation	Every commit

Level	Owner	Purpose	Frequency
Integration Performance	Developer / QA Engineer	Validate latency contracts between services	Every merge to main
E2E System Load Tests	Platform Engineer / SRE	Validate the full system under production-scale load	Daily or pre-release

This journey focuses primarily on the top of the pyramid: system-level load testing at scale. But effective performance engineering requires all three levels working together.

Go Deeper

A phased approach for performance engineering in the AWS Cloud (AWS Prescriptive Guidance)
PERF05-BP04: Load test your workload (Well-Architected Framework)

When and Why to Performance Test

Every team that invests in performance testing arrives with a specific question. Some are preparing for a launch that cannot afford to fail. Others have inherited a system that occasionally degrades under load and need to understand why. Some are migrating workloads to AWS and need proof that the new architecture meets or exceeds what they had on premises. The question you bring determines where you focus first, but the discipline you build serves all of them.

Performance testing is not a single activity. It is a practice that spans seven distinct use cases, each with its own trigger, stakeholders, and definition of success.

The Seven Use Cases

Use Case	Who Typically Drives It	The Core Question
Launch and High-Traffic Event Prep	Platform Eng, SRE, Product	Can the system handle projected peak load on day one?
Reliability Validation	SRE, Platform Eng, Architects	How does the system behave under different load condition s?
SRE and Continuous Reliabili ty	SRE, DevOps	Are we meeting our SLOs? Will we catch degradation before customers do?
SDLC and Developer Testing	App Developers, DevOps	Did my code change introduce a performance regression?
Migration from On-Premises to AWS	Cloud Architects, Platform Eng	Does the new architecture match or exceed on-prem performance baselines?
Capacity and Sizing Validation	Cloud Architects, FinOps	Are we right-sized? What is our headroom before we hit limits?

Use Case	Who Typically Drives It	The Core Question
Cost Optimization	FinOps, Platform Eng	Can we reduce instance sizes or shift to Graviton without degrading performance?

What Each Use Case Looks Like in Practice

Launch and High-Traffic Event Prep. Your product is launching in six weeks, or Prime Day is approaching, or you are running a marketing campaign expected to triple normal traffic. You need to prove, with data, that the system handles the projected peak. Success means running a full-scale load test that simulates the expected concurrency, validating that latency stays within SLOs, and identifying the breaking point so you know your margin. This use case is time-bounded: you test, you validate, you launch.

Reliability Validation. You need to understand how the system behaves as load increases, decreases, and fluctuates. This goes beyond "can it handle peak" to "what happens at 50% capacity, at 80%, at 110%?" You are mapping the system's performance envelope: where it performs optimally, where it degrades gracefully, and where it fails. This is the foundation that informs architecture decisions, Auto Scaling configurations, and operational runbooks.

SRE and Continuous Reliability. Performance testing is not a one-time event for your team. It is a recurring signal, like uptime monitoring or error rate tracking. You run scheduled load tests against production-like environments to validate SLOs continuously. When a test shows degradation against the baseline, you investigate before customers notice. This transforms performance from a reactive discipline (something broke) into a proactive one (we caught it early).

SDLC and Developer Testing. Every pull request is a potential performance regression. You want developers to know, within minutes of merging, whether their change degraded response times or increased resource consumption. This requires lightweight, fast-executing tests integrated into CI/CD that compare each build against a known baseline. The goal is shifting performance accountability left, into the hands of the engineers writing the code.

Migration from On-Premises to AWS. You are moving workloads from on-premises infrastructure to AWS. Stakeholders need evidence that the migrated system performs at least as well as what it replaces. This means establishing baselines on the existing on-prem system, replicating equivalent test scenarios on AWS, and producing a comparison that demonstrates parity or improvement.

Migrations often reveal latency differences from network path changes, storage I/O characteristics, or instance sizing mismatches that only surface under load.

Capacity and Sizing Validation. You have deployed the architecture, but are the instance types, container resource limits, and database configurations right-sized for your workload? Under-provisioning risks degradation at peak; over-provisioning wastes money. Load testing at progressively increasing concurrency reveals where each component saturates, what your actual headroom is, and whether Auto Scaling policies respond fast enough to protect the user experience.

Cost Optimization. You suspect you are over-provisioned but cannot reduce capacity without proof that performance holds. Performance testing provides that proof. Run your standard load profile on smaller instances, on Graviton processors, on Spot-backed capacity, or with reduced replica counts. If the test passes your SLOs, you have data-driven justification to right-size. If it fails, you know exactly where the boundary is.

Mapping Use Cases to This Journey

Why Every Stage Matters

Most teams begin with one of these use cases and discover the others along the way. A team preparing for a product launch will establish baselines and SLOs that naturally feed into continuous SRE practice. A team validating a migration will uncover capacity and sizing questions that lead to ongoing cost optimization. A team running developer-level regression tests will realize those same tests, scaled up, become their reliability validation suite.

The stages that follow build a progressive discipline that serves all seven use cases. Defining success criteria, choosing the right test types, designing realistic scenarios, interpreting results, and automating the entire cycle are not optional steps you pick from a menu. They are layers that compound. Skip one and the others lose their foundation. The journey is designed to be read and applied end to end, because performance testing done well is not a checklist — it is a practice that strengthens every time you add a layer.

Go Deeper

- AWS Well-Architected Framework: Performance Efficiency Pillar
- AWS Well-Architected Framework: Reliability Pillar

- Operational Readiness Reviews (ORR)

Defining Success: SLIs, SLOs, and Performance Budgets

Before running a single test, you need to define what "good" looks like. Without clear success criteria, load test results become interesting data that never drives decisions. Service Level Indicators (SLIs), Service Level Objectives (SLOs), and error budgets provide the framework.

SLI, SLO, SLA: The Measurement Stack

Concept	Definition	Example	Who Owns It
SLI (Service Level Indicator)	A quantitative measure of service performance	p99 latency of the / checkout API	Engineering team
SLO (Service Level Objective)	The target value for an SLI	p99 latency < 500ms over a 30-day window	Engineering + Product
SLA (Service Level Agreement)	A contractual commitment with consequences	99.9% availability or service credits issued	Business + Legal

SLIs are what you measure. SLOs are what you target internally. SLAs are what you promise externally. Your load tests should validate SLOs, not SLAs. SLOs are always tighter than SLAs to provide a safety buffer.

Understanding Percentiles (p50, p90, p95, p99)

Performance metrics reported as averages hide critical information. An average response time of 200ms might mean all requests complete in 200ms, or it might mean 99% complete in 100ms while 1% take 10 seconds. Percentiles reveal the distribution.

Percentile	What It Tells You	Example
p50 (median)	The typical experience. Half of requests are faster, half are slower	p50 = 120ms means the "normal" user sees 120ms
p90	How the slower end of normal behaves. 10% of requests are slower than this	p90 = 250ms means 1 in 10 requests takes longer than 250ms
p95	The experience for users who are unlucky but not extreme outliers	p95 = 400ms means 1 in 20 requests is this slow or worse
p99	The tail: the worst 1% of experiences. Often where real pain lives	p99 = 1200ms means 1 in 100 requests takes over a second

Why p99 matters more than average: A system serving 1 million requests per day with p99 = 1200ms means 10,000 requests per day are painfully slow. At scale, tail latency affects real users in real numbers.

Throughout this journey, SLOs and pass/fail criteria are defined in terms of **p99** because it captures the experience of your most affected users, not just the majority.

Choosing the Right SLIs for Load Testing

Not every metric matters equally. Focus load test validation on these categories:

SLI Category	Metric	Why It Matters for Load Testing
Latency	p50, p90, p95, p99 response time	Reveals how the system degrades as load increases
Throughput	Requests per second (RPS) sustained	Identifies the ceiling before degradation begins

SLI Category	Metric	Why It Matters for Load Testing
Error Rate	Percentage of 5xx responses	Shows when the system breaks, not just slows
Saturation	CPU, memory, connection pool utilization	Predicts when autoscaling or limits will trigger

Error Budgets and Performance Budgets

An error budget is the acceptable amount of SLO violation within a time window. If your SLO is p99 < 500ms over 30 days, you can tolerate roughly 7 hours of violation per month. When load tests show you are consuming error budget faster than expected, it is a signal to halt feature work and invest in performance.

Apply the same concept to performance budgets in CI/CD:

- Define a baseline p99 from your last stable release
- Set a performance budget: new deployments must not exceed baseline + 10%
- Automate this check in your pipeline using DLT scheduled tests with baseline comparison

Go Deeper

Improve application reliability with effective SLOs (AWS Blog)
AWS Observability Best Practices: SLI/SLO/SLA (AWS Observability)

Understanding Test Types

The term "load test" is often used as a catch-all, but there are eight distinct test types, each answering a different question about your system. Running the wrong type wastes time and produces misleading confidence.

