Engineer

Load test

a test that measures a system under simulated traffic

What it is

A load test sends a planned amount of simulated traffic at a system and measures how the system behaves while it carries it: how long responses take, how many requests fail, how much throughput it sustains and what the servers spend doing so. The traffic comes from a script that plays many users at once; the k6 tool calls each of them a virtual user. Load testing is the activity, and a load test is one run of it, with a shape, a duration and criteria for passing.

The term is used both broadly and narrowly. The k6 documentation, published by Grafana Labs, sorts tests by the load they apply. A smoke test runs a handful of virtual users to prove the script works. An average-load test simulates a typical day in production. A stress test goes above that average, a soak test holds average load for hours, a spike test applies a sudden and very high burst for a few minutes, and a breakpoint test keeps raising the load until something gives. Because “load test” can mean any test that simulates traffic, k6 calls the narrow kind an average-load test and notes that some teams say day-in-life test or volume test. Its guide is candid about the vocabulary: “no consensus even exists about the names of these test types”. The categories are relative as well: the load that stresses one application is an ordinary day for another.

A load test is not a functional test. It does not ask whether the checkout adds up the basket correctly; it asks whether the checkout still answers, and how quickly, while a realistic crowd is using it. The two need each other: a script that sends broken requests at high speed measures only how fast the system says no.

Use the concept in a decision

Put “What is a load test?” to work in a real decision. We can help relate the concept to the system you are building or buying. One team taking an idea from a sketch to something your customers can log into, instead of handing it between three suppliers.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Led a GIS map application to 10x revenue growth — 10x revenue growth, led from the map product

Led a GIS map application to 10x revenue growth Get in touch →

Why it matters

Every system has a level of traffic at which it stops keeping up, and the only question is who finds it first. Locust, an open-source load testing tool whose tests are plain Python, shows the shape in its quickstart. As the tool adds users, there comes a point, around 20 users in its example, where response times climb so fast that the number of requests per second stops rising even though more users keep arriving. The documentation calls the target overloaded, or saturated. That knee is the real capacity of the system as built, whatever the architecture diagram promises, and it is far cheaper to find in a rehearsal than in front of customers.

The ramp towards the knee is informative on its own. The k6 guide to average-load tests says the first result appears while load is still rising: whether response time degrades as traffic grows, and whether the system fails before it even reaches full load. The plateau then shows whether performance and resource use stay stable, since some systems turn erratic during exactly that stretch.

Load tests also turn performance from an opinion into a gate. k6 lets a script declare thresholds, pass or fail criteria on its metrics, such as fewer than 1% of requests failing or 95% of responses arriving within 200 ms. The k6 documentation notes that testers often use thresholds to write down their service level objectives. When every threshold holds, k6 exits with code 0, and when any fails it exits with a non-zero code, so a pipeline can refuse a release that made the service slower, with no one reading a graph.

What it looks like in practice

Take a shop preparing for a sale. Most of the traffic during a sale is people who will not get the item, so the answer that has to hold up is “sold out”, and that is the path to rehearse first. The test below is written in k6. It starts requests at a set rate rather than with a set number of users, ramps to 2,000 requests a second over three minutes, holds that for twenty and ramps down again. The ramp follows k6’s guidance for average-load tests: the ramp-up takes 5 to 15% of the test, the plateau lasts at least five times as long as the ramp-up, and the

ramp-down about as long as the ramp-up. A spike test, which k6 suggests for ticket sales, product launches and seasonal sales, would follow with almost no ramp at all; this one first proves that the sold-out answer holds at a steady rate.

```
import http from 'k6/http';

export const options = {
  scenarios: {
    sold_out: {
      executor: 'ramping-arrival-rate',
      startRate: 0,
      timeUnit: '1s',
      preAllocatedVUs: 400,
      stages: [
        { target: 2000, duration: '3m' },
        { target: 2000, duration: '20m' },
        { target: 0, duration: '3m' },
      ],
    },
  },
  thresholds: {
    http_req_failed: ['rate<0.01'],
    http_req_duration: ['p(95)<200'],
    dropped_iterations: ['count<1'],
  },
};

export default function () {
  http.get('https://shop.example/sale/items/4711');
}
```

