Gatling

2026

DATASHEET

6 load testing patterns and what they reveal

A practical guide to choosing the right traffic curve in Gatling

• TYPES OF LOAD TESTING

Most teams run load tests. Few run the right traffic profile.

This datasheet explains 6 common load testing patterns, what each reveals, and how to reproduce them in Gatling. It focuses on open workload models (user arrival rate). For closed workload guidance and examples, reach out to us.

To make selection easier, the test patterns are grouped into three sections:

Validation → Scaling → Resilience

COMPONENT	WHAT IT DOES
injectOpen(...)	Declares an open workload injection profile. Users are injected based on arrival rate (users per second), not a fixed number of concurrent users.
atOnceUsers(users)	Injects a fixed number of users immediately at time zero. Used for smoke tests and fast validation.
constantUsersPerSec(rate).during(time)	Inject users gradually over time
rampUsersPerSec(start).to(end).during(time)	Linearly increases arrival rate from start to end users per second over time. Used for ramp, breakpoint, or the ramp phase of ramp–hold tests.
stressPeakUsers(rate).during(time)	Quickly reaches a high arrival rate and sustains it for time. Used to push the system into overload conditions to expose failure behavior.
incrementUsersPerSec(step)	Defines a stepwise increase in arrival rate. Each step increases the current rate by step users per second. Used in capacity tests (stair-step curve).
.startingFrom(rate)	Sets the baseline arrival rate before applying increments. In your capacity test, this is the first plateau (example: 10 users per second).
.times(n)	Repeats the increment pattern n times. In a capacity test, this defines the number of steps (how many plateaus above baseline).
.eachLevelLasting(time)	For step-based profiles, holds each arrival-rate plateau for a fixed duration. This stabilizes the system at each step and makes measurements comparable.
.separatedByRampsLasting(time)	Inserts ramp transitions between steps (instead of abrupt jumps). This avoids shock effects and produces smoother scaling behavior.
.during(time)	Specifies how long an injection phase lasts. Used for constant load, stress plateaus, and linear ramps.
vu (variable)	In next code snippets, vu represents the arrival rate or increment size (users per second). It is not the number of concurrent users.
duration / ramp_duration	These variables define the time window for holding a plateau (duration) or the time window for ramping (ramp_duration). They shape the curve but do not change the target arrival rate.

Smoke test

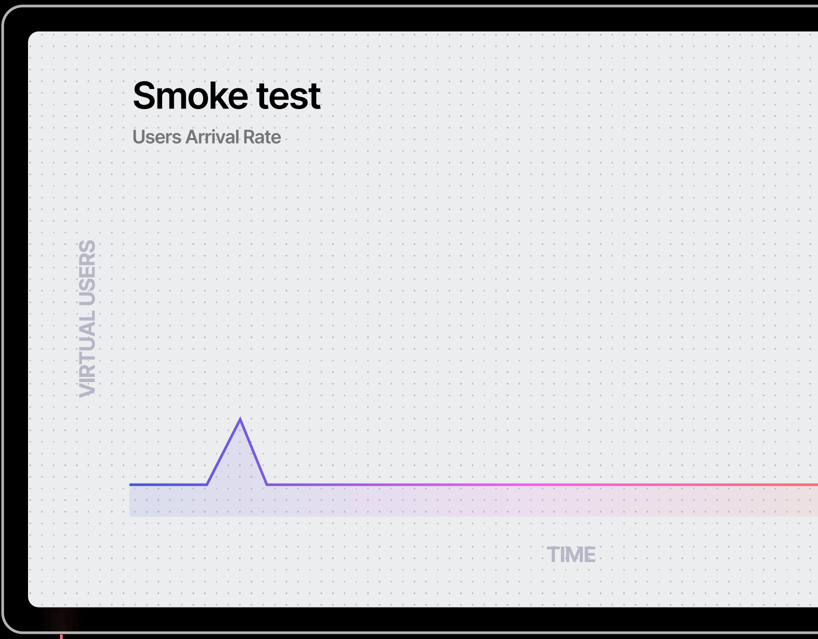
Minimal traffic to verify scenario correctness and environment readiness before scaling.