The Eight Performance Test Types

Test Type	Question It Answers	Load Pattern	Duration	When to Use
Smoke Test	Is the system fundamentally working?	Minimal load (1-5 VUs, Virtual Users)	1-2 min	After every deployment, before running heavier tests
Load Test	Can the system handle expected peak traffic?	Ramp to target concurrency, hold steady	15-60 min	Before every release
Baseline Test	What is our normal performance sweet spot?	Expected average load, held steady	15-30 min	After infra changes, to establish the reference for deviation alerts
Stress Test	Where does the system break?	Ramp beyond expected peak until failure	Until breaking point	Quarterly or after architecture changes
Endurance (Soak) Test	Does performance degrade over time?	Moderate sustained load	4-24 hours	Monthly, after memory-leak fixes
Scalability Test	Does the system scale linearly?	Stepped increases (1x, 2x, 4x, 8x)	30 min per step	After Auto Scaling

Test Type	Question It Answers	Load Pattern	Duration	When to Use
				configuration changes
Spike Test	Can the system handle sudden bursts?	Instant jump to high load, then back to baseline	5-15 min	Before flash sales, marketing campaigns
Volume Test	How does large data volume affect performance?	Normal user load with large payloads/ datasets	30-60 min	Before data migrations, bulk operations

Smoke Tests: Fail Fast Before You Invest

A smoke test is the quickest sanity check in your performance toolkit. It runs minimal traffic (1-5 virtual users) for a short duration against your critical endpoints immediately after deployment. The goal is not to stress the system. The goal is to confirm nothing is fundamentally broken before you invest time in longer, heavier tests.

Think of it as the performance equivalent of a health check. If response times are wildly off or error rates spike under trivial load, something regressed in the deployment. You catch it in 60 seconds instead of discovering it 30 minutes into a full load test, or worse, from real users.

Smoke tests belong at the very start of your CI/CD performance stage: deploy → smoke → proceed to load/stress only if smoke passes.

Baseline Tests: Know Your Sweet Spot

A baseline test establishes the known-good performance profile of your system under expected average conditions. It captures your p50, p90, p99 latencies, throughput ceiling, error rates, and resource utilization at a steady, sustainable load. This becomes your reference point.

Once a baseline is established, every subsequent test result is compared against it. The question shifts from "is this fast?" to "is this different from what we expect?" A deviation beyond a defined threshold triggers investigation, regardless of whether the absolute number looks acceptable.

Baseline Metric	Example Value	Investigation Threshold
p99 Latency	320ms	> 368ms (+15%)
Throughput	4,200 RPS	< 3,570 RPS (-15%)
Error Rate	0.02%	> 0.1%

Baselines are not static. Re-establish them after significant architecture changes, capacity modifications, or major feature releases. Store baseline results alongside your test scripts in version control so the team can trace performance evolution over time.

Load Tests: Validate Your Peak Capacity

A load test answers the most fundamental question before any release: can the system handle the traffic you expect? You ramp concurrency to your projected peak (derived from production analytics, campaign forecasts, or contract SLAs), hold it steady for 15-60 minutes, and verify every SLO remains within bounds.

The key to a useful load test is realistic target concurrency. Pulling numbers from thin air leads to false confidence. Use your CloudWatch metrics, Application Load Balancer access logs, or analytics platform to determine your actual peak concurrent users over the last 30 days, then add a 20-30% buffer for organic growth.

Load tests belong at two points in your workflow: before every production release (to catch regressions against the known baseline), and after any infrastructure change (instance type migration, scaling policy update, database engine upgrade). A load test that passes gives your team confidence to ship. A load test that fails gives you specific data about what broke and where.

Stress Tests: Find Your Breaking Point

A stress test is not designed to pass. It deliberately pushes the system beyond expected peak until something fails. The purpose is to discover your weakest link before your customers do.

Ramp concurrency progressively beyond your load test target. Keep increasing until you observe one of these failure signals: error rates spike above your SLO, response times degrade past acceptable thresholds, a component crashes or becomes unresponsive, or Auto Scaling hits a

resource quota. The specific failure mode tells you where to invest. A database connection limit hit means you need connection pooling or read replicas. A Lambda concurrency throttle means you need reserved concurrency or architecture redesign. A memory exhaustion crash means you have a leak or undersized instances.

Run stress tests quarterly or after any significant architecture change. Document the breaking point and the failure mode. When the breaking point increases over time, your architecture is improving. When it decreases, something regressed.

Endurance (Soak) Tests: Catch Time-Dependent Degradation

Some bugs only surface after hours of sustained operation. Memory leaks that accumulate 10MB per hour are invisible in a 30-minute load test but crash your service overnight. Connection pool exhaustion that takes 6 hours to manifest never appears in shorter runs. Log file growth that consumes disk space over days is undetectable in minutes.

An endurance test applies moderate, steady load (your average traffic level, not peak) for an extended duration: 4-24 hours depending on your confidence level. While it runs, you watch for metrics that trend in one direction rather than oscillating around a stable baseline: memory utilization creeping upward, response times slowly degrading, available connections declining, or disk usage growing.

The interpretation is straightforward. If metrics are stable after 8+ hours, your system handles sustained operation well. If any metric trends monotonically in a bad direction, you have a resource leak that will eventually cause a production incident. Run endurance tests monthly, after deploying memory-related fixes, or when production monitoring shows unexplained gradual degradation.

Scalability Tests: Verify Linear Scaling

A scalability test answers whether your system scales proportionally. If you double the load, does throughput double? Or does latency more than double, indicating a bottleneck that gets worse with scale?

Run stepped increases: 1x baseline, then 2x, 4x, and 8x. At each step, hold steady for 30 minutes and record throughput, latency, and resource utilization. Plot these on a graph. Linear scaling means throughput increases proportionally with load while latency remains flat. Sub-linear scaling (throughput gains diminish at higher load) indicates a shared resource bottleneck: a database lock, a singleton connection, or a component that cannot be horizontally scaled.

Scalability tests are essential after Auto Scaling configuration changes, when migrating from vertical to horizontal scaling architectures, or when preparing for a known traffic event that exceeds any previous peak. They tell you not just whether your system can handle a specific number, but whether your architecture can grow.

Spike Tests: Validate Elasticity Under Sudden Bursts

A spike test simulates what happens when traffic jumps instantly: a flash sale goes live, a marketing email lands in millions of inboxes, or a viral post sends a flood of visitors in seconds. Unlike a load test that ramps gradually, a spike test jumps directly from baseline to peak with no warm-up period.

The questions it answers: How quickly does Auto Scaling respond? Do requests queue, timeout, or drop during the scaling lag? Does the system recover to baseline performance after the spike subsides? Does any component have a cold-start penalty that compounds under sudden demand?

Run spike tests before any planned high-traffic event (product launches, sales campaigns, live broadcasts) and after changing Auto Scaling policies. The pass criteria are: no requests dropped during the spike, error rate remains below SLO, and the system returns to baseline latency within a defined recovery window (typically 30-60 seconds after scaling completes).

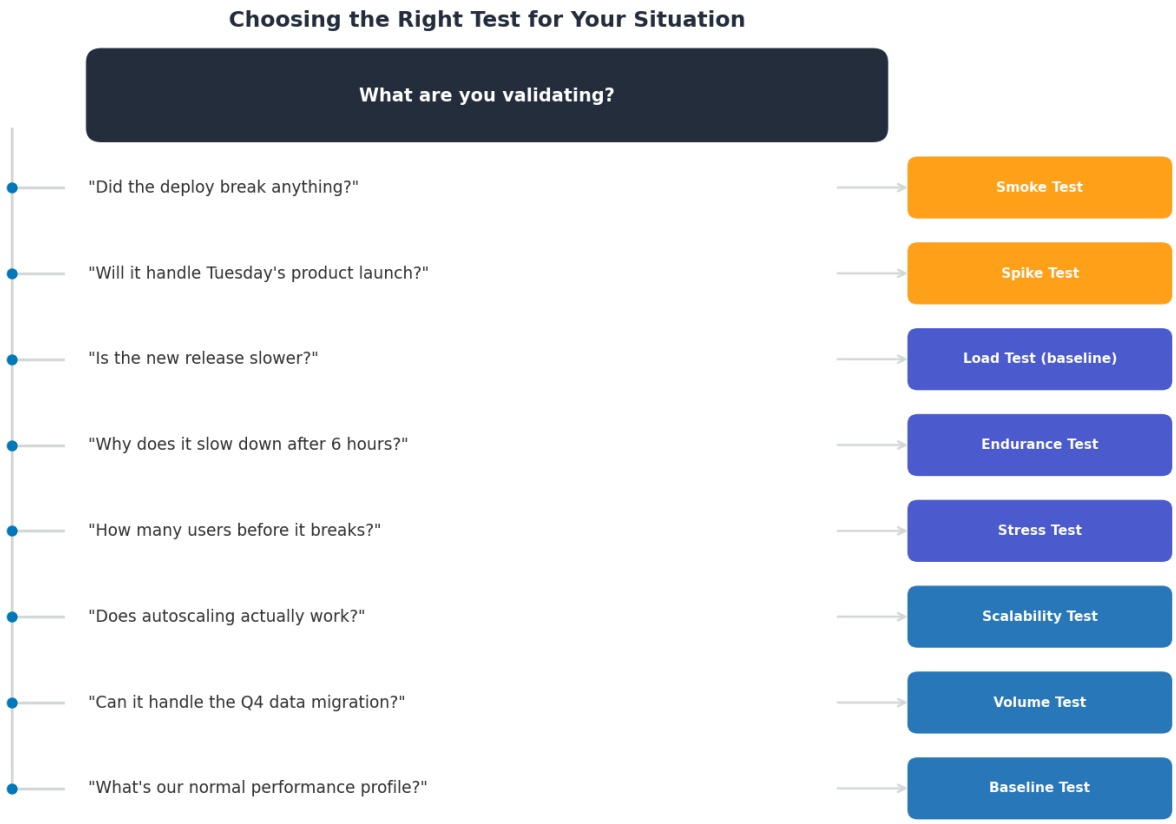
Volume Tests: Measure Data-Scale Impact

A volume test keeps user concurrency normal but dramatically increases data size. Normal traffic with 10x larger payloads, database tables with 100x more rows, or API responses returning full datasets instead of paginated subsets. This catches issues that only appear at data scale, not user scale.

