Technology Experiment: Evaluating NGINX Load-Balancing Strategies

1) The Big Picture

Web servers often sit behind a load balancer. When many requests arrive, the balancer must decide which backend server should handle each request. Good choices mean faster pages and fewer slowdowns. Bad choices can make one server overloaded while others sit idle.

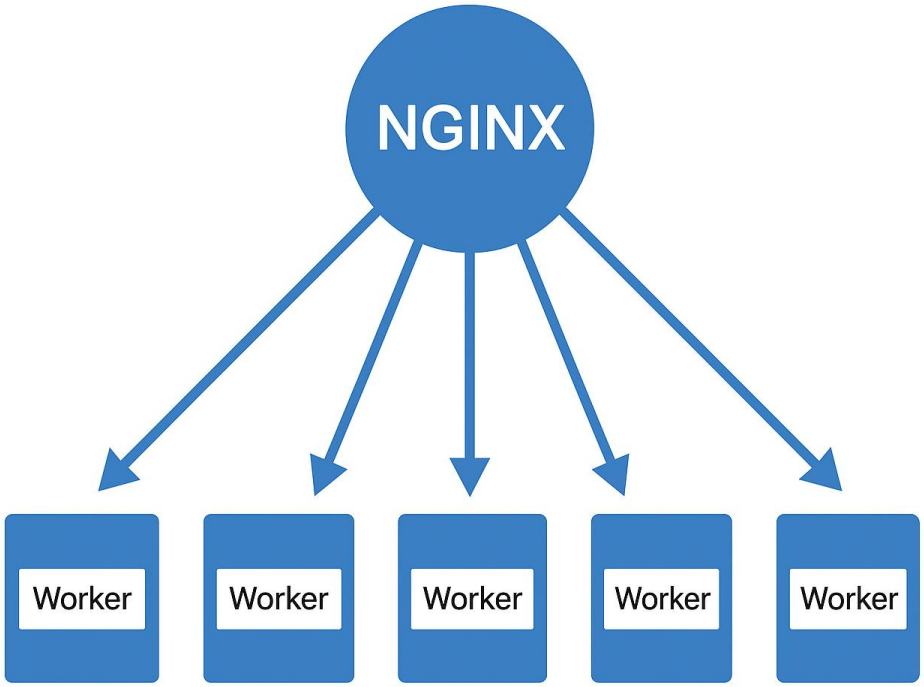


Figure 3. Nginx routing to five 1×1 workers

2) What We're Testing

We compared two simple ways Nginx can spread requests across servers:

• Random send each request to a server without checking how busy it is.

• Least-connections (LC): send each request to the server that currently has the fewest active requests.

Our question: when the system is under pressure (more work than servers can handle at once), does LC keep things smoother and faster?

3) Setup at a Glance

• 5 backend apps (Flask + Gunicorn).

• Each app is limited to 1 request at a time (Gunicorn --workers 1 --threads 1).

• Nginx sits in front with two upstreams we can switch between:

  – api_lc: least_conn; each server capped to 1 connection (max_conns=1).

  – api_rand: random selection.

• We used HTTP/1.0 (or Connection: close) for the test route to avoid keepalive stickiness. This forces Nginx to make a fresh decision on every request.

• A small Python sender container generates traffic in waves (batches) with controlled concurrency.

4) Two Strategies in Plain Terms

Imagine 5 checkout lines at a store:

• Random: you pick a line at random. Sometimes you pick the long line by accident.

• Least-connections: you look around and join the shortest line. Fewer bad picks, less waiting.

5) How We Measured

• We send bursts of requests (e.g., 8 total requests with 5 in flight at once).

• Each request does a CPU-heavy task (PBKDF2 hashing) so it takes a predictable time.

• We record how long batches take and compute p50/p95 (typical and tail latency) and overall throughput (RPS).

