CreoBench: An Open-Loop, Distributed Load Tester for Modern Microservice Benchmarking

Yannik Lubas[1], Martin Straesser[2], Ivo Rohwer[1], Samuel Kounev[1]
[1]University of Würzburg, Germany
[2]Aschaffenburg University of Applied Sciences, Germany

Abstract

Open-loop, distributed load generation is a well-established pattern for realistic microservice benchmarking, yet implementations remain scarce. Existing tools force practitioners into unsatisfying trade-offs. Single-host open-loop generators bottleneck on one machine, while distributed closed-loop frameworks suffer from coordinated omission. Existing open-loop distributed load testers introduced useful architectural patterns, but their implementations have not kept pace with modern workload demands.

Building on these principles, we present CreoBench, an open-loop distributed load tester written in Rust. In this paper, we provide (i) our design decisions for distributed, open-loop load generation including precisely paced Poisson arrivals, (ii) an open-source implementation we hope the community can collectively improve, and (iii) an empirical evaluation demonstrating that our tool generates Poisson arrivals with sub-millisecond precision, and comparing its resource usage to the original Java implementation.

1 Introduction

Benchmarking modern microservices requires load generation that faithfully reproduces real-world request patterns. In production, arrival processes are typically open-loop, i.e., requests arrive independently of system state, and the system has no control over their timing [3]. Yet many popular load testing tools adopt closed-loop models, where the completion of one request triggers the next. Closed-loop generation suits systems with a fixed user count, but not services receiving requests from a large population of independent clients. Under closed-loop generation, a temporarily saturated system naturally throttles its own load, masking latency spikes and creating a false impression of stability; a phenomenon known as coordinated omission [4]. Practitioners relying on such tools overestimate real-world performance, because closed-loop generation conceals the saturation behavior that genuine open-loop traffic would expose.

Open-loop load generation avoids this pitfall by decoupling request arrival from request completion. The load intensity is prescribed externally (typically as a target arrival rate until a target time) and the generator attempts to maintain that rate regardless of system response. When the system under test saturates, excess load is not silently suppressed, yielding an accurate picture of behavior under stress. Single-machine generators suffice for small scales or powerful machines, but benchmarking modern microservices demands distributed generation to overcome single-host bottlenecks. This pattern is well understood in the research community [3], but distributed, open-loop implementations remain scarce.

The HTTP Load Generator (HLG) [2] introduced valuable architectural patterns for distributed open-loop generation, including a director–load generator separation, time-varying load intensity profiles, and scriptable user behavior using Lua. Building on these principles, we present a new, open-source implementation of an open-loop, distributed load tester written in Rust. Our design generates arrivals as a precisely paced Poisson process, and drops requests it cannot send on time rather than sending them late, so that capacity limits of the load generator are exposed instead of distorting the workload. It replaces thread-based concurrency with asynchronous I/O for higher scalability, leverages Rust to eliminate runtime dependencies and simplify deployment, and adopts configuration-as-code to meet modern reproducibility requirements without sacrificing flexibility.

We provide empirical evidence for the fidelity of the generated arrival process and the exposure of capacity limits, and compare the CPU and memory usage with the original HLG across increasing target load levels.

2 Related Work

The HLG [2] employs an architecture that separates a director node from multiple load generator workers, enabling distributed execution. The generator models time-varying arrival rates in the LIMBO [1] format, and user behavior is scripted in Lua. The tool also integrates with external power measurement daemons, enabling energy measurements during load testing.

However, some properties of the tool motivate our work. The implementation employs blocking I/O with OS-thread-based concurrency, which becomes a bottleneck when scaling to high connection counts on a single host. It requires a Java runtime, complicating

deployment in containerized or resource-constrained environments. Finally, the project is no longer actively maintained and lacks automated tests, which makes changes to the codebase risky. Our tool preserves the original design contributions while adopting modern implementation practices, including asynchronous I/O and automated testing.

Other open-loop tools exist, but are either constant throughput[1,2], require paid licenses[3,4] for distributed load testing, or also require a runtime[5,6,7].

3 Design

Our design adapts four core concepts from the HLG: (i) the open-loop workload model, (ii) the distributed coordination, (iii) the Lua scripting for user behavior, and (iv) time-varying load profiles. We modernize the execution layer by using a programming language that does not require a runtime and asynchronous instead of blocking I/O.

