Load Fiction: How Your Load Generator is Misleading You

Amit Das
University of Illinois
Urbana-Champaign, USA
Urbana, Illinois, USA
amitdas3@illinois.edu

Anant Goyal
University of Illinois
Urbana-Champaign, USA
Urbana, USA
anantg2@illinois.edu

Yug Mittal
University of Illinois
Urbana-Champaign, USA
Urbana, USA
yugm2@illinois.edu

Saurabh Jha
IBM Research
New York, USA
saurabh.jha@ibm.com

Chirag C Shetty
University of Illinois
Urbana-Champaign, USA
Urbana, USA
cshetty2@illinois.edu

Indranil Gupta
University of Illinois
Urbana-Champaign, USA
Urbana, USA
indy@illinois.edu

Abstract

Load Generators (LGs) are widely used to stress systems, measure performance, and identify bottlenecks. "Open-loop" LGs, used to schedule the sending of requests to the "system under test" (SUT) independent of the SUT's response time, have become standard. Our main finding is that many open-loop LGs "silently" fail to deliver the intended workload. We call these failures silent because aggregate metrics reported by LGs, such as send rate or total request count, can make a load test appear correct. This paper identifies various ways in which LGs unintentionally fail to remain open-loop, examines the impact of these failures, and introduces a framework for detecting and classifying them. Our proposed XLG Inspector is an observability layer that treats the SUT as a black box and diagnoses LG behavior using in-flight request counts and request-level delay signatures. We integrate XLG Inspector with the popular LG called Vegeta. Empirical evaluation with systematic fault injections demonstrates that XLG Inspector identifies bottlenecks that prevent intended workloads from reaching the SUT.

CCS Concepts

• **Computing methodologies** → **Simulation evaluation**.

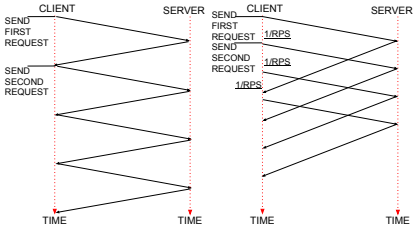
Keywords

Load Testing, Stress Testing, Performance Testing, Load Generators

ACM Reference Format:
Amit Das, Anant Goyal, Yug Mittal, Saurabh Jha, Chirag C Shetty, and Indranil Gupta. 2026. Load Fiction: How Your Load Generator is Misleading You. In *ACM Symposium on Cloud Computing Volume 2 (SoCC '26 Vol 2.), November 18–20, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3828161.3857900

1 Introduction

Load Generators (LGs) [1–3, 9, 19, 27, 31, 42–44] are widely used in industry to measure system reliability during normal operation, planned traffic surges, and major product launches [20, 38, 41].



(a) Closed Loop **(b) Open Loop**

Figure 1: Closed-loop vs. open-loop load generation.

They are also used for capacity planning [8, 16, 28] and outage postmortems [12, 40]. In academic systems research, LGs are critical for empirically comparing designs, justifying optimizations, and supporting claims about system behavior under load [21, 23, 39]. They have been used in research areas such as microservices [15], serverless [13], machine learning [34], and others. They provide latency and throughput measurements that resource-allocation and autoscaling systems use as feedback signals when making and evaluating policy decisions [10, 30].

Two kinds of LGs: An LG measures the performance of the "system under test" (SUT) by sending requests over the network, according to a configured rate. LGs fall into two operational models [37]: (i) open-loop and (ii) closed-loop.

When the goal is to measure how the SUT behaves under a specified workload, the desired property for LGs is *open-loop execution*. In this model, the LG issues requests to the SUT at predetermined times based on a configured schedule, independent of SUT responses (Figure 1b) [26, 37]. For example, if the intended workload specifies a constant rate of 100 requests per second (RPS), the LG should continue issuing requests at that rate even if the SUT slows down or speeds up. This response-independent scheduling is widely used in research and industry [3, 9, 45, 46].

Closed-loop LGs wait for a response before issuing the subsequent request (Figure 1a). Alternatively, a limit on the number of outstanding requests can be configured (default size 1). Closed-loop execution is reasonable for settings where strict preservation of an offered request rate is not the goal. Such cases arise when workloads sent by an LG aim to model response-driven user sessions, such as user requests to a ticketing service [11]. In these session-based workloads, a user may send a request, receive a response,

This work is licensed under a Creative Commons Attribution 4.0 International License.
SoCC '26 Vol 2., Singapore, Singapore
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2816-7/2026/11
https://doi.org/10.1145/3828161.3857900

optionally wait, and then issue another request. Each emulated user operates independently and does not rely on an external schedule to send subsequent requests. This makes a closed-loop LG easier to implement than an open-loop one.