When to use it

- Validate test setup end-to-end (authentication, routing, data, dependencies)
- Sanity-check pre-merge or pre-pipeline before heavier tests
- Confirm telemetry wiring (metrics visible, dashboards populated, traces available)

What it reveals

- Immediate functional failures (unauthorized responses, server errors, timeouts)
- Environment misconfiguration (domain name resolution, credentials, routing rules)
- Missing instrumentation or broken telemetry before load



One immediate user arrival.
Used for fast validation, not for performance measurement.

Code example to reproduce this pattern

```
scn.injectOpen(atOnceUsers(1));
```

Ramp-hold test

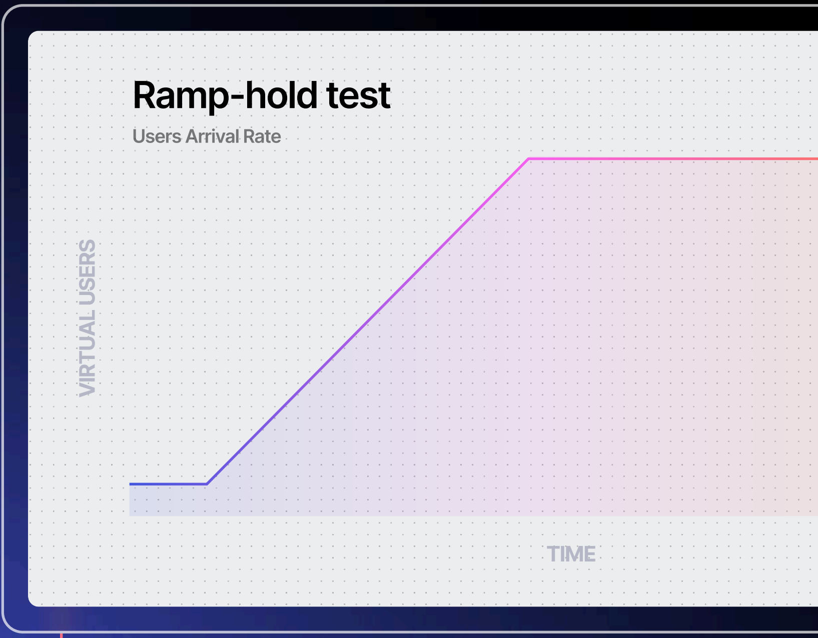
Ramp to a target arrival rate, then hold steady to validate performance and service level compliance under stable peak conditions.

When to use it

- Release validation or continuous integration performance gating
- Confirm service level objectives at expected peak traffic
- Compare baseline vs new release under identical traffic shape

What it reveals

- Ramp behavior vs steady-state behavior (warm-up, scaling stabilization)
- Whether performance stabilizes or continues degrading under constant load
- Regression signals compared to previous baseline runs



Ramp from 0 → vu users per second over ramp_duration, then hold at vu users per second for duration.

Code example to reproduce this pattern

```
scn.injectOpen(
rampUsersPerSec(0).to(vu).during({amount: ramp_duration, unit: "minutes"}),
constantUsersPerSec(vu).during({ amount:duration, unit: "minutes"}));
```