Our workload model schedules arrivals as a Poisson process according to a prescribed rate λ that may vary over time. Inter-arrival times of transactions follow an exponential distribution. We use a dedicated pacing thread to dispatch each transaction at its scheduled arrival time. To keep the dispatch timing precise, even for high workloads, the pacer sleeps until shortly before an arrival and busy-waits for the remaining millisecond. Each worker maintains a pool of virtual users (VU), where each new transaction executes the next request in the behavior loop of a random VU in the pool. A VU cannot have multiple in-flight requests because a VU's next request may depend on the response to its previous one, such as in a login flow that extracts an authentication token.

Pure open-loop generation (i.e., dispatching requests regardless of system state) requires both unbounded concurrency and stateless, independent requests. Neither holds in practice: hardware limits concurrency, and typical web workloads require per-session state and response-dependent sequencing. However, our tool never sends a request late. If a transaction cannot be sent within 10 ms of its scheduled arrival (no VU is free or the load generator itself falls behind), it is dropped and reported with its cause. This ensures that all requests reach the target following the prescribed arrival process, and any drop marks the run as invalid. Unlike closed-loop generators, which silently throttle load, our model explicitly exposes capacity limits.

The distributed coordination in our implementation follows the original director–worker pattern. A single coordinator process partitions the target request count across the N workers, synchronizes experiment start, and aggregates transaction results. Each worker independently draws its arrivals using its own monotonic clock relative to its start. Thus, clock offsets between machines do not affect the workload, and workers start apart only by the network latency of a single, concurrently-sent, 16-byte message, typically well below a millisecond within a data center. Since the superposition of independent Poisson processes also forms a Poisson process, the workers jointly generate the prescribed workload.

User behavior is scripted in Lua. Some workloads may require establishing session state (e.g., authentication tokens, or user-specific resources) before the load test begins. The original HLG executed all requests within the same behavior loop, offering no distinction between setup and measured traffic. Our tool introduces setup requests within the Lua behavior, which execute once per VU before the load test, initializing state without polluting the measured workload.

Our implementation is written in Rust, offering efficient asynchronous I/O enabling high per-worker request counts without requiring a language runtime. Our tool can be compiled into static binaries, facilitating deployment via container images or directly on bare metal. Every load test persists detailed artifacts, such as per-request outcomes and latencies, in addition to aggregated results. We also aim to maintain a thorough automated test suite, which currently covers over 400 unit and integration tests with a code coverage of 92%. Our tool is publicly available on GitHub[8] and we publish a CodeOcean replication package[9].

4 Evaluation

We evaluate how precisely our implementation generates the prescribed workload, and compare its resource efficiency against the original HLG [2] under identical workloads, focusing on the fidelity of the arrival process, the exposure of capacity limits, and CPU and memory usage as the target load increases.

4.1 Experimental Setup

We use a simple HTTP echo server that returns a response after a fixed delay of 90 ms, resulting in a mean response time of about 92 ms. The load generators and echo server run on separate but identical machines, each with an Intel Xeon E5-2650 v4 (12 cores, 24 threads) and 32 GB of memory, running Debian with Linux 6.1 and connected via 1 Gbit/s Ethernet. The load generator is pinned to 22 hardware threads, while the coordinator and monitoring run on the remaining two.

For a target rate λ and response time W, Little's law dictates that $a = \lambda \times W$ VUs are busy on average [3]. However, in a Poisson process, the num-

[1] https://github.com/giltene/wrk2
[2] https://github.com/tsenart/vegeta
[3] https://www.artillery.io/
[4] https://gatling.io/
[5] https://jmeter.apache.org/
[6] https://locust.io/
[7] https://k6.io/

[8] https://github.com/DescartesResearch/CreoBench
[9] https://doi.org/10.24433/CO.2085565.v1

ber of transactions in progress fluctuates around a with a standard deviation of $\sqrt{a}$, so a pool of a VUs would regularly run out. We therefore size the pool as $a + 7\sqrt{a}$, which practically avoids drops even in day-long runs. Our selected target rates are 500, 1,000, 5,000, 10,000, and 50,000, yielding 94, 160, 611, 1,133, and 5,075 VUs. We run all combinations of rates and VU counts, including undersized pools.

Each configuration runs a 10-second warmup at the target rate, followed by a 5-second pause, and a 60-second measurement period. Each configuration is repeated 5 times. Both tools execute an equivalent Lua script that performs a simple HTTP GET request to the echo server with a timeout of 1 s. We sample CPU and memory usage every second using `pidstat` and extract the achieved request rate, the count of successful transactions, and the count of dropped transactions from each tool's output. For our tool, we additionally assess the arrival process from the per-request scheduled and actual send times, using the coefficient of variation of inter-arrival times and the index of dispersion of arrival counts for windows of 1, 10, and 100 ms. For a Poisson process both should equal 1.