This synchronous nature of closed-loop LGs makes them unsuitable for stressing a SUT's performance. For example, if the intended workload is 100 RPS and the SUT stalls for one second, the closed-loop LG will issue no further requests (while an open-loop LG would have issued 100 requests during that second). This leads to a phenomenon under closed-loop LGs called *coordinated omission* [33, 37], wherein the measured latency distribution appears to be artificially low, because fewer requests were sent. Engineers may thus underestimate latency and failure risks under heavy load. Some closed-loop LGs use statistical techniques to correct the latency distribution post-hoc [19, 42]. However, user reports show this can inflate latencies as well [5].

Issues with Open-Loop LGs: For open-loop LGs, correctness depends on two requirements: (i) the LG must send requests according to the configured schedule throughout the run and (ii) those requests must reach the SUT at the expected rate, in spite of network and transport delays. We call a violation of either requirement a *silent failure*. Such silent failures remain hidden in aggregate LG reports: the LG may report the correct total request count sent to the SUT, and therefore the target average rate over a fixed-duration test, even though the timing or distribution of arrivals at the SUT differs from the intended workload.

Assuring both requirements requires sufficient client-side capacity and transport capacity throughout the measurement run. Table 1 lists popular open-loop LGs. Many of these require users to configure finite resources, such as connections, to sustain the target rate [1, 19, 27, 42, 44]. This creates a cyclic dependency: choosing enough connections requires knowing how long requests will take under load, but estimating that behavior is the purpose of the load test. Some LGs address this by adaptively managing client-side resources, including connections [3, 43]. Yet neither static provisioning nor adaptation guarantees open-loop execution throughout the run: fixed budgets can become limiting as SUT behavior changes, while adaptive execution can create LG-side resource pressure. This has led to operators reporting unexpected under-delivery of requests [6], questioning deviations from the target rate [7], and observing discrepancies between LG-side and SUT-side measurements [4].

Contributions: Our work is the first empirical investigation of these issues in open-loop LGs. Our paper makes two contributions. *(I)* Via detailed empirical studies, we show that silent failures are common across LGs and configurations. Concretely, we distill these concerns into two research questions (RQs):

(RQ1) *Can an open-loop load test appear correct in aggregate LG reports while silently delivering a workload whose timing or distribution differs from the intended workload?*

(RQ2) *Do such silent failures materially affect latency measurements and the conclusions drawn in downstream system evaluations?*

To address RQ1, we demonstrate silent failures in four widely used LGs, showing they arise from transport budgets, resource constraints, and connection churn.

For RQ2, we report a series of findings, including the fact that LGs under-report p90 and p95 latency by 31.7% and 11.9%, respectively.

LG	Lang.	Adapt.	Report.	Attrib.
wrk2-DSB (NV*)† [15]	C	✗	✓	✗
Hyperfoil (v0.28.0) [19]	Java	✗	✓	✗
Vegeta (v12.13.0)† [43]	Go	✓	✓	✗
Locust (v2.43.4) [27]	Python	✗	✓	✗
JMeter (v5.6.3) [1]	Java	✗	✓	✗
h2load (v1.69.0)† [44]	C	✗	✓	✗
k6 (v1.7.1)† [3]	Go	✓	✓	✗

Table 1: Popular LGs. ***Adaptation*: LG adjusts resources to sustain the target rate.** ***Reporting*: LG provides summary and warning metrics.** ***Attribution*: LG classifies open-loop deviations across transport, client, and CPU bottlenecks. NV*: not versioned. LGs marked with † are studied in §4.**

(II) These findings motivate us to design an *observability* inspection layer that can be integrated with any open-loop LG. We present *XLG Inspector*, which treats the SUT as a black box, monitors LG-side signals to assess whether the intended workload is likely delivered faithfully to the SUT, and classifies the root cause of open-loop violations under LG-side bottlenecks across client concurrency, transport connections, and CPU resource constraints.

2 Operationalizing Open-Loop Fidelity

We characterize the conditions under which an open-loop LG faithfully executes an intended workload. Let $R(t)$, $S(t)$, and $A(t)$ denote the cumulative requests scheduled, sent by the LG, and arriving at the SUT application, respectively, by time t. For an ideal open-loop LG, SUT slowdowns may increase response times, but they should not reduce or reshape the workload sent or delivered to the SUT.

To operationalize these requirements for a discrete request process, we compare event counts in fixed time windows. Let $\mathcal{W}_\Delta$ be a set of Δ-second windows, and let $n_X(w)$ be the number of events of process X in window w. We define the largest relative under-delivery of process X with respect to reference process Y as

$$D_\Delta(X, Y) = \max_{w \in \mathcal{W}_\Delta} \frac{max\{0, n_Y(w) - n_X(w)\}}{max(1, n_Y(w))} \quad (1)$$

First, sending fidelity requires the LG's sends to follow the schedule:

$$D_\Delta(S, R) \leq \epsilon_{\text{send}} \quad (2)$$

This equation is said to be violated if, in at least one window in $\mathcal{W}_\Delta$, the requests sent are less than $(1 - \epsilon_{\text{send}})\times$requests scheduled.