Common findings from volume tests: queries that perform well on 10,000 rows become unacceptable on 10 million rows (missing indexes, full table scans), serialization of large API responses causes memory pressure and garbage collection pauses, file upload processing times grow non-linearly with file size, and batch operations that work fine with 100 items timeout at 10,000 items.

Run volume tests before data migrations, when introducing bulk import/export features, ahead of known data growth milestones (e.g., approaching the first million records), or when production monitoring shows query performance degrading as tables grow. The fix is almost always better indexing, pagination, streaming, or architectural changes to how you handle large datasets.

Choosing the Right Test for Your Situation



Common Mistakes in Test Type Selection

Teams frequently make errors that render their test results misleading:

Running only load tests. A load test validates expected capacity but tells you nothing about what happens when that capacity is exceeded. Without stress tests, you discover breaking points from customers, not from controlled experiments. Schedule quarterly stress tests alongside your regular load tests.

Skipping endurance tests. Memory leaks, connection pool exhaustion, thread starvation, and log file growth are time-dependent failures that never appear in a 15 minute load test. If your application runs 24/7, you need endurance tests that run for at least 4 hours. Many production outages trace back to resource exhaustion that only manifests after sustained operation.

Confusing spike tests with stress tests. A stress test gradually increases load to find the ceiling. A spike test jumps instantly to a high level and measures recovery. They answer different questions:

"how much can it handle?" versus "can it handle sudden shock?" Flash sales, marketing email blasts, and viral social media moments are spike scenarios, not stress scenarios.

Using inappropriate test durations. A 2 minute load test does not exercise Auto Scaling, connection pool recycling, or JIT compilation warm-up. Most load tests need a minimum of 15 minutes at steady state (after ramp-up) to produce meaningful results. Soak tests need a minimum of 4 hours to catch time-dependent degradation.

Combining Test Types into a Testing Calendar

Mature teams run multiple test types on different schedules:

Start with the simplest schedule: a smoke load test on every PR merge. This catches the most damaging regressions (a broken endpoint, a missing environment variable, a misconfigured timeout value) before they compound.

Cadence	Test Type	Purpose
Every PR merge	Smoke load test (50 VUs, 2 min)	Catch obvious regressions
Nightly	Full load test (production-scale, 15 min)	Continuous capacity validation
Weekly	Endurance test (moderate load, 4-8 hours)	Detect time-dependent degradation
Quarterly	Stress test (ramp to failure)	Update known capacity ceiling
Pre-event	Spike test (150% expected peak, instant ramp)	Validate event readiness

Go Deeper

Load testing applications (AWS Prescriptive Guidance)
Validate system reliability with performance testing (QA.NT.2) (Well-Architected DevOps Guidance)

Choosing Your Testing Tool

Performance testing tools fall into two categories: scripting frameworks (how you define test scenarios) and runtime infrastructure (how you execute them at scale). Most teams start with an open-source scripting framework like K6, Locust, or JMeter. The challenge is not writing the script. It is operating the infrastructure to run that script at production-scale concurrency: provisioning load generators, distributing traffic across regions, collecting results, and tearing everything down afterward. That infrastructure management becomes a project in itself.

Distributed Load Testing on AWS (DLT) eliminates that operational burden entirely. DLT handles all infrastructure provisioning, scaling, orchestration, monitoring, and reporting. You write a test script in the framework you already know; DLT runs it at any scale you need. No servers to manage, no load generator fleets to maintain, no scaling logic to build.

Distributed Load Testing on AWS (DLT) is an open-source, fully supported AWS Solution that automates performance testing at scale. It deploys as a self-contained stack in your AWS account, providing a web console, REST API, CLI, and optional MCP server for AI-assisted analysis. You do not need to provision or manage any testing infrastructure. DLT is actively maintained by AWS and receives regular updates, security patches, and feature enhancements. The latest release (v4.2) introduces native integration with AWS DevOps Agent, enabling automated root cause analysis of load test results directly from the DLT console. When a test completes, you can send the full run context to a registered Agent Space, which correlates CloudWatch metrics, logs, and traces to identify bottlenecks and suggest remediation steps without manual investigation.

If your organization uses commercial load testing platforms, DLT is worth evaluating as a complement or replacement, particularly for teams looking to reduce licensing costs while maintaining the same scripting flexibility. DLT is fully open-source and supported by AWS at no additional software cost (you pay only for the underlying compute).

DLT supports scripts written in three open-source frameworks natively:

Framework	Script Language
K6	JavaScript / TypeScript
Locust	Python
JMeter	.jmx (XML)

Network dependency for K6

K6 binaries are downloaded from Grafana's CDN at test runtime. If your VPC has restricted outbound internet access (private subnets without NAT, strict egress security groups, or proxy-only configurations), K6 tests will fail at the download step. Ensure outbound HTTPS access to `github.com/grafana/k6/releases` from your Fargate tasks, or consider Locust/JMeter which are bundled in the container image. A third option is to customize the container image to include K6 directly. However, K6 is licensed under AGPL-3.0, so you must confirm that bundling it complies with your organization's licensing policies.

If your team already has test scripts in any of these frameworks, bring them directly to DLT. No migration, no rewrite. DLT runs .js, .py, and .jmx files through its Taurus testing engine.

Traffic shape awareness

DLT's Traffic Shape configuration controls concurrency, ramp-up duration, and hold duration for your test. Taurus, the underlying execution engine, applies these values and overrides any multi-step or dynamic traffic patterns defined inside your script (K6 stages, Locust custom load shapes, JMeter Ultimate Thread Group configurations). This means complex traffic shaping such as spike tests, stepped ramps, or sawtooth curves cannot be driven from script-level logic alone when running through DLT. If your scenario requires dynamic traffic patterns, use K6 or Locust directly on self-managed infrastructure where the framework controls execution end-to-end, or model the pattern using multiple sequential DLT test runs with different Traffic Shape configurations for each phase. This is a known limitation in the current release (4.2.x). The DLT team is continuously improving the solution and dynamic traffic shaping is an area of active interest. We encourage you to check the latest release notes for updates.

Running Scripts at Scale with DLT

DLT executes each script inside an Amazon ECS task on AWS Fargate. We recommend up to 200 virtual users (VUs) on each ECS task. However, based on your needs, the ECS task can be sized to support more users.

As a starting guideline:

- 1,000 concurrent users → 5 tasks
- 10,000 concurrent users → 50 tasks
- 100,000 concurrent users → 500 tasks

DLT also supports generating traffic from multiple AWS Regions simultaneously, allowing you to simulate geographically distributed user traffic and validate how your application performs for users in different locations.

Example: 1,000,000 concurrent users across 5 Regions

Traffic distribution should mirror your actual user base. If 40% of your customers are in North America, 25% in Europe, and the rest spread across Asia and Latin America, your test should reflect that:

Region	VUs	Tasks (at 200 VUs/ task)	Rationale
us-east-1	400,000	2,000	40% of traffic from North America
eu-west-1	250,000	1,250	25% of traffic from Europe
ap-southeast-1	150,000	750	15% from Southeast Asia
ap-northeast-1	150,000	750	15% from Northeast Asia
sa-east-1	50,000	250	5% from Latin America
Total	**1,000,000**	**5,000**	

Distribute VUs proportionally based on real market share, customer geography, or historical traffic analytics. This ensures your test validates performance where your users actually are, not just under idealized uniform load.

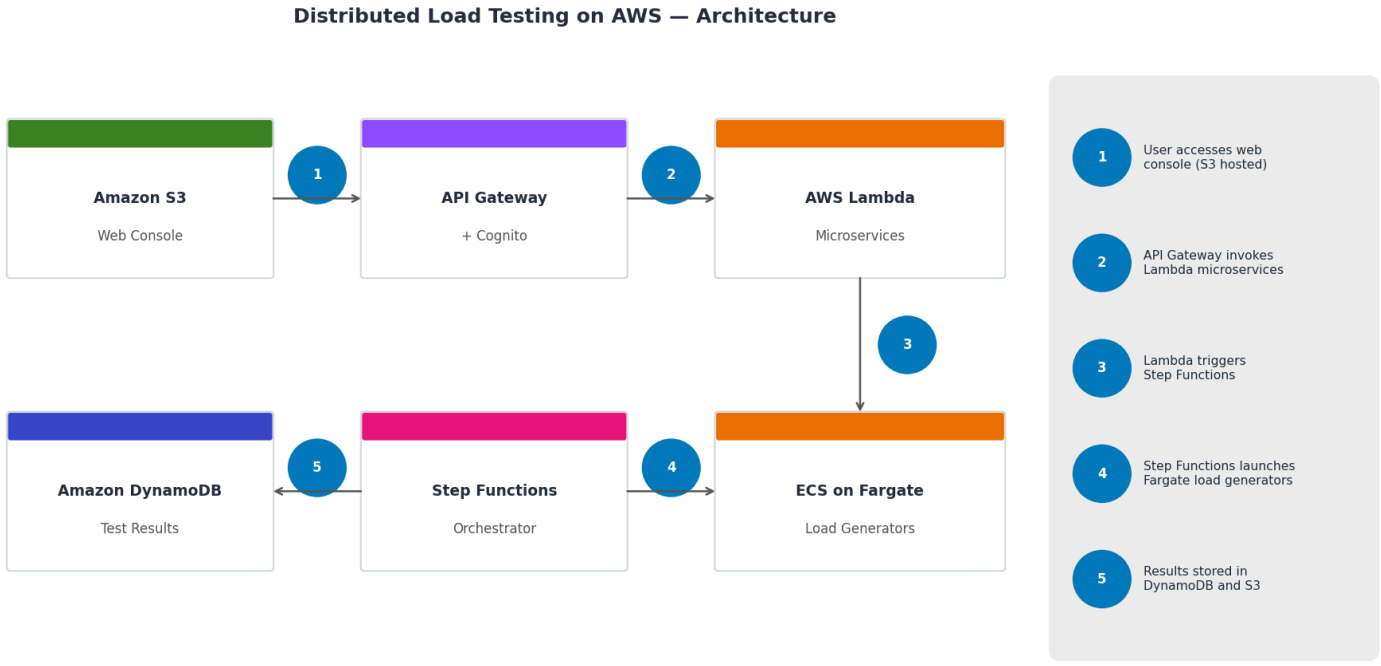
Monitor task-level CPU and memory to determine if you need to adjust task count or sizing. DLT handles task orchestration, scheduling, and result aggregation automatically across all regions.