4.2 Results

With a sufficiently large VU pool, CreoBench achieves every target rate without failures. All requests are sent on time with 99% being sent within 142 μs of their scheduled arrival, and at most after 7.9 ms. Both the scheduled and actually sent arrivals match a Poisson process. The coefficient of variation of inter-arrival times deviates from 1 by at most 0.036, and the index of dispersion by at most 0.12 across all window sizes. With undersized pools, CreoBench drops transactions instead of silently throttling the load and correctly attributes all drops to missing VUs. The sending rate saturates within 0.3% of the pool's capacity predicted by Little's law. For instance, 94 VUs sustain 1,023 RPS compared to the predicted $94 \div 0.092\,\text{s} \approx 1{,}022$.

Figure 1 (top) compares the average CPU utilization. CreoBench's CPU usage includes its pacing thread, which busy-waits for precise dispatch, using between 30% of one core at 500 RPS and a full core from 5,000 RPS on, trading CPU time for scheduling fidelity. Hence, HLG uses less CPU at lower rates (22% vs. 44% at 500 RPS), while CreoBench uses 29% less at 10,000 RPS, as its per-request cost is lower.

Figure 1 (bottom) shows the average resident set size (RSS) memory usage. CreoBench uses 17 MB at 500 RPS and 438 MB at 50,000 RPS, while HLG uses 9 to 19 times more. HLG creates a separate HTTP client for each VU, resulting in at least 45,000 threads for 5,075 VUs. HLG fails to create threads at this pool size, so only CreoBench sustains 50,000 RPS.

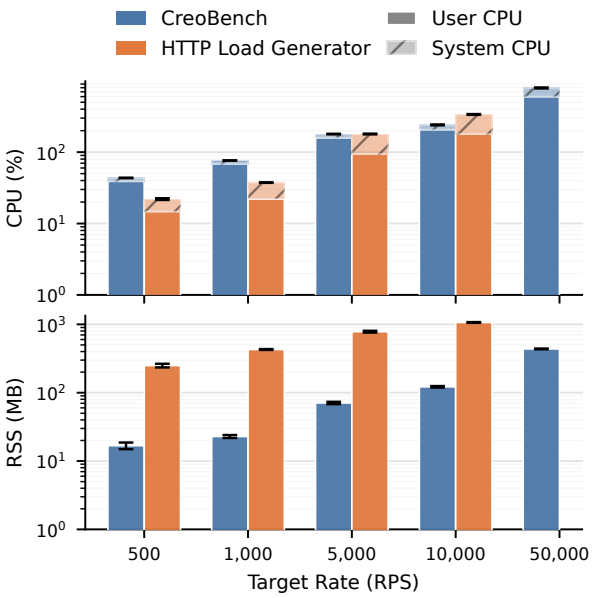

Figure 1: Average CPU utilization (top) and memory RSS (bottom) per configuration

5 Conclusion

In this paper, we presented our open-loop, distributed load tester CreoBench, which preserves the proven concepts of the HLG [2] and extends its design to meet modern benchmarking requirements. Our evaluation shows that CreoBench generates Poisson arrivals with sub-millisecond precision, and explicitly exposes undersized virtual user pools instead of silently distorting the workload. Compared to the HLG, CreoBench uses up to 19 times less memory and sustains 50,000 RPS, at the cost of up to one CPU core for precise pacing. In future work, we plan to evaluate CreoBench against systems with queuing behavior and across multiple machines. We publish our implementation as an open-source tool that we hope the community can use and improve in the future.

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 536362030.

References

[1] J. v. Kistowski, N. Herbst, and S. Kounev. "LIMBO: A Tool for Modeling Variable Load Intensities". In: *ICPE*. 2014.

[2] J. von Kistowski, M. Deffner, and S. Kounev. "Run-time Prediction of Power Consumption for Component Deployments". In: *ICAC*. 2018.

[3] S. Kounev, K.-D. Lange, and J. Von Kistowski. *Systems Benchmarking*. Vol. 1. Springer, 2020.

[4] G. Tene. *How Not to Measure Latency*. `https://www.infoq.com/presentations/latency-response-time/`. Last acc.: Aug. 21st, 2026.