# Load Testing the Auth App with k6 and Grafana Cloud

Follow-up to `GUIDE.md`. Same app, new question: does it hold up under traffic?

---

## 1. What is load testing?

Unit and integration tests answer "does it work?". Load tests answer "does it still work when many people use it at once?"

You simulate traffic and measure:

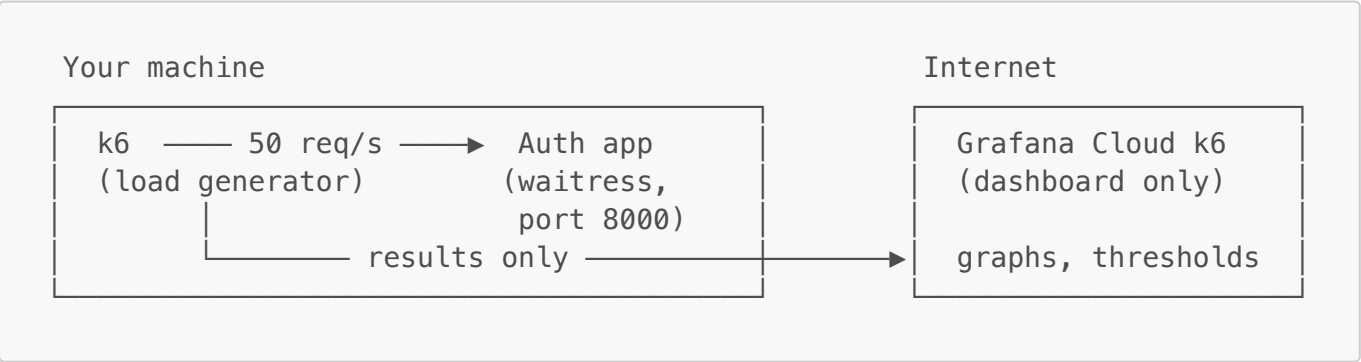
| Metric | Question it answers |
|---|---|
| Throughput (requests/s) | How much traffic are we serving? |
| Latency (p95 response time) | How long do most users wait? |
| Error rate | How many requests fail? |

> Why p95 and not the average? An average hides slow outliers. p95 = "95% of requests were faster than this". It describes what real users feel.

Common types: smoke (tiny, does the script work?), load (expected traffic), stress (beyond expected, find the breaking point), soak (long duration, find leaks). Today: a load test at 50 requests per second.

---

## 2. The setup




- k6: free, open-source load testing tool. Tests are written in JavaScript.
- Local execution: k6 runs *on your machine* and hits `127.0.0.1`. The app never needs to be online.
- Grafana Cloud: receives only the results, so you can explore them in a web dashboard.
- waitress: a production-grade Python web server. Flask's `python app.py` server is for development and would distort results.

Free tier: 500 virtual-user hours (VUh) per month, each test counts as at least 1 VUh, results kept 14 days. One run of today's test ≈ 1 VUh.

---

## 3. Create a Grafana Cloud account

1. Go to https://grafana.com → Create free account. Sign up with Google, GitHub or email. No credit card needed.
2. Choose a stack name, e.g. `yourname-loadtests`. Your stack URL becomes `https://yourname-loadtests.grafana.net`. Pick the region closest to you.
3. Once the stack opens, go to Testing & synthetics → Performance. The first visit initialises the k6 app (takes a few seconds).
4. Get your token: Testing & synthetics → Performance → Settings → Access → Personal token. Copy it and keep it private, like a password.

---

## 4. Install what you need locally

Python side (from the project folder, venv activated as in `GUIDE.md`):

```
pip install -r requirements.txt     # now includes waitress
```

k6 (pick your OS):

| OS | Command |
|---|---|
| macOS | `brew install k6` |
| Windows | `winget install k6 --source winget` |
| Linux | follow the apt/dnf steps at https://grafana.com/docs/k6/latest/set-up/install-k6/ |
| Any (Docker) | `docker pull grafana/k6` |

Check it works:

```
k6 version
```

Connect k6 to your Grafana account (once per machine):

```
k6 cloud login --token <YOUR_TOKEN> --stack <YOUR_STACK_SLUG>
```

`<YOUR_STACK_SLUG>` is the name from step 3 (e.g. `yourname-loadtests`). Run `k6 cloud login` with no flags to be prompted instead.

---

## 5. Write the load test

### 5.1 Start the app with waitress

In terminal 1:

```
waitress-serve --listen=127.0.0.1:8000 --threads=8 --call app:create_app
```

Open http://127.0.0.1:8000 to confirm it runs. Leave it running.

### 5.2 Plan the traffic

Real users don't all do the same thing. Our 50 req/s is split into two scenarios:

| Scenario | Request | Rate | Cost |
|---|---|---|---|
| `browse` | `GET /login` | 40/s | cheap (render a page) |
| `login` | `POST /login` | 10/s | expensive (password hashing) |

### 5.3 Build the script: `loadtests/load_test.js`

Build it in four steps. The complete file is provided in the project.