Capacity test

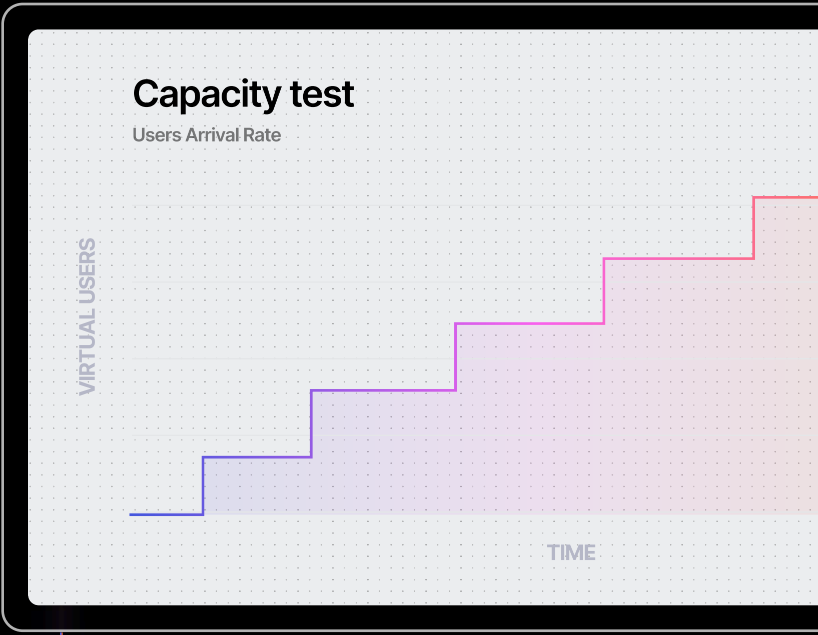
A stepwise traffic increase with stabilized plateaus to measure sustainable throughput and identify scaling limits.

When to use it

- Validate expected peak traffic under controlled increments
- Measure scaling behavior (automatic scaling, resource saturation, throughput ceilings)
- Establish a repeatable capacity baseline to compare releases or infrastructure changes

What it reveals

- The traffic level where latency percentiles (p95 and p99) start diverging
- Throughput ceilings and early saturation thresholds (CPU, queues, connection pools, load balancers and gateways)
- Non-linear scaling zones (automatic scaling delays, contention, backlog accumulation)



Progressively increase the arrival rate in steps, holding each level long enough for the system to stabilize and reveal sustainable throughput limits.

Code example to reproduce this pattern

```
scn.injectOpen(incrementUsersPerSec(vu).times(4).eachLevelLasting({ amount:
duration, unit: "minutes" }).separatedByRampsLasting(4).startingFrom(10));
```

Breakpoint test

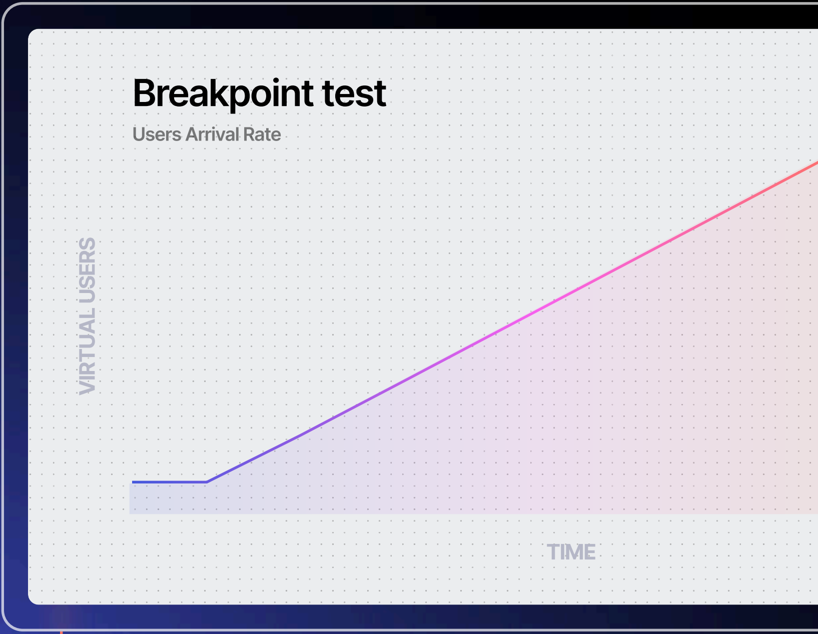
A continuous ramp-up to determine the traffic threshold where service level objectives break and the system loses stability.

When to use it

- Find the maximum sustainable arrival rate (true capacity limit)
- Quantify headroom and define a safe operating range
- Validate infrastructure ceilings such as load balancers, gateways, and database connection limits

What it reveals

- The load level where service level objectives fail (p99 spikes, errors increase, throughput collapses)
- Which layer saturates first (edge, gateway, service, or dependency)
- Whether automatic scaling can delay or prevent failure



Linear ramp from 0 → vu users per second over duration. Often paired with automatic stop criteria (error rate thresholds, percentile thresholds, time limits).