Architecture Overview

DLT consists of two main components:

Front End: A React-based web console (hosted on Amazon CloudFront + Amazon S3, or ALB + ECS) that provides test management, real-time monitoring, and result visualization. Authentication is handled by Amazon Cognito.

Back End: An orchestration engine that uses AWS Step Functions to coordinate test execution. When you start a test, the engine launches Amazon ECS tasks on AWS Fargate, each running your test script inside a container with the Taurus load testing framework. Results flow back through Amazon DynamoDB and Amazon S3.



Deployment Options

DLT offers multiple deployment paths: a guided wizard experience and direct CloudFormation templates for teams that prefer infrastructure-as-code control.

Deployment Method	Use Case	Experience
AWS Launch Wizard (Recommended)	First-time deployments, teams unfamiliar with CloudFormation	Step-by-step guided wizard in the AWS Console with validation and version management
CloudFormation: Default (CloudFront + S3)	Most self-service deployments	Web console hosted via CloudFront
CloudFormation: ALB + ECS	Regions without CloudFront, network restrictions, private deployments	Web console via Application Load Balancer
CloudFormation: Headless	CI/CD only, no web UI needed	Backend services only (API + CLI access)

AWS Launch Wizard handles prerequisite validation, parameter configuration, and stack deployment in a single guided flow. It also simplifies upgrades: when a new DLT version is available, Launch Wizard presents the update path without requiring manual template URL management.

Deployment Checklist

- Review the Amazon EC2 Testing Policy (required before running large-scale tests)
- Determine your primary deployment region (this is where orchestration runs)
- Identify additional regions for multi-region testing (optional)
- Deploy the main CloudFormation stack (takes approximately 15 minutes)
- Verify you received the admin invitation email from Amazon Cognito
- Log in to the web console and create your first test scenario
- For multi-region: deploy the regional stack in each additional region

Multi-Region Testing

DLT supports distributing load from multiple AWS Regions simultaneously. This is critical for validating:

- Global application performance from different geographies
- CDN cache behavior under distributed load
- Regional failover and routing policies

Deploy the regional CloudFormation template in each additional region. Regional stacks register themselves with the main stack and appear in the web console automatically.

DLT CLI for CI/CD Integration

The DLT CLI provides headless interaction for pipeline automation:

- Start test scenarios programmatically
- Query test results and compare against baselines
- Download test artifacts for analysis
- Supports browser OAuth, SRP headless, and IAM authentication modes

Put It Into Practice

- Deploy DLT in your preferred region using the deployment guide
- Create a simple HTTP test scenario in the web console (no script required for basic HTTP endpoint testing)
- Run the test with 1 task and review the results dashboard
- Explore the percentile metrics (p50, p90, p95, p99) and error rates
- Set up the DLT CLI for your CI/CD tool of choice

Troubleshooting Common Deployment Issues

Teams occasionally encounter issues during initial DLT deployment. These are the most common:

Symptom	Cause	Fix
CloudFormation stack fails during deployment	Insufficient IAM permissions for the deploying principal	Ensure the deploying role has permissions for ECS, Step Functions, DynamoDB, S3, Cognito, API Gateway, Lambda, and CloudWatch
Cannot log in after deployment	Cognito invitation email went to spam or expired	Check spam folder, or manually set the password in the Cognito User Pool console
Test tasks fail immediately	Fargate task cannot pull container image	Verify NAT Gateway or VPC endpoints for ECR access in private subnets
Results show 0 requests	Test script has errors that prevent execution	Download the task logs from CloudWatch (/aws/ecs/DLT) and check for script syntax errors
Test never completes	Step Functions execution timeout	Check the test duration configuration; default execution timeout may be too short for long soak tests

If deploying in a private subnet configuration (no internet egress), ensure VPC endpoints exist for Amazon ECR, Amazon S3 (gateway endpoint), Amazon DynamoDB (gateway endpoint), and CloudWatch Logs. Without these, Fargate tasks cannot pull images or write results.

Go Deeper

Distributed Load Testing on AWS: Implementation Guide
Ensure Optimal Application Performance with DLT (Architecture Blog)
DLT Features: Test Framework Support (DLT Documentation)
Performance Testing on AWS (re:Invent 2025, CMP351) (Video, 20 min)

Designing Realistic Test Scenarios

A load test is only as useful as the traffic pattern it simulates. Testing with uniform request patterns against a single endpoint produces misleading results. Real users navigate flows, carry sessions, generate varied payloads, and arrive in unpredictable patterns.

Principles of Realistic Workload Design

Model real user journeys, not individual requests. A checkout flow involves authentication, product browsing, cart operations, and payment. Test the full sequence.

Use production traffic analysis to determine request ratios. If 60% of traffic hits your product catalog, 25% hits search, and 15% hits checkout, your test script should reflect these proportions.

Include think time between requests. Real users pause between actions. Without think time, your test generates artificial burst patterns that do not represent production behavior. A 1-5 second think time between requests is typical for web applications.

Vary data inputs. Use parameterized test data. If every virtual user searches for the same product ID, you are testing your cache, not your system. CSV data files with varied inputs produce more realistic database and service load.

Test Data Strategy

Data Need	Approach
User credentials	Generate test accounts in bulk, never use production credentials
Product/entity IDs	Extract a representative sample from production, sanitize PII
Geographic distribution	Deploy DLT regional stacks to simulate users from multiple regions
Payload variety	Create parameterized templates that inject random but valid data

Data Need	Approach
Session state	Use cookie handling and token management in your scripts

Environment Considerations

Factor	Recommendation
Infrastructure parity	Test environment should match production instance types, replica counts, and configura tion
Database state	Pre-load test data that represents production data volume
External dependencies	Use stubs for third-party services, or coordinat e testing windows
Auto Scaling	Verify scaling policies match production (or test them)
Service quotas	Check AWS service quotas before generating large load
Cost monitoring	Set AWS Budgets with automated actions (stop instances, halt deployments) before long-running tests. Note: Cost Explorer data can lag up to 8 hours, so billing alerts alone may not catch runaway costs in time. Use Budgets Actions for hard stops.

Common Pitfalls

- **Testing only the happy path.** Include error scenarios: invalid tokens, 404 pages, timeout flows.

- **Ignoring warm-up.** JIT compilation, connection pool initialization, and cache warming affect early metrics. Include a ramp-up period and exclude warm-up data from analysis.
- **Testing in isolation.** If your production system depends on downstream services, the load test should include those interactions.
- **Using unrealistic concurrency ramps.** Real traffic rarely jumps from 0 to 10,000 instantly (unless you are testing spike scenarios specifically). Model gradual ramp patterns.

Script Structure Best Practices

Regardless of which tool you choose (K6, Locust, or JMeter), well-structured test scripts share common patterns:

Pattern	What It Does	Why It Matters
Setup/teardown phases	Create test data before load, clean up after	Prevents test pollution across runs
Transaction grouping	Group related requests into named transactions	Enables per-flow latency analysis
Parameterized data	Load inputs from CSV/JSON files	Prevents cache-only testing
Assertions within script	Check response codes and body content	Catches functional errors under load
Custom metrics	Tag business operations (e.g., "checkout_complete")	Connects technical metrics to business outcomes

A common anti-pattern is testing individual API endpoints in isolation (unless you are Unit Testing). Real users execute multi-step flows. If your checkout API depends on a session created by the login API, test the full flow. Session dependencies, cookie propagation, and token refresh all affect real-world performance and must be modeled in your scripts.

Determining the Right Concurrency Target

One of the most frequent questions: "how many virtual users should I simulate?" The answer comes from your production analytics:

1. **Find peak concurrent sessions.** Check your ALB or CloudFront access logs for the highest simultaneous active connections in the past 90 days.
2. **Apply a safety multiplier.** Test at 1.5x to 2x your observed peak to account for growth and unexpected spikes.
3. **Convert sessions to VUs.** Each DLT virtual user represents one concurrent session executing your scripted flow.
4. **Calculate task count.** Divide your target VU count by 200 (the VUs-per-task guideline) to determine how many Fargate tasks you need.

Example: If your peak is 3,000 concurrent sessions, test at 4,500 to 6,000 VUs (23 to 30 DLT tasks).