Second, transport fidelity requires application arrivals to follow intended sends, after a fixed reference transport offset d_0:

$$D_\Delta(A, R_{d_0}) \leq \epsilon_{\text{arrival}} \quad (3)$$

where $R_{d_0}(t) = R(t - d_0)$. This equation is said to be violated if, in any window in $\mathcal{W}_\Delta$, application arrivals are less than $(1 - \epsilon_{\text{arrival}})\times$corresponding d_0-shifted intended sends. ϵ_{send} and $\epsilon_{\text{arrival}}$ are workload-specific tolerances, not universal constants. §4 uses $\Delta = 1$ s and $\epsilon_{\text{send}} = \epsilon_{\text{arrival}} = 0.10$. A one-second under-delivery greater than 10% constitutes a fidelity violation.

3 Related Work

Existing LG tools and reporting. Popular LGs are listed in Table 1. wrk2 [42], wrk2-DSB [15], Vegeta [43], and h2load [44] report aggregate request counts, throughput, and latency metrics. k6 [3] provides metrics for virtual users (VUs), which are LG-side execution

units that simulate clients and issue requests to the SUT. JMeter [1] reports comparable thread-level execution unit metrics. Both k6 and JMeter report HTTP timing metrics associated with connection behavior. Hyperfoil [19] and Locust [27] report resource-pressure signals, including connection blocking and LG CPU warnings. Although these signals are valuable, they are typically reported in isolation rather than to clarify whether an open-loop deviation has occurred. Our XLG Inspector addresses this diagnostic gap.

Self-correcting and statistically aware load generators. Treadmill [48] and Lancet [22] are related efforts to make LGs aware of their measurement limitations. Treadmill shows how open-loop versus closed-loop generation, client deployment scale, and client-side queuing can bias tail-latency measurements. Our work goes a step further and studies whether an LG, configured for open-loop operation, remains open-loop in reality. Lancet reports tail-latency percentiles with confidence intervals based on independent, stable measurements. It reports the experiment as inconclusive when its validity checks fail. Our XLG Inspector goes a step further by tracing failures to worker, connection, or CPU bottlenecks.

4 Silent Failures in Open-Loop Load Testing

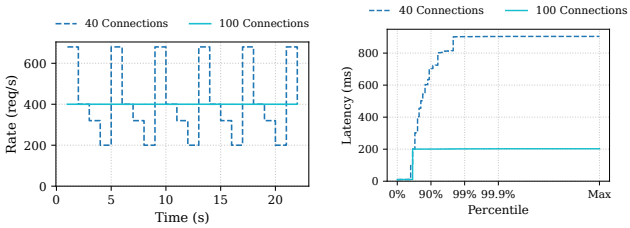
We present an empirical study of silent failures in open-loop LGs. §4.1 examines silent failures caused by protocol and transport constraints in wrk2-DSB and h2load, §4.2 examines LG-side resource pressure in Vegeta and k6, and §4.3 studies a real SUT.

Experimental Settings: All experiments were run on CloudLab [14, 35]. Experiments in §4.3 ran on Ubuntu 24.04 using CloudLab c220g2 machines, each with 40 vCPUs and 160 GB of RAM. In §5, the exponential-delay and HotelReservation experiments ran on two CloudLab c8220 nodes, each with 40 vCPUs and 256 GB of RAM. SocialNetwork ran on CloudLab c6320 nodes in the Clemson cluster, each with 28 CPUs and 256 GB of RAM.

4.1 Transport Budgets Reshape Delivered Load

We tackle two popular open-loop LGs: (i) wrk2-DSB [15], which is DeathStarBench's integration of the popular wrk2 (note that wrk2 is itself used in systems research [10, 21, 23]); and (ii) h2load [44] [24, 32, 47]. They require users to pre-allocate transport resources, e.g., connections or streams. We configure a "mock" SUT to alternate between a 1.5 s slow phase with 200 ms service time and a 2.5 s fast phase with 10 ms service time. wrk2-DSB ran at 400 RPS for 30 s with 40 and 100 HTTP/1.1 connections. Figure 2 shows that both configurations maintain the target send rate, but 40 connections cause a maximum 1-s arrival deficit of 50%, exceeding the 10% tolerance and thus violating Equation 3. This causes client-observed tail latency to include transport-induced waiting, inflating it relative to the 100-connection baseline. While we selected the 100-connection baseline after careful analysis, users generally lack such insight and may be oblivious to such erroneous results.

Even advanced HTTP/2's multiplexing cannot eliminate head-of-line (HOL) blocking. Since HTTP/2 enforces a finite stream-concurrency budget, when slow requests occupy all available streams, subsequently scheduled requests are delayed. We design a mock SUT that alternates between a 10 s slow phase with a 10 s service time and a 15 s fast phase with a 0 s service time. We evaluate this using the HTTP benchmarking tool h2load [44]. h2load



(a) SUT arrival rate (b) Latency percentiles

Figure 2: HTTP/1.1 connection budgets can hide delivery distortion. Both wrk2-DSB configurations send near the 400 req/s target, but with only 40 connections, (a) head-of-line blocking reshapes SUT arrivals and (b) inflates measured tail latency relative to the 100-connection run.