Code example to reproduce this pattern

```
scn.injectOpen(
rampUsersPerSec(0).to(vu).during({amount: duration, unit: "minutes"}));
```

Stress test

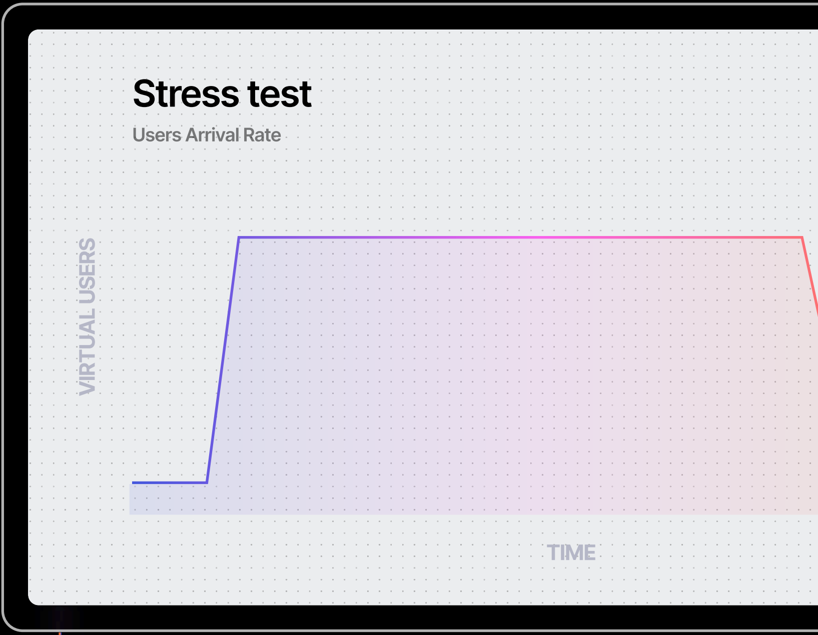
Sustained high arrival rate to push the system beyond normal operating conditions and expose failure modes under overload.

When to use it

- Validate resilience beyond expected peak traffic
- Observe failure behavior: error cliffs, timeouts, and recovery dynamics
- Test protective mechanisms such as rate limiting, backpressure, and circuit breakers

What it reveals

- Tail latency explosions (p99 and p99.9) caused by contention and queueing
- Error cliffs and timeout cascades caused by retries and overload
- Stability limits under sustained saturation (CPU, database connections, network throughput)



Sustained peak traffic at vu users per second for duration. Designed to force saturation and observe failure behavior.

Code example to reproduce this pattern

```
scn.injectOpen(
constantUsersPerSec(vu).during({amount: duration, unit: "minutes"}));
```

Soak test

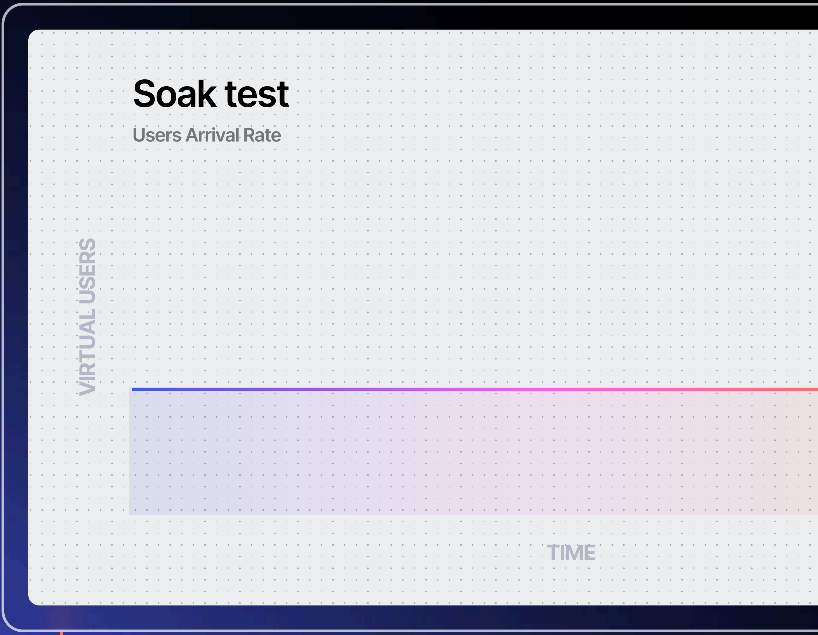
A constant arrival rate sustained over time to validate stability and detect slow degradation.