**Step A: imports and target**

```
import http from "k6/http";
import { check } from "k6";

const BASE_URL = __ENV.BASE_URL || "http://127.0.0.1:8000";
const USER = { username: "loadtester", password: "loadtest123" };
```

**Step B: the traffic shape**

```
export const options = {
  cloud: { name: "Auth app - 50 RPS" },
  scenarios: {
    browse: { executor: "constant-arrival-rate", exec: "browse",
              rate: 40, timeUnit: "1s", duration: "2m",
              preAllocatedVUs: 20, maxVUs: 100 },
    login:  { executor: "constant-arrival-rate", exec: "login",
              rate: 10, timeUnit: "1s", duration: "2m",
              preAllocatedVUs: 10, maxVUs: 100 },
  },
```

- `constant-arrival-rate` starts exactly `rate` iterations per `timeUnit`, however slow the server is. That's how we guarantee 50 req/s.
- A VU (virtual user) is a worker that runs iterations. If responses slow down, k6 adds VUs (up to `maxVUs`) to keep the rate.

**Step C: pass/fail criteria (thresholds)**

```
  thresholds: {
    http_req_failed: ["rate<0.01"],                    // < 1% errors
    http_req_duration: ["p(95)<500"],                  // 95% under 500 ms
    "http_req_duration{scenario:login}": ["p(95)<800"],
    http_reqs: ["rate>=45"],                           // we really hit ~50
rps
    checks: ["rate>0.99"],
    dropped_iterations: ["count<10"],
  },
};
```

Thresholds turn a load test into a real test: k6 exits with an error if any fails, just like a failing `pytest`.

**Step D: setup and the two scenario functions**

```
export function setup() {                    // runs once, before the load
  const res = http.post(`${BASE_URL}/register`, USER, { redirects: 0 });
  check(res, { "test user ready": (r) => r.status === 302 || r.status ===
400 });
}

export function browse() {
  const res = http.get(`${BASE_URL}/login`, { tags: { name: "GET /login" }
});
  check(res, { "login page 200": (r) => r.status === 200 });
}

export function login() {
  const res = http.post(`${BASE_URL}/login`, USER,
                        { redirects: 0, tags: { name: "POST /login" } });
  check(res, { "login succeeded": (r) => r.status === 302 });
}
```

- `check` is like `assert`, but doesn't stop the test. It records a pass rate.
- `redirects: 0` keeps each iteration to one request, so the count stays exactly 50/s.
- `400` in setup is fine: the user already exists from a previous run.

### 5.4 Run it

In terminal 2, first a quick local run (no cloud, nothing billed). Press `Ctrl+C` after ~10 s once requests flow:

```
k6 run loadtests/load_test.js
```

Then the real run, results streamed to Grafana Cloud:

```
k6 cloud run --local-execution loadtests/load_test.js
```

k6 prints a link to the live dashboard. Open it while the test runs.

---

## 6. Read the results in Grafana Cloud

Open the run: Testing & synthetics → Performance → Projects → (default project) → "Auth app - 50 RPS" → latest run (or use the link k6 printed).

### Performance Overview (top graph)

Requests per second, response time and VUs over time.

- Request rate is a flat line at ~50.
- Response time stays flat and low.
- VU count climbing = responses are slowing down, k6 needs more workers to keep up. Rule of thumb: *VUs needed ≈ rate × response time* (50 req/s × 0.02 s ≈ 1 busy VU).

### Cloud Insights

Automatic warnings (high error rate, slow responses, too many URLs...). Should be empty or green.

### Tabs at the bottom

| Tab | What to look at |
|---|---|
| Thresholds | All green? Each threshold from step C with its actual value. |
| Checks | Pass rate of each `check`. Should be 100%. |
| HTTP | One row per request name. Compare `GET /login` vs `POST /login`: count, p95, errors. |

### Questions to answer

1. Did we hold 50 req/s for the full 2 minutes?
2. Which request is slower, and by how much? Why? (Hint: look at `hash_password` in `auth.py`.)
3. Did any threshold fail? If so, which one, and what does it mean for users?
4. How many VUs did k6 actually need?

---

## 7. Updated code

| File | Change |
|---|---|
| `requirements.txt` | added `waitress>=3.0` |
| `loadtests/load_test.js` | new: the complete k6 script |

The app itself (`app.py`, `auth.py`, templates) is unchanged. `create_app()` is already a factory, so waitress can call it directly with `--call app:create_app`.

---

## 8. Exercises

1. Stress test. Raise `login` to 50/s, then 100/s. At what rate do thresholds start failing?
2. Tune the server. Repeat exercise 1 with `--threads=2` and `--threads=16`. What changes?
3. Security vs speed. In `auth.py`, the hash uses 100 000 iterations. Try 10 000 and rerun. Faster? Why is this a bad trade in production?
4. Ramp up. Replace `constant-arrival-rate` in `login` with `ramping-arrival-rate` going 0 → 50 → 0 req/s over 3 minutes. Where does the response time curve bend?
5. Wrong password traffic. Add a third scenario sending bad credentials at 5/s. Expect `401` in your check. Does `http_req_failed` complain? (Hint: 4xx counts as failed by default.)