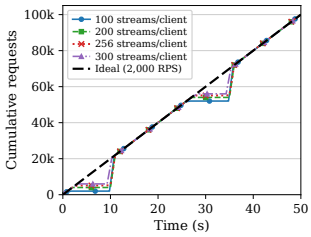


Figure 3: With h2load at 2K RPS, limited HTTP/2 stream capacity causes under-delivery during SUT slowdown, masked by the cumulative count at 50 s.

is configured with 20 clients, each sending 100 RPS, for 50 s. Figure 3 shows that concurrency is limited by the configured stream and connection budget. SUT-side logs indicate that fewer requests arrive during the slow phase than the intended open-loop schedule requires. However, the *aggregate* statistics do not reflect this. With 300 streams, the maximum 1-s arrival deficit is 100%, exceeding the 10% tolerance and violating Equation 3 (from §2). The LG's mean latency is 1.20 s, while the ground-truth mean latency is 4 s, a 70% under-reporting. We summarize our findings:

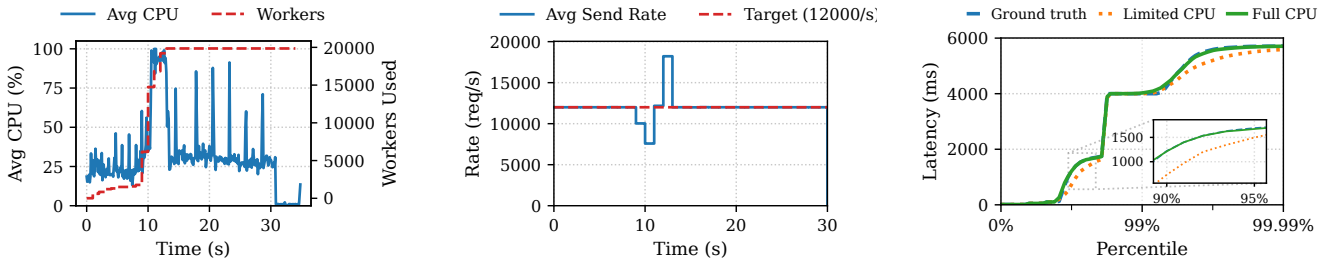
Takeaway 1: *Even when an LG reports sending load to the SUT at the intended rate, transport protocol constraints can lead to erroneous conclusions about the delivered load.*

4.2 LG Resource Pressure Breaks Open-Loop

In this section we study the Vegeta [43] and k6 [3] LGs.

4.2.1 Adaptive Worker Growth Creates LG-Side Pressure. In an attempt to avoid silent failures arising from fixed-budget LG configurations, Vegeta adaptively allocates additional workers to send requests when existing workers are occupied with outstanding requests. Nevertheless, we find that rapid worker growth can create LG-side CPU pressure and violate open-loop request assumptions.

We model transient SUT behavior in which a short-lived shift in workload mix or backend performance increases the fraction of long-running requests. We construct a limited-concurrency SUT where an arriving request can have a service time of 10 ms, 50 ms, 100 ms, or 4 s. Initially, the SUT assigns requests to the faster tiers, then enters a 1 s *shock* phase where it increases the fraction of 4 s requests. Thereafter it returns to the initial faster regime. We run Vegeta at 12K RPS for 30 s against a SUT handling 4800 concurrent requests. We compare (10 trials) an 8-CPU constrained



(a) CPU pressure and worker growth (b) Send-rate perturbation (c) Latency distribution shift

Figure 4: Vegeta [43] under SUT stress: (a) adaptive worker growth induces LG-side CPU pressure, (b) disrupting request sending, and (c) reduced arrivals during the latency-critical interval cause the 8-CPU run to under-report tail latencies.

LG, emulating a smaller load-generation node, vs. a 56-CPU baseline that closely emulates ideal open-loop delivery; both use Vegeta's default adaptive worker settings and no connection cap.

Figure 4 shows that the 56-CPU baseline closely tracks the intended workload. Yet adaptive worker growth on the 8-CPU client creates a CPU spike that perturbs request sending near the shock phase: all constrained runs violate Equation 2: maximum 1-s send under-delivery averages 28.42% (95% t-CI = confidence interval from the t-distribution: [22.47%, 34.36%]; minimum 14.55%), exceeding the 10% tolerance. SUT-arrival under-delivery similarly violates Equation 3 in all 10 trials (mean 28.26%; 95% t-CI: [22.34%, 34.18%]; minimum 14.52%) over latency-critical windows. The 56-CPU baseline violates the send and arrival conditions over these windows in only 1 and 2 trials, respectively. While its aggregate send rate later recovers, the 8-CPU client sends fewer requests near the shock, decreasing queuing at the SUT. Across 10 paired runs, mean SUT queue wait during recovery decreases by 169.3 ± 50.6 ms vs. the 56-CPU baseline—mean p95 queuing delay is 1078.9 ms in the 8-CPU trace, vs. 1363.7 ms in the ideal trace. The 8-CPU client has a mean p90 latency of 792.750 ± 67.729 ms, vs. 1160.698 ± 80.332 ms for the 56-CPU baseline, under-reporting p90 by 31.7%. It reports a mean p95 of 1396.076 ± 118.011 ms, vs. 1584.113 ± 138.093 ms for the 56-CPU baseline, under-reporting p95 by 11.9%.