Put It Into Practice

- Export one hour of peak production access logs from your ALB or CloudFront distribution
- Identify the top 5 request paths by volume percentage and use these ratios in your test script
- Add think time to your test script: `sleep(randomBetween(1, 5))` between each request in K6, or `time.sleep(random.uniform(1, 5))` in Locust
- Create a CSV file with at least 1,000 unique parameterized inputs (user IDs, product IDs, search terms)
- Run two tests: one with parameterized data and one without. Compare the p99 difference to see how much your cache was masking

Go Deeper

Load testing applications: Design realistic scenarios (AWS Prescriptive Guidance)

Interpreting Results and Taking Action

Raw load test data is noise without a framework for interpretation. The goal is not to generate graphs but to make decisions: is this release safe to deploy? Do we need to scale differently? Which component is the bottleneck?

The Results Hierarchy

Step 1: Did it pass? Compare p99 latency, error rate, and throughput against your SLOs. If all metrics are within bounds, the test passes. Use DLT's baseline comparison feature to detect regressions automatically.

Step 2: Where did it degrade? Look for inflection points in the results timeline. The moment p99 jumps from 200ms to 800ms often correlates with a specific concurrency level or time in the test. This is your system's practical capacity ceiling.

Step 3: Why did it degrade? Correlate DLT results with infrastructure metrics:

Symptom in DLT	Likely Cause	Where to Look
p99 increases while p50 stays flat	Resource contention affecting tail requests	CPU utilization, garbage collection, thread pool saturation
Error rate spikes at specific concurrency	Hard limit reached	Database connections, service quotas, load balancer limits
Throughput plateaus despite more VUs	Bottleneck somewhere in the path	CloudWatch Application Signals service map, queue depths, downstream service metrics
Latency increases linearly with load	Insufficient horizontal scaling	Auto Scaling policies, ECS task count, Lambda concurrency

Symptom in DLT	Likely Cause	Where to Look
Intermittent timeouts	Network or DNS issues	VPC flow logs, NAT gateway throughput, DNS resolution times

Observability During Load Tests

Effective load test analysis requires instrumentation beyond what DLT provides. Set up these observability layers before your test:

Layer	Tool	What It Reveals
Distributed Tracing	AWS X-Ray (instrumented via ADOT)	Which service in the call chain is slow, with vendor-neutral OpenTelemetry instrumen tation
Infrastructure Metrics	Amazon CloudWatch	CPU, memory, network, disk I/O per component
Application Observability	CloudWatch Application Signals	End-to-end service map, automatic service discovery and metric collection (latency, availability) with user-defined SLOs tracked against those metrics
Log Analysis	Amazon CloudWatch Logs Insights	Error patterns, slow query identification

OpenTelemetry is the recommended instrumentation standard (the X-Ray SDK entered maintenance mode in February 2026). ADOT collects traces, metrics, and logs using the OpenTelemetry SDK and exports them to the appropriate AWS backends: traces to AWS X-Ray, metrics and logs to CloudWatch. CloudWatch Application Signals consumes this telemetry to provide a unified service map, automatically collected service-level indicators, and user-defined

SLO tracking. Traces are accessible via Transaction Search directly in the CloudWatch console, so you do not need to switch between X-Ray and CloudWatch during post-test analysis.

Service-Specific Insights

Beyond general-purpose CloudWatch metrics, AWS provides specialized observability tools that surface performance data specific to each compute and data layer. Enable these before running load tests to get deeper visibility into service-level bottlenecks:

Service	Insight Tool	What It Reveals During Load Tests
Amazon ECS / EKS	**Container Insights**	Per-task and per-pod CPU, memory, network, and disk metrics. Identifies which containers are saturated, which tasks are restarting, and whether cluster Auto Scaling is keeping pace with demand.
AWS Lambda	**Lambda Insights** + standard CloudWatch metrics	Cold start frequency and duration, init duration, memory utilized vs. allocated , CPU time, disk and network I/O per invocation. Throttle monitoring (concurrency limit rejections) comes from the standard CloudWatch Lambda `Throttles` metric, not from Lambda Insights. When concurrency limits are hit, Lambda rejects synchrono us invocations immediately rather than queuing them.

Service	Insight Tool	What It Reveals During Load Tests
Amazon RDS / Aurora	**Performance Insights**	Top SQL queries by load (Active Sessions), wait events (I/O, lock, CPU), and database load decomposi tion. Pinpoints exactly which queries degrade under concurrent access.
Amazon DynamoDB	**CloudWatch Contributor Insights**	Hot partition keys and throttled items. Reveals uneven access patterns that only surface at scale when certain partition keys receive disproportionate traffic. Note: Contributor Insights offers two modes: full access pattern visibility (accessed and throttled keys) and a cost-optimized throttled-keys-only mode that emits data only during throttling events. The throttled-keys-only mode is the AWS-recom mended default for always-on monitoring.

Enable these tools in your staging environment before running tests. The overhead varies by service: DynamoDB Contributor Insights introduces no performance impact per AWS documentation, Lambda Insights adds a lightweight extension per invocation, and Container Insights and Performance Insights are designed for always-on production use. The diagnostic value during post-test analysis far exceeds what aggregate CloudWatch metrics alone can provide.

Building a Performance Analysis Workflow

Raw DLT output provides aggregate percentiles and throughput curves. Turning those numbers into architectural decisions requires a systematic analysis workflow:

Step 1: Identify the inflection point. In the DLT results timeline, find the exact moment where p99 diverges from p50. This concurrency level is your system's "knee" where contention begins. Note the timestamp and correlate it with infrastructure metrics.

Step 2: Isolate the bottleneck layer. Use the CloudWatch Application Signals service map to identify which service in the call chain contributed the most latency at the inflection point. Application Signals correlates traces, metrics, and SLO status in a single view, so you can pinpoint the degrading service without switching between consoles. Common bottlenecks: database queries that lack indexes, synchronous calls to downstream services that should be async, connection pools that are too small for the concurrency level.

Step 3: Quantify the impact. Calculate the "latency budget breakdown" at peak load:

Component	Latency at Baseline	Latency at Peak	Delta
API Gateway	5ms	8ms	+3ms
Lambda cold start	0ms (warm)	200ms (cold starts under scale)	+200ms
DynamoDB query	10ms	10ms	+0ms
Downstream API	50ms	450ms	+400ms
Total	**65ms**	**668ms**	**+603ms**

This breakdown immediately reveals that the downstream API is the dominant contributor to degradation. Without this breakdown, teams often optimize the wrong component.

Step 4: Classify the finding. Is the bottleneck fixable by you (code, configuration, architecture), or does it require an external dependency change? Route each finding to the right team immediately, not after the full analysis is complete.

CloudWatch Logs Insights Queries for Post-Test Analysis

After a load test, these CloudWatch Logs Insights queries help pinpoint issues:

```
# Find the slowest requests during the test window
fields @timestamp, @message
| filter @timestamp between '2026-07-01T10:00:00Z' and '2026-07-01T11:00:00Z'
| filter latency_ms > 1000
| sort latency_ms desc
| limit 20
```

Put It Into Practice

- Before running your next load test, define pass/fail criteria tied to SLOs
- Instrument your application with OpenTelemetry (via ADOT) and enable CloudWatch Application Signals
- Create a CloudWatch dashboard with key metrics for your application
- Run a load test and practice correlating DLT results with CloudWatch metrics
- Identify the first bottleneck and document it as a performance improvement backlog item

Automated Root Cause Analysis with AWS DevOps Agent

Manually correlating metrics, logs, and traces after a load test is time-consuming and error-prone. DLT v4.2 introduces native integration with AWS DevOps Agent, enabling automated investigation of load test results directly from the DLT console.

When you connect an Agent Space to your DLT deployment, you can send any completed test run to the agent for analysis. The agent proactively identifies performance bottlenecks, determines root causes, and suggests remediation steps without requiring you to manually parse CloudWatch metrics, dig through log groups, or trace individual requests through your service graph.

How it works:

Step	Action	What Happens
1	Register an Agent Space	Tag your Agent Space with `dlt-integration:`

Step	Action	What Happens
		`allowed`, then register its ARN in the DLT console under Agent Integration
2	Run a load test	Execute any test scenario (load, stress, endurance, spike) and wait for completio n
3	Investigate	Click "Investigate with DevOps Agent" on the test run detail page. Optionally provide context about recent infrastructure changes
4	Review findings	The agent produces a structured root-cause analysis with symptoms, contributing components, and concrete remediation suggestions

When to use automated investigation:

Use DevOps Agent as your **first-pass triage** after every test run. Let the agent surface likely causes, then validate its findings using the manual correlation techniques described in the previous sections. The agent excels at identifying patterns across hundreds of metrics and log streams simultaneously, a task that would take a human analyst significantly longer.