Run it with `k6 run sold-out.js`. The number of virtual users is an estimate k6's documentation gives a formula for: the median iteration time multiplied by the rate, plus a margin. At 2,000 requests a second and about a tenth of a second each, 200 virtual users are busy at once, and 400 leaves room. If responses slow down, each iteration lasts longer, the pool runs dry and k6 records every iteration left without a free virtual user in `dropped_iterations`. That is why the third threshold is there: a dropped iteration means the test did not send the load it claims, and k6 notes that the system may itself be slowing down when the generator drops iterations. The other two thresholds fail the run if 1% or more of requests fail or the 95th percentile is 200 ms or higher.

The scenario comes from this site's note on selling the last ticket exactly once. With a million buyers and ten thousand units, it points out, 990,000 requests have to fail, so the rejection has to be the cheapest answer the system gives. Its

section on proving the design includes load-testing the rejections: the sold-out path has to stay fast when it is 99% of the traffic, and the edge of the network has to absorb it. A test that only rehearses a successful purchase measures the 1% that matters least to capacity.

Where it goes wrong

The test waits for the system it is measuring. In what k6 calls the closed model, each virtual user starts its next iteration only when the last one has finished. When the server slows down, the users slow with it, the arrival rate falls and the system gets a gentler test than the one planned, an effect k6 says some literature calls coordinated omission. Real visitors do not wait for each other. Where the question is whether a system can handle a given rate of requests, use an open-model executor such as `constant-arrival-rate` or `ramping-arrival-rate`, which starts iterations on schedule whatever the response time.

The load generator is the bottleneck. If the machine running the test runs out of virtual users, CPU or memory, the results describe the generator, not the target. k6 advises against `maxVUs` in most cases, because adding virtual users while the test runs costs CPU and memory and can overload the generator and skew results. Size `preAllocatedVUs` beforehand with a few local trials, and watch `dropped_iterations`. Locust spreads load over several machines with one master process and many workers when one is not enough.

The first run is the biggest. k6's documentation reports that thousands of its users have run load tests that quickly crashed their applications or their staging environments. Start with a smoke test of a few virtual users to prove the script and the environment, then run an average-load test, and only after that a stress test; k6 calls that sequence essential. Problems that appear at low volume are much easier to find before the load is high.

The load goes up and down like a rollercoaster. A pattern that rises and falls several times wastes resources and makes it hard to tell which change caused which symptom. k6 recommends one direction at a time: ramp up, hold a plateau, ramp down. Keep the scenario, data and environment the same between runs as well, so that two results can be compared at all.

Nobody decided what passing means. A run that ends with a page of charts and no criteria is read differently by everyone who looks at it, usually generously. Write the thresholds before the run, from the objective the service is held to, and

let the exit code decide. For long runs, k6 can abort as soon as a threshold fails with `abortOnFail`, and `delayAbortEval` keeps it from stopping before enough samples have arrived to mean anything.

One test type is treated as the whole answer. An average-load test says little about a sale that arrives in seconds, and a spike test says little about a fault that only shows after hours of steady use. k6 is explicit that no single test type removes all risk, and that the risk profile decides the mix: a system at risk from long use needs soak tests first, one at risk from intensive use needs stress tests. A spike may also exercise processes that never run on an ordinary day, so the script has to include them.

Further reading

- Load test types — the k6 overview of smoke, average-load, stress, soak, spike and breakpoint tests, with the load and duration typical of each.
- Average-load testing — how to shape the ramp-up, plateau and ramp-down, and what to read from each.
- Thresholds — k6's pass and fail criteria, their syntax, and how a failed threshold sets the exit code.
- Open and closed models — why a test whose users wait for responses under-reports a slow system.
- Arrival-rate VU allocation — how many virtual users an open-model test needs, and what dropped iterations mean.
- What is Locust? — the Locust documentation's own description of a Python load testing tool and its distributed mode.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a load test?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/load-test/