We explain this behavior as follows. Shock phase requests consume the SUT's available concurrency for several seconds, causing queue buildup to persist. Under-delivery during this phase makes the SUT appear less congested than it actually is.

Takeaway 2: *Adaptive LGs designed to preserve open-loop execution may avoid fixed transport bottlenecks but introduce LG-side CPU pressure, coupling the delivered workload to SUT slowdown. Aggregate reports can hide intermittent CPU spikes and distort SUT conclusions.*

4.2.2 SUT-Induced Connection Churn Creates Time-Local Under-Delivery. To model the effect of broken connections, we introduce a *connection-churn window*. The SUT in this experiment is an HTTPS server with HTTP/1.1. During normal operation, requests have service times of 10 ms. This is followed by the churn window, and then a brief slow phase when requests receive higher 2 s service times. The churn phase requires subsequent k6 requests to establish new TCP/TLS connections instead of reusing existing ones.

We study the k6 LG [3] under its open-model constant-arrival-rate executor [17]. With delayed SUT responses, k6 needs available

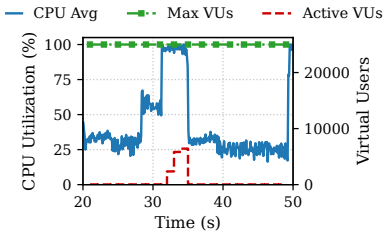


Figure 5: An SUT closing connections causes the k6 client to spend CPU and VU capacity establishing new ones, creating LG-side pressure ahead of the latency-critical interval.

virtual users (VUs) to maintain the scheduled iteration rate. We overprovision k6 with 25K preallocated VUs. We run k6 at 3000 RPS, comparing a close-to-ideal 56-CPU baseline vs. a constrained 8-CPU variant, with identical VU configuration and SUT behavior.

Figure 5 shows that connection churn causes a brief CPU spike on the 8-CPU client. During and immediately after this spike, SUT arrivals decrease: across 10 trials, maximum 1-s under-delivery averages 68.24% (95% t-CI: [58.12%, 78.36%]; minimum 44.50%), exceeding the 10% tolerance in every run and violating Equation 3. In contrast, the provisioned baseline exceeds this threshold in only 1 run. Since the arrival rate later recovers and k6 reports no explicit dropped-iteration warnings, the constrained run appears healthy. Across 10 trials, however, the 8-CPU client delivers 2961.60 ± 850.67 requests during the slow phase, compared with 5993.20±3.29 for the 56-CPU baseline—a 50.6% reduction. With fewer requests receiving the 2 s service time, the constrained client reports a p95 latency of 448.43 ± 84.38 ms, compared with 2000.42 ± 0.01 ms for the baseline, under-reporting p95 by 77.6%.

Takeaway 3: *Even with sufficient client-side concurrency, an LG may violate open-loop semantics when SUT behavior raises load-generation cost. Aggregate reports can hide the resulting intermittent LG-side pressure and distort SUT conclusions under duress.*

4.3 Silent Failures Affect System Evaluations

We go beyond mock SUTs to a real SUT. We use Cilantro's [10] microservice variant under DeathStarBench's [15] HotelReservation service. Cilantro is an online resource-allocation system that uses performance feedback, including tail-latency measurements, to choose resource allocations. We compare Cilantro against its three baselines: *ResourceFair* (equal resource allocation), *EvoAlg* (optimization of the 99th percentile latency), and $\epsilon - Greedy$ (probabilistic selection of the lowest 99th percentile latency).

Workload Setting	EvoAlg	ϵ-Greedy	ResourceFair
wrk2 fixed, 32t/32c	6,567	3,613	**2,150**
wrk2 fixed, 40t/1000c	**2,226**	2,432	2,643
wrk2 fixed, 40t/3000c	**882**	1,063	5,351
k6, 1500 VUs	1,790	**954**	1,973
k6, 3000 VUs	**892**	1,072	1,556
k6, 6000 VUs	719	**424**	2,054

Table 2: Earth mover distance [36] between Cilantro and baseline p99-latency distributions across LG configurations. The closest Cilantro baseline (in bold) changes.