Automated investigation is particularly valuable for:

- **Post-regression detection.** When your CI/CD pipeline flags a performance regression, immediately trigger an investigation to understand *why* the regression occurred before it reaches production
- **Endurance test analysis.** After multi-hour soak tests, the volume of metrics data makes manual analysis impractical. DevOps Agent can identify time-dependent degradation patterns (memory leaks, connection pool exhaustion, log file growth) that emerge gradually

- **Multi-service bottleneck identification.** When your application spans dozens of microservices, the agent correlates signals across all components to pinpoint which service in the chain is the true root cause

Prerequisites for DevOps Agent integration:

- The Agent Space must be in the same AWS account as your DLT deployment
- The Agent Space must be tagged with the key `dlt-integration` and value `allowed` (this establishes dual consent between the DLT operator and the Agent Space administrator)
- The tag must be present before DLT will accept connection or investigation requests

What findings include:

- A structured root-cause analysis identifying which components contributed to latency or errors
- A list of symptoms observed during the test window (CPU saturation, connection pool exhaustion, throttling events, error spikes)
- Concrete remediation suggestions with specific configuration changes, architectural adjustments, or scaling actions
- Investigations remain in the DLT Investigations tab for the lifetime of the test run record, building a historical knowledge base as your application evolves

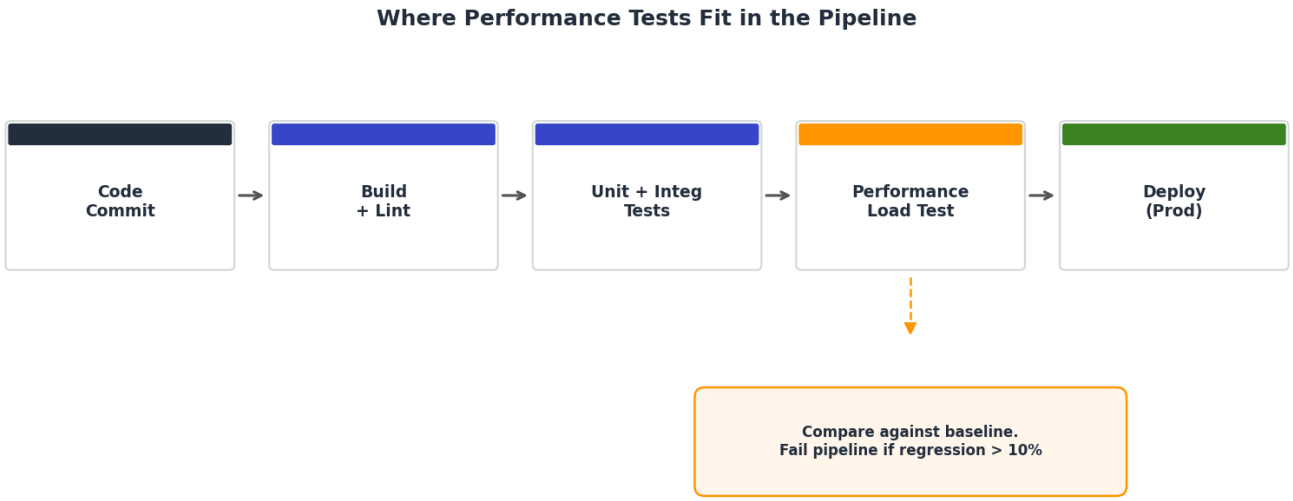
Go Deeper

AWS DevOps Agent Integration (DLT Documentation)
Amazon CloudWatch Service Level Objectives (CloudWatch Documentation)
Monitor application health using SLOs with CloudWatch Application Signals (AWS Blog)

Automating Performance in CI/CD

Performance testing delivers the most value when it runs automatically on every change, not as a manual gate before major releases. The "Test After Every Change" principle requires pipeline integration that makes performance results as visible as unit test results.

Where Performance Tests Fit in the Pipeline



Integration Approach: Tool Agnostic

DLT provides multiple integration points that work with any CI/CD system:

Integration Method	How It Works	Best For
DLT REST API	Start tests and poll results via HTTP calls	Any CI/CD tool with HTTP support
DLT CLI	Command-line interface with IAM/Cognito auth	Pipeline scripts, headless environments
Scheduled Tests	Cron-based recurring tests configured in DLT	Nightly regression suites

Integration Method	How It Works	Best For
MCP Server	AI-assisted analysis of test results	Developer productivity workflows

Pipeline Integration Pattern

The general pattern for any CI/CD tool:

1. **Trigger:** Pipeline stage invokes DLT CLI or API to start a pre-defined test scenario
2. **Wait:** Poll for test completion (DLT CLI supports waiting with timeout)
3. **Evaluate:** Compare results against the designated baseline run
4. **Gate:** If p99 latency or error rate exceeds threshold, fail the pipeline
5. **Report:** Publish results as pipeline artifacts for team review

Concrete Example: GitHub Actions with DLT CLI

This workflow runs a performance gate on every push to `main`. It starts a DLT scenario, waits for completion, and fails the build if p99 exceeds the baseline by more than 15%.

```
# .github/workflows/perf-gate.yml
name: Performance Gate
on:
  push:
    branches: [main]

jobs:
  load-test:
    runs-on: ubuntu-latest
    permissions:
      id-token: write    # For OIDC auth to AWS
      contents: read
    steps:
      - uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::111122223333:role/dlt-ci-role
          aws-region: us-east-1
```

```
      - name: Install DLT CLI
        run: pip install distributed-load-testing-cli

      - name: Run load test
        run: |
          dlt run \
            --test-id "regression-suite" \
            --wait \
            --timeout 1200 \
            --output results.json

      - name: Evaluate results against baseline
        run: |
          # Extract p99 from results and compare to baseline
          P99=$(jq '.results.avg_lt_p99' results.json)
          BASELINE_P99=$(jq '.baseline.avg_lt_p99' results.json)
          THRESHOLD=$(echo "$BASELINE_P99 * 1.15" | bc)
          if (( $(echo "$P99 > $THRESHOLD" | bc -l) )); then
            echo "::error::Performance regression detected. p99=${P99}ms exceeds
baseline+15%=${THRESHOLD}ms"
            exit 1
          fi
          echo "Performance gate PASSED. p99=${P99}ms within threshold=${THRESHOLD}ms"

      - name: Upload results artifact
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: dlt-results
          path: results.json
```

Adapt the same pattern for GitLab CI (use `script:` blocks), Jenkins (use `sh` steps in a Declarative Pipeline), or AWS CodePipeline (invoke DLT via a Lambda action or CodeBuild step).

Baseline Management

DLT supports designating any test run as a "baseline" for comparison. Establish your baseline strategy:

- Run a baseline test on every stable release (tag it in DLT)
- Compare every subsequent test against this baseline
- Update the baseline only when performance improvements are intentional

- Track baseline drift over time to detect gradual degradation

Scheduling Recurring Tests

For the "Always Test" principle, configure DLT scheduled tests:

Test Cadence	Purpose	Typical Configuration
Nightly	Catch regressions from daily merges	Full load test, moderate concurrency
Weekly	Endurance validation	4-8 hour soak test at 60% expected peak
Pre-event	Validate capacity for planned spikes	Spike test at 150% expected peak
Monthly	Stress testing to find new ceilings	Ramp to failure

Handling Performance Test Failures in the Pipeline

When a performance test fails your gate criteria, the team needs a clear escalation path. Without one, teams either ignore the failure ("just override it") or block releases indefinitely.

Failure Severity	Response	Example
Minor (p99 regressed 10-20%)	Investigate, create performan ce ticket, allow deploy with team lead approval	New logging added 15ms to p99
Moderate (p99 regressed 20-50%)	Block deploy, investigate same day, fix or revert within 24 hours	N+1 query introduced in new feature

Failure Severity	Response	Example
Critical (p99 regressed >50% or errors >5%)	Revert immediately, root cause analysis before re-deploy	Connection pool misconfig uration

Automate severity classification in your pipeline. Compare the regression percentage against thresholds and route notifications appropriately: minor failures go to a Slack channel, moderate failures page the team lead, critical failures trigger an automatic revert.

Dealing with Flaky Performance Tests

Performance tests have inherent variance. A 2% latency difference between runs is noise, not signal. Reduce false positives with these techniques:

- **Run multiple iterations.** Execute the same test 3 times and use the median result for gate decisions. DLT scheduled tests support this pattern.
- **Use statistical significance thresholds.** A 5% regression that falls within normal variance (based on historical run data) should not fail the pipeline.
- **Warm up the environment.** Run a 30-second warm-up phase before the measured phase. Exclude warm-up metrics from gate evaluation.
- **Isolate test infrastructure.** Shared staging environments produce inconsistent results. Dedicated performance test environments with consistent instance types eliminate infrastructure variance.
- **Monitor the test infrastructure itself.** If your Fargate tasks are CPU-constrained, results reflect the test harness limit, not the application's actual capacity.