When to use it

- Detect memory leaks or slow resource exhaustion
- Validate long-run stability under steady business-as-usual traffic
- Identify drift caused by cache eviction, background jobs, or state accumulation

What it reveals

- Latency drift and tail degradation over time
- Slow-growing error rates and saturation trends
- Resource growth patterns leading to collapse (memory pressure, connection pool exhaustion)



Constant traffic at vu users per second for the full duration.

Code example to reproduce this pattern

```
scn.injectOpen(
constantUsersPerSec(vu).during({amount: duration, unit: "minutes"}));
```

Quick selection guide

Use this table as a shortcut: each test pattern answers a different engineering question and reveals a different class of failure modes, scaling limits, or stability risks.

TEST TYPE	MAIN QUESTION	COMMON DURATION	WHEN TO PERFORM	KEY OUTPUTS
Smoke	Does the scenario run end-to-end?	15–30 min	Before any serious performance testing	setup errors, broken flows, missing dependencies
Ramp–Hold	Do service level objectives hold at peak?	30 min	Before production releases / after major updates	p95 / p99, error rate, steady-state stability
Capacity	What is the maximum sustainable traffic?	Several hours (until failure)	Before scaling decisions / infrastructure changes	throughput ceiling, saturation thresholds, scaling behavior
Breakpoint	At which traffic level do we break?	1 hour	Before a known high-traffic event / product launch	failure threshold, first saturated layer, safe operating range
Stress	How does the system behave under overload?	2–3 hours	Before a known high-traffic event / product launch	error cliffs, timeout cascades, recovery behavior
Soak	Does performance degrade over time?	24–72 hours	Before production / after major updates	drift, leaks, long-run degradation patterns

Built to test any use case or protocol, at scale

Gatling Enterprise Edition is a developer-first load testing platform for modern, high-scale systems. It supports APIs, microservices, real-time protocols, and legacy tech across HTTP, WebSockets, gRPC, and more.

Create tests with code, low-code, or no-code options. Integrate seamlessly into CI/CD pipelines and APM tools to ensure resilience at any scale.

Web applications

Simulate user interactions and ensure fast, reliable front-end performance under load.

Public and private APIs

Validate your API performance, latency, and error handling across high concurrency.

Cloud-based infrastructures

Evaluate performance and resiliency of cloud-native apps, especially after migrations from on-premise to cloud environments.

SQL databases

Measure query response times and throughput under real-world usage scenarios.

Microservices architectures

Test internal service-to-service communication and isolate performance bottlenecks.

IoT systems and protocols

Emulate device fleets and message flows using MQTT, AMQP, and other IoT protocols.

Mobile applications

Reproduce mobile usage patterns and variable network conditions for realistic testing.

LLM and AI-powered APIs

Evaluate AI inference latency and ensure consistent performance for high-volume requests.

Gatling

Gatling is the leading solution for modern load testing, enabling developers and organizations to deliver fast, reliable applications at scale.

With its powerful open-source and enterprise platforms, Gatling empowers teams to test APIs, microservices, and web apps in real-world conditions.

Trusted by thousands of companies worldwide, Gatling is the performance backbone for development, QA, and DevOps teams building the next generation of software.

Whether you're scaling APIs, migrating to the cloud, or handling flash traffic spikes, Gatling helps you deliver fast, reliable performance.

Ready to evaluate Enterprise Edition?

Whether you're scaling APIs, migrating to the cloud, or handling flash traffic spikes, Gatling helps you deliver fast, reliable performance.

Talk to an expert ›