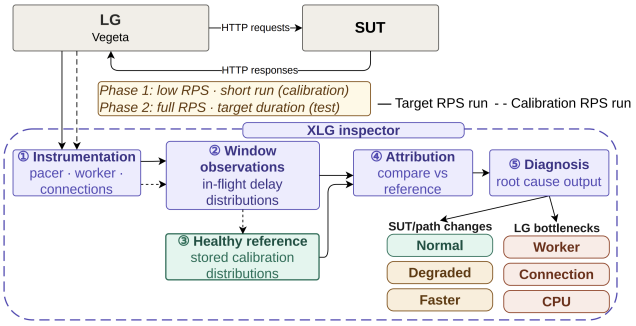


Figure 6: XLG Inspector architecture.

The run for each policy/LG lasts at least 8 hours. Table 2 uses Earth mover distance (EMD, or Wasserstein distance) [36] to compare the variants. Lower EMD indicates the baseline's p99-latency is more similar to Cilantro's. *The wide variation of EMDs across the variants and across workloads shows that it is difficult to derive conclusions about relative performance across these variants.*

Takeaway 4: *Under open-loop LGs, empirically comparing variants of a SUT can be inconsistent and fraught with erroneous conclusions.*

5 XLG Inspector: Observability for LGs

We present XLG Inspector (XLG, in short), an LG-side *observability* layer integrated with Vegeta [43] that classifies LG runs into failure categories. XLG treats the SUT as a black box and uses only LG-side signals, making XLG applicable to any SUT.

5.1 Taxonomy, Signals, and Architecture

XLG organizes runs into a taxonomy of failures. While not exhaustive, it covers the main failure modes observed in §4:

(1) **CPU contention.** At high target rates, the LG is compute-bound and fails to send at target rate, violating Equation 2.
(2) **Connection bottlenecks.** Connection limits prevent requests from arriving at the target rate, violating Equation 3.
(3) **Worker bottlenecks.** When sender workers are occupied, the target send rate is not met, violating Equation 2.
(4) **SUT/path changes.** The SUT or network path may become slower, faster, or more variable (not LG failures).

Figure 6 shows XLG's architecture. XLG analyzes LG behavior using fixed-size windows at fine granularity. In each window, XLG augments Vegeta's pipeline with instrumentation for three signals. (i) *In-flight requests*: assigned to a Vegeta worker but have not received SUT responses (counts are sampled during the window). (ii) *Scheduler delay*: time between when the internal Vegeta pacer schedules a request and when a Vegeta worker starts processing it. (iii) *Connection delay*: time a Vegeta worker needs to obtain a transport connection before sending the SUT a request. XLG compares these observations to a healthy baseline (reference) collected during calibration, and then applies the per-window attribution logic online, shown in Figure 7. Six total root causes are possible. The SUT/path behavior (item 4) has three further subcategories: Normal, SuTDegraded, and SuTFaster, indicating (respectively) SUT/path latency consistent with, higher than, or lower than the healthy reference. The remaining states indicate CPU pressure delaying request scheduling (CpuContention; item 1), workers waiting for a connection (FewConnections; item 2), or a worker cap too low to sustain open-loop execution (FewWorkers; item 3).

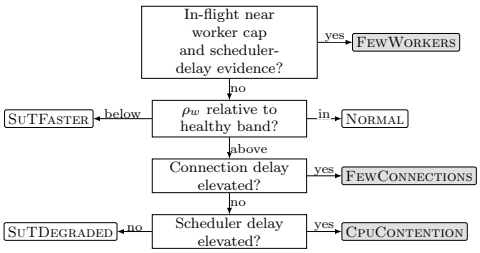


Figure 7: XLG's per-window attribution logic. Shaded boxes are terminal LG-side diagnoses.

Portability to Other LGs. XLG requires only three LG-side signals: when a request was scheduled, when a worker begins handling the request, and how long it takes for the request to obtain a connection. These already appear in many LGs. k6 [3] uses VUs to maintain an arrival rate and reports time spent waiting for a free TCP connection slot. Hyperfoil [19] separates users/sessions from HTTP connection pools, making scheduler and connection bottlenecks natural to instrument. For fixed-connection tools like wrk2-DSB [15], connection bottlenecks may map directly to configured connection limits. When Locust [26] is configured as open-loop, CPU contention and user/concurrency limits may remain relevant signals.

5.2 Calibration and Attribution

XLG analyzes execution windows using an in-flight deviation ratio based on Little's Law [25], combining it with live logs of request-level scheduler and connection delay distributions. XLG's in-flight deviation ratio, ρ_w, identifies departures from healthy LG behavior. In open-loop load generation, the expected in-flight request count via Little's Law is $L^* = \lambda \cdot W$, where λ is the target request rate and W is the SUT's mean processing time. During a load test, for each window w, XLG computes $\rho_w = \frac{\bar{L}_w}{L^*}$, where $\bar{L}_w$ = the observed average count of in-flight requests in window w, and L^* = expected in-flight count computed from intended target rate and mean latency measured during healthy calibration. XLG compares ρ_w vs. calibrated healthy band $\rho_0 \pm \epsilon$, where ρ_0 is the empirical center of the in-flight deviation ratio of healthy runs. Lower values indicate LG under-delivery or a faster SUT, while higher values indicate request accumulation caused by SUT slowdown or an LG-side bottleneck.