Put It Into Practice

- Identify which CI/CD tool your team uses
- Deploy the DLT CLI in your pipeline agent/runner (IAM auth recommended for automation)
- Create a "smoke" performance test: 50 VUs, 2 minutes, run on every PR merge
- Create a "full" performance test: production-scale VUs, 15 minutes, run nightly
- Set up pipeline failure criteria: fail if p99 > baseline + 15% or error rate > 1%

Go Deeper

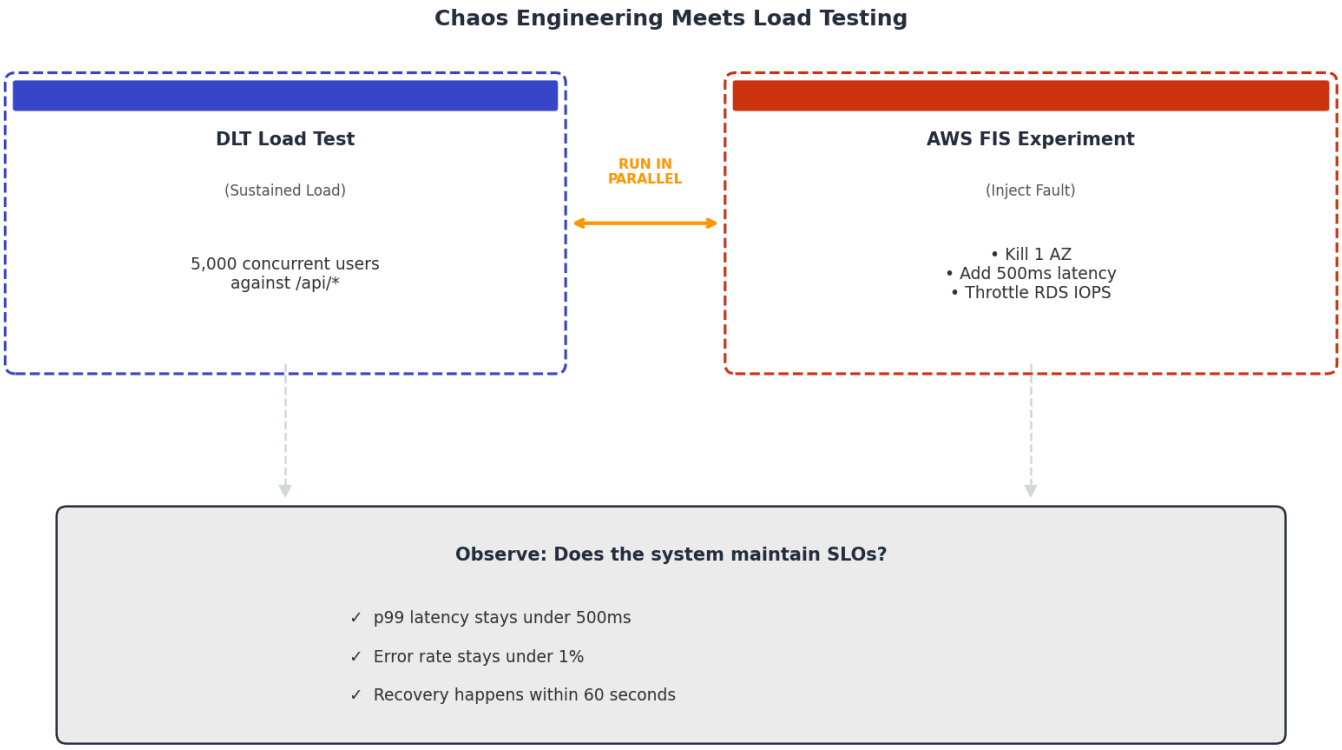
DLT CLI Documentation
Performance Efficiency Pillar: Process and Culture (Well-Architected)

Chaos Engineering Meets Load Testing

Load testing tells you how the system performs under expected conditions. Chaos engineering tells you how it performs when things go wrong. Combining both reveals whether your system degrades gracefully or catastrophically when failures occur under load.

The Combined Pattern: DLT + AWS Fault Injection Service

AWS Fault Injection Service (FIS) lets you inject controlled faults into your AWS environment: terminate instances, throttle APIs, add network latency, disrupt AZ connectivity. Running FIS experiments during an active DLT load test creates the most realistic validation of system resilience.



Common DLT + FIS Experiment Combinations

Scenario	DLT Configuration	FIS Action	What You Learn
AZ failure under load	Steady-state load at 70% capacity	Disrupt one AZ	Does traffic rebalance without user impact?
Database failover	Normal transactional load	Force RDS failover	How long is the brownout during failover?
Network degradation	Full production-scale load	Add 200ms latency to VPC	Does the system timeout or gracefully degrade?
Instance termination	Peak load simulation	Terminate 30% of EC2 instances	Does Auto Scaling recover before SLOs breach?
Dependency throttlin g	Normal load	Throttle downstream API calls	Does circuit breaking activate correctly?

Designing Your First Chaos + Load Experiment

1. **Start with a steady-state hypothesis.** Example: "Under 5000 VUs, if one AZ fails, p99 latency will not exceed 1 second and error rate will not exceed 0.5% after a 60-second recovery window."
2. **Establish the baseline.** Run the DLT load test without any faults. Record p99, p50, error rate, and throughput.
3. **Inject the fault.** While DLT is running at steady state, trigger the FIS experiment. Observe the delta from baseline.
4. **Validate recovery.** After the FIS experiment completes, the system should return to baseline metrics within your recovery time objective.
5. **Document findings.** Record whether the hypothesis held, what the actual impact was, and what improvements are needed.

Starter FIS Experiment: AZ Failure Under Load

This CloudFormation snippet defines a reusable FIS experiment template that disrupts connectivity to a single Availability Zone. Run it while a DLT load test is active to validate multi-AZ resilience.

```
# fis-az-failure-template.yaml
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  AZFailureExperiment:
    Type: AWS::FIS::ExperimentTemplate
    Properties:
      Description: "Simulate single-AZ failure during active load test"
      RoleArn: !GetAtt FISRole.Arn
      StopConditions:
        - Source: aws:cloudwatch:alarm
          Value: !GetAtt SafetyAlarm.Arn
      Targets:
        TargetSubnets:
          ResourceType: aws:ec2:subnet
          ResourceTags:
            az-chaos-target: "enabled"
          SelectionMode: COUNT(2)
      Actions:
        DisruptConnectivity:
          ActionId: aws:network:disrupt-connectivity
          Parameters:
            scope: all
          Targets:
            Subnets: TargetSubnets
      Tags:
        experiment-type: resilience-validation
```

Safety guardrails: The `StopConditions` alarm should fire if error rate exceeds 10% or if a business-critical health check fails. This automatically halts the experiment before it causes real customer impact. Tag subnets you want FIS to target with `az-chaos-target: enabled` so the blast radius is always explicit and auditable.

Running the combined test:

1. Start a DLT load test and wait for steady-state metrics to stabilize (typically 2-3 minutes)

2. Trigger the FIS experiment via the console, CLI (`aws fis start-experiment`), or your CI/CD pipeline
3. Monitor the DLT real-time dashboard alongside CloudWatch Application Signals service map
4. After FIS completes, observe the recovery curve: how long until metrics return to pre-fault levels?

Interpreting Chaos Results

The most valuable output of a chaos experiment is not whether the test "passed" but what the degradation curve looked like during the fault:

Observation	Implication	Action
p99 spiked briefly then recovered within 30 seconds	Health checks and routing are working correctly	Document as expected behavior, add to runbook
Error rate increased but never exceeded SLO	System degrades gracefully under partial failure	Increase blast radius in next experiment
Metrics never recovered until fault was manually stopped	Auto-healing is not working as expected	Fix health checks, scaling triggers, or circuit breakers before the next production deployment
No observable impact at all	Either the fault did not reach the right component , or redundancy is working perfectly	Verify fault reached target, then celebrate if redundancy confirmed

Put It Into Practice

- Deploy AWS Fault Injection Service in your test environment
- Create a simple FIS experiment template (start with: terminate one ECS task)
- Run a DLT load test at moderate load (50% expected peak)
- While the load test is running, trigger the FIS experiment

- Compare the "with fault" results against your "no fault" baseline
- Document the system's actual behavior vs. your hypothesis

Go Deeper

Chaos Testing with AWS Fault Injection Service and AWS CodePipeline (Architecture Blog)
Simulating partial failures with AWS Fault Injection Service (Management & Governance Blog)

Scaling to Production-Grade Testing

Moving from "my first load test" to "production-grade performance engineering" requires addressing multi-region distribution, cost optimization, organizational governance, and pre-event readiness.

Multi-Region Load Distribution

For applications serving global users, single-region testing produces incomplete results. CDN caching, geographic routing, and regional database replicas behave differently under distributed load.

DLT supports multi-region testing through regional stack deployments. The main stack orchestrates tests across all registered regions simultaneously, aggregating results in a single dashboard.

Consideration	Guidance
Region selection	Match your actual user distribution (use CloudFront or ALB access logs)
Load distribution	Weight traffic by region proportional to real user patterns
Network effects	Each region adds authentic network latency to results
Cost optimization	Deploy regional stacks only when running multi-region tests

Cost Management for Load Testing

Load testing consumes compute resources proportional to test scale and duration. Plan costs carefully:

Cost Driver	Mitigation
Fargate task-hours	Right-size task count using the 200 VU guideline; avoid over-provisioning
NAT Gateway data transfer	Use VPC endpoints for AWS service targets when possible
CloudWatch data ingestion	Disable live monitoring for routine automated tests
Test environment infrastructure	Tear down environments after testing (infrastructure as code)
Long-running soak tests	Schedule during off-peak hours for potential Spot pricing on target infra

AWS Countdown Premium for Critical Events

When preparing for high-stakes events (product launches, seasonal peaks, migration go-lives), AWS Countdown Premium Short Term Engagements provide expert guidance alongside your DLT implementation.

What Countdown Premium engineers provide:

- Performance testing strategy and methodology guidance
- JMeter, K6, and Locust script structure review
- Results analysis and performance optimization recommendations
- Resource utilization analysis and best practices alignment
- End-to-end guidance from setup through results interpretation

Your team maintains: Test script development, execution, and operations. This is a "do-it-yourself with expert guidance" model that builds internal capability.

When to engage Countdown Premium:

If your team lacks in-house performance testing expertise, or if the stakes are high enough that you want expert guidance, consider engaging AWS Countdown Premium when:

- Preparing for a launch event with >10x normal traffic expected
- First large-scale performance test for a new workload
- After discovering production performance issues that need systematic investigation
- Before peak retail seasons, gaming launches, or media events

Ready to validate at production scale? For critical launches and peak events, AWS Countdown Premium Short Term Engagements provide expert AWS engineers to guide your performance testing strategy. Sign up directly through AWS Countdown and select "Use Case Implementation."