6) Key Results

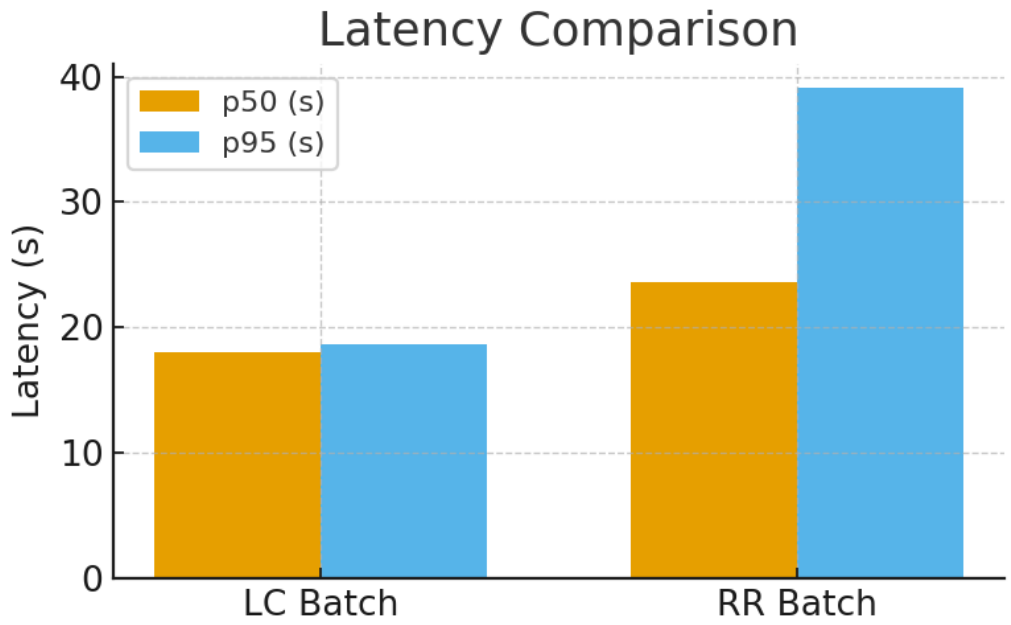
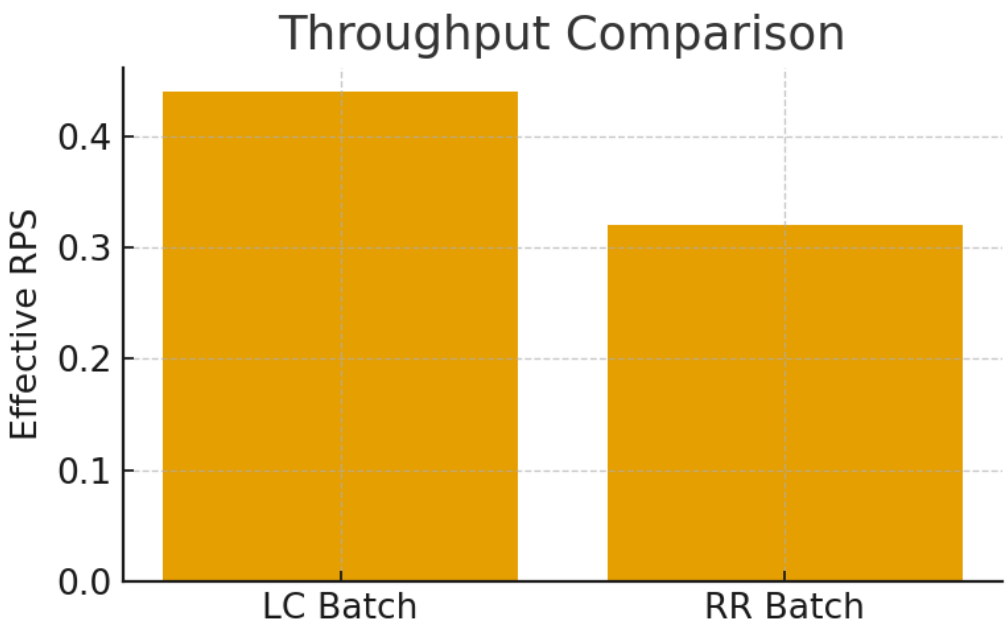


Figure 1. Latency comparison (p50 vs p95).



• LC batches ~18–19 seconds (very consistent).

• Random batches often ~23–39 seconds (some batches slow due to collisions).

• Throughput improved ~35–40% with LC under contention.

Translation: LC kept the work evenly spread. Random sometimes sent two long jobs to the same server, causing a queue and slowing the whole batch.

7) Why LC Wins Under Load (Intuition)

• Under pressure, some servers will be busy and some will be free.

• LC always tries to place the next request on a free (or less busy) server.

• Random ignores who is free, so it can pile onto a busy server by accident.

• With our 1-request-per-server limit, picking a busy server guarantees a wait. LC avoids that.

8) Common Gotchas

• Hidden extra capacity: environment variables like GUNICORN_WORKERS can silently raise concurrency. Check the actual Gunicorn command line.

• Keepalive reuse: HTTP/1.1 upstream keepalive can make Nginx reuse the same backend connection, reducing how often it makes new balancing decisions.

• Timeout mismatch: align timeouts (Nginx proxy_read_timeout ≥ Gunicorn --timeout ≥ client timeout).

9) Takeaway

When the system is at or slightly over capacity, least-connections keeps things balanced and fast. Random (or plain round-robin) can accidentally overload a single server, which hurts batch time and tail latency.