Figure 8 shows the request-level delay signatures XLG uses to attribute deviations in ρ_w. Under a worker bottleneck, Vegeta workers are occupied waiting for SUT responses, raising the time new requests wait to be assigned to an available worker (scheduler delay). Connection delay also rises under connection bottlenecks as Vegeta workers wait for a free connection to dispatch requests.

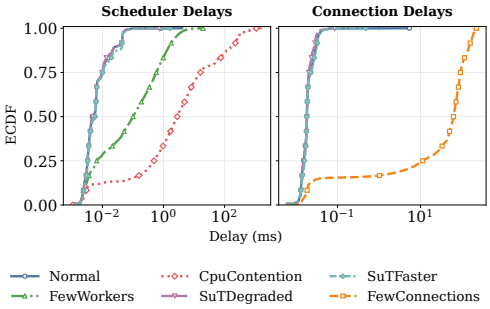


Figure 8: ECDFs from a Vegeta LG load test at 1.5K RPS with HotelReservation [15] for various failures and SUT path changes: scheduler (left) and connection (right) delays.

Fault	CPU cont.			Few conn.			Few workers		
RPS	1K	2K	3K	1K	2K	3K	1K	2K	3K
Send/s	999.36	1996.36	2940.95	1000.01	2000.02	3000.01	999.86	2000.02	2998.82
Δ lat.	+304.70%	+99.17%	+108.55%	+342.41%	+325.47%	+325.00%	-6.36%	-5.82%	-6.68%

Table 3: Aggregate-rate and latency distortion for ExpServer bottlenecks. Latency change averaged across severities for each fault type. Positive values indicate latency inflation; negative values indicate under-reporting vs. reference run.

5.3 Evaluation

We evaluate XLG in two scenarios: (i) LG-side failures hidden by aggregate reports and (ii) microservices. We introduce contentions of CPU, worker, and connection at three severity levels. Higher severities impose tighter baseline-derived resource caps, while healthy references impose none. We validate healthy reference behavior with *ExpServer*, which uses an exponential distribution (mean 20 ms) for service times. With no imposed LG resource cap, XLG has a mean per-run $D_1(S,R) = 0.09\%$ (95% t-CI: [0.09%, 0.10%]) and reports mean latency consistent with the expected distribution. We emulate SUT/path changes by varying LG-SUT network latency.

Failures hidden by aggregate reports. Against *ExpServer*, we inject LG-side bottlenecks that remain hidden in the LG's aggregate reports. Although aggregate send rates remain within 2% of their targets, the LG-reported mean latencies are distorted (Table 3). We evaluate two scenarios. First, in the *same-RPS* test, calibration and evaluation use different SUT profiles at the same request rate. We find that **XLG correctly classifies** 9/9 **connection,** 9/9 **worker, and** 7/9 **CPU-contention bottlenecks**. The two CPU-contention errors are classified as FewConnections and SuTDegraded.

Second, in the *cross-RPS* test, calibration and evaluation use different SUT profiles and request rates. We find that **XLG correctly classifies** 6/9 **connection,** 8/9 **worker,** 8/9 **CPU-contention, and** 8/9 **path-change cases**. Table 4 (a) shows full class-by-class results for both ExpServer settings. The three connection errors are classified as CPU contention. Most errors occur when calibration at 1K RPS is applied at 3K RPS: the scheduler-delay threshold learned at lower load is too conservative for the higher-load run. Hence scheduler-delay evidence fires even when the injected condition is not a worker bottleneck or CPU contention. Overall, **scheduler delay is more load-sensitive than other delay types**.

Microservices. We run XLG with two DeathStarBench microservice applications [15]. HotelReservation exhibits diverse latency profiles, with mean latencies of 19.512 ms, 31.371 ms, and 568.108 ms at 1K, 1.5K, and 2K RPS, respectively. SocialNetwork exhibits stable mean latencies of 13.758 ms, 13.111 ms, and 13.415 ms, respectively.

(a) ExpServer

T\P	FC	FW	CC	N	D	F
FC	**15**	0	3	0	0	0
FW	0	**17**	1	0	0	0
CC	1	0	**15**	0	2	0
N	0	0	0	**3**	0	0
D	0	0	1	0	**11**	0
F	0	0	0	0	0	**3**

(b) Microservices

T\P	FC	FW	CC	N	D	F
FC	**18**	0	0	0	0	0
FW	0	**18**	0	0	0	0
CC	0	0	**17**	0	1	0
N	0	0	0	**1**	2	3
D	0	0	0	1	**5**	0
F	0	0	0	0	2	**4**

Table 4: Confusion matrices for XLG's diagnoses: (a) ExpServer, (b) DeathStarBench. Rows are experimental ground-truth; columns are XLG's diagnoses; bold shows correct diagnoses. FC: Few Connections; FW: Few Workers; CC: CPU Contention; N: Normal; D: SUT Degraded; F: SUT Faster.

SUT	Severity	FewConn.	FewWorkers	CPUCont.
HotelReservation	Mild	8.67	8.67	3.50
	Moderate	7.67	6.33	7.67
	Severe	7.00	4.33	6.33
SocialNetwork	Mild	15.00	11.00	3.00
	Moderate	11.00	5.00	3.00
	Severe	5.00	4.00	3.00

Table 5: Average 1-s window index when XLG confirms a terminal LG-side diagnosis. Confirmation requires 3 consecutive windows with the same terminal label. Lower is faster.

Across both benchmarks in the cross-RPS setting, **XLG correctly classifies all** 18 **connection bottleneck runs, all** 18 **worker bottleneck runs, and** 17 **of** 18 **CPU contention runs.** Table 4 (b) gives the complete pooled class-by-class results. Table 5 shows that XLG attributes an LG bottleneck sooner as its severity increases.

Results Summary. XLG identifies LG-side bottlenecks that are silent to aggregate-rate reporting. XLG's signals transfer across SUT profiles and, to some extent, across RPS settings. The main exception is scheduler delay under the more extreme ExpServer cross-RPS case (1K RPS applied to 3K RPS). Microservice experiments are far more stable in spite of the workload having varied latency profiles. Hence XLG's signals are largely transferable, while scheduler-delay thresholds may require load-aware calibration for wider RPS shifts.

XLG Overhead. Against a 50 ms Go server at 4K, 8K, & 16K RPS over three trials, XLG preserves time-local sending with a mean per-run 1-s sending discrepancy of $D_1(S,R) = 0.31\%$. XLG's mean latencies are 50.84, 50.84, and 51.23 ms, respectively, vs. 50.85, 50.85, and 51.06 ms for vanilla Vegeta. At 16K RPS, XLG's p99 latency is 62.54 ms, vs. 61.21 ms for vanilla Vegeta. Vegeta averages 5.81 CPU cores and 83 MiB RSS (resident set size = RAM used by a process), while the consumer uses 0.19 CPU cores and 149 MiB RSS.

6 Conclusion

Our findings show that caution is warranted when interpreting load generator results. Aggregate LG results may hide intricacies of system behavior, leading to erroneous conclusions about robustness and competing designs. An observability tool like XLG can help.

Future Directions. XLG can be made adaptive via live inspection of streaming ingestion under CPU pressure, profiling additional load points, and interpolating references. Bursty workloads may cause under-delivery even in trace-driven LGs [18, 29].

Acknowledgments. This work was supported in part by NSF grant CNS 2504595, by IIDAI, and a gift from Microsoft. All our code is available at https://dprg.cs.uiuc.edu/traces/go.php?id=45.

References

[1] [n. d.]. JMeter. https://jmeter.apache.org/

[2] [n. d.]. Load testing designed for DevOps and CI/CD. https://gatling.io/

[3] [n. d.]. Load testing for engineering teams | Grafana k6. https://k6.io/

[4] 2019. The wrk2 console latency is much larger the jaeger tracing latency when the rps is high · Issue #10 · delimitrou/DeathStarBench — github.com. https://github.com/delimitrou/DeathStarBench/issues/10. Accessed: 2025-02-24.

[5] 2019. Wrong latency distribution on the benchmark test · Issue #81 · giltene/wrk2 — github.com. https://github.com/giltene/wrk2/issues/81. Accessed: 2025-02-24.

[6] 2023. Constant arrival rate inconsistent number of requests · Issue #3052 · grafana/k6 — github.com. https://github.com/grafana/k6/issues/3052. Accessed: 2025-03-31.

[7] 2023. Have parameter to abort if test doesn't reach target iterations per second · Issue #2837 · grafana/k6 — github.com. https://github.com/grafana/k6/issues/2837. Accessed: 2025-02-24.

[8] Amazon Web Services. [n. d.]. REL07-BP04 Load Test Your Workload. https://docs.aws.amazon.com/wellarchitected/latest/framework/rel_adapt_to_changes_load_tested_adapt.html. AWS Well-Architected Framework. Accessed: 2026-06-17.

[9] Artillery. [n. d.]. Artillery · The Complete Load Testing Platform — artillery.io. https://www.artillery.io/. Accessed: 2025-03-28.

[10] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2023. Cilantro: Performance-Aware Resource Allocation for General Objectives via Online Feedback. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*. USENIX Association, Boston, MA, 623–643. https://www.usenix.org/conference/osdi23/presentation/bhardwaj

[11] Shaun Brown. 2025. Understanding Workload Models for Load Tests. https://gatling.io/blog/workload-models-in-load-testing. Accessed: 2026-07-01.

[12] Coinbase Tech Blog. 2021. Incident Post Mortem: October 27, 2021. https://www.coinbase.com/en-in/blog/incident-post-mortem-october-27-2021.

[13] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *International Middleware Conference* (Québec City, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 64–78. https://doi.org/10.1145/3464298.3476133

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *USENIX Annual Technical Conference (USENIX ATC '19)*. USENIX Association