Governance and Organizational Patterns

As performance testing matures across your organization:

- **Establish a performance testing Center of Excellence (CoE)** that maintains shared DLT infrastructure, script templates, and baseline standards
- **Define ownership:** Each service team owns their performance SLOs and scripts; the platform team owns shared infrastructure
- **Create a test catalog:** Document standard test scenarios (smoke, load, stress, soak) with predefined configurations
- **Automate reporting:** Publish weekly performance trend reports from DLT results to stakeholders
- **Track improvement:** Use DLT baseline comparison to demonstrate performance improvements quarter over quarter

Pre-Event Testing Playbook

For high-stakes events (product launches, Black Friday, gaming tournaments, marketing campaigns), follow this structured approach:

Timeline	Action	Outcome
T-8 weeks	Define event traffic model (expected peak, duration, geographic distribution)	Clear test parameters
T-6 weeks	Deploy multi-region DLT stacks matching user distribut ion	Test infrastructure ready
T-4 weeks	Run first full-scale test, identify bottlenecks	Baseline and gap list
T-3 weeks	Address bottlenecks (scaling, caching, pre-warming)	Architecture hardened
T-2 weeks	Re-test at 150% expected peak, run chaos experiments	Confidence in resilience
T-1 week	Final validation test, pre-warm caches and connection pools	Go/no-go decision
T-1 day	Smoke test infrastructure, verify monitoring dashboards	Operational readiness

Service Quota Planning for Large Tests

Large-scale load tests often fail due to AWS service quotas, not application bottlenecks. Before running tests above 10,000 VUs, verify these quotas:

- **ECS Fargate tasks per cluster** (default: varies by region, request increase 2 weeks in advance)
- **NAT Gateway bandwidth** (default: 100 Gbps per NAT, but burst behavior matters)
- **Elastic IP addresses** (if using multiple NAT Gateways for source IP distribution)
- **Target application quotas** (API Gateway throttling, Lambda concurrency, RDS max connections)
- **CloudWatch PutMetricData TPS** (can throttle at very high test volumes)

Request quota increases at least two weeks before your planned test date. AWS Trusted Advisor provides a quota usage dashboard that shows current consumption against limits.

Put It Into Practice

- Create a "Pre-Event Runbook" document in your team wiki using the timeline table above as a template
- Run `aws service-quotas list-service-quotas --service-code ecs` to check your current Fargate task limits
- Deploy DLT regional stacks in at least 2 regions matching your top user geographies
- Run a multi-region test distributing load proportionally (e.g., 60% us-east-1, 25% eu-west-1, 15% ap-southeast-1) and compare regional latency differences
- Document your organization's performance testing governance model: who owns infrastructure, who owns scripts, who reviews results

```
# Check ECS Fargate task quota in your region
aws service-quotas get-service-quota \
  --service-code ecs \
  --quota-code L-4FC25E62 \
  --region us-east-1

# Request a quota increase for large-scale testing
aws service-quotas request-service-quota-increase \
  --service-code ecs \
  --quota-code L-4FC25E62 \
  --desired-value 500
```

Investigation at Scale with AWS DevOps Agent

Moving from "my first load test" to "production-grade performance engineering" requires not just larger tests but also mature analysis practices. Manual investigation of test results becomes impractical as test frequency and complexity increase. DLT's native AWS DevOps Agent integration transforms post-test analysis from a labor-intensive, manual process into an automated, repeatable workflow.

Scaling investigation across environments:

Register multiple Agent Spaces for different purposes. A dedicated Agent Space per environment (development, staging, production) allows each team to investigate independently without cross-contamination of findings:

Agent Space	Purpose	When to Investigate
Dev Agent Space	Early regression detection	After every PR merge smoke test
Staging Agent Space	Pre-release validation	After full load tests on staging
Production Agent Space	Production incident correlati on	After canary tests or observed degradation

Building a historical investigation record:

Every DevOps Agent investigation is stored in the DLT Investigations tab for the lifetime of the test run record. Over time, this builds a searchable knowledge base of performance findings. Use this history to:

- **Track regression patterns.** If the same component appears as a bottleneck across multiple investigations, it signals a systemic architectural issue rather than a one-time configuration problem
- **Validate remediation effectiveness.** After implementing a fix, compare the investigation findings from the next test run against the original. If the root cause no longer appears, the fix was effective
- **Inform capacity planning.** Historical investigations reveal which components hit limits first under increasing load, helping you prioritize scaling investments

Integrating DevOps Agent into CI/CD pipelines:

Combine the DLT CLI with DevOps Agent for fully automated performance gate workflows:

1. **CI/CD pipeline triggers a load test** via the DLT CLI after deployment to staging
2. **DLT compares results against baseline** and detects if any SLO is violated
3. **If a regression is detected**, the pipeline automatically triggers a DevOps Agent investigation

4. **The investigation findings** are captured and attached to the deployment record
5. **The pipeline fails the deployment** with actionable root-cause information, not just a "latency exceeded threshold" error

This pattern transforms performance gates from binary pass/fail signals into diagnostic workflows that tell engineers *why* performance degraded and *what to fix*, eliminating the manual triage step that typically adds hours or days to incident resolution.

Providing context for better investigations:

When triggering an investigation (either manually or via CLI), provide additional context to improve the quality of findings:

- Recent infrastructure changes (new deployment, scaling policy change, instance type migration)
- Expected performance baseline (the target SLOs for this test)
- Known environmental differences (reduced replica count in staging, stubbed external services)

Context helps the agent distinguish between expected behavior and genuine anomalies, reducing false positives and focusing findings on actionable issues.

Go Deeper

AWS DevOps Agent Integration (DLT Documentation)
AWS Service Quotas: Managing your quotas (Service Quotas Documentation)
AWS Countdown Premium for DLT (DLT Documentation)
Imagine Learning Case Study: Performance Testing at Scale (Customer Case Study)

What Comes Next

Performance testing is not a destination. It is a continuous discipline that evolves with your application. After establishing the practices in this journey:

Expand coverage: Add performance tests for new services as they launch. Every service that has an SLO should have automated performance validation.

Deepen analysis: Move from "did it pass?" to "why did it behave this way?" Invest in distributed tracing correlation and automated anomaly detection.

Shift left further: Explore component-level benchmarking and contract testing for latency budgets between services.

Connect to business outcomes: Link performance metrics to business KPIs. Show stakeholders that a 100ms latency improvement translates to measurable conversion increases.

Additional Resources

Related AWS Services

AWS Auto Scaling — Scale compute resources automatically

AWS CloudFormation — Model and set up AWS resources with infrastructure as code

AWS CodeBuild — Build and test code

AWS CodePipeline — Continuous delivery service

AWS Distro for OpenTelemetry — Secure, production-ready open source distribution of OpenTelemetry

AWS Fargate — Serverless compute for containers

AWS Fault Injection Service — Improve resiliency with controlled fault injection experiments

AWS IAM — Securely manage access to AWS services and resources

AWS Lambda — Run code without thinking about servers

AWS Step Functions — Visual workflows for distributed applications

AWS Trusted Advisor — Optimize AWS environment with best practice recommendations

AWS Well-Architected Tool — Review and improve workload architecture

Amazon API Gateway — Create, publish, and manage APIs

Amazon CloudFront — Fast content delivery network

Amazon CloudWatch — Monitoring and observability service

Amazon Cognito — Identity management for apps

Amazon DynamoDB — Fast, flexible NoSQL database service

Amazon EC2 — Scalable compute capacity in the cloud

Amazon ECR — Fully managed container registry

Amazon ECS — Fully managed container orchestration service

Amazon RDS — Managed relational database service

Amazon S3 — Object storage built to retrieve any amount of data from anywhere

Amazon VPC — Isolated cloud resources

Distributed Load Testing on AWS — AWS Solution for scalable load testing

Elastic Load Balancing — Distribute incoming application traffic

Other Related Assets

A phased approach for performance engineering in the AWS Cloud — AWS documentation

PERF05-BP04: Load test your workload — AWS documentation

Improve application reliability with effective SLOs — AWS blog post with implementation guidance

Load testing applications — AWS documentation

Validate system reliability with performance testing (QA.NT.2) — AWS documentation

DLT Features: Test Framework Support — AWS documentation

Performance Testing on AWS (re:Invent 2025, CMP351) — Video walkthrough

Distributed Load Testing on AWS: Implementation Guide — AWS documentation

Ensure Optimal Application Performance with DLT — AWS blog post with implementation guidance

Load testing applications: Design realistic scenarios — AWS documentation

Amazon CloudWatch Service Level Objectives — AWS documentation

Monitor application health using SLOs with CloudWatch Application Signals — AWS blog post with implementation guidance

DLT CLI Documentation — AWS documentation

Performance Efficiency Pillar: Process and Culture — AWS documentation

Chaos Testing with AWS Fault Injection Service and AWS CodePipeline — AWS blog post with implementation guidance

Simulating partial failures with AWS Fault Injection Service — AWS blog post with implementation guidance

AWS Service Quotas: Managing your quotas — AWS documentation

AWS Countdown Premium for DLT — AWS documentation

Imagine Learning Case Study: Performance Testing at Scale — AWS Solutions implementation

Feedback

Share your feedback

Help us improve this journey by sharing your experience: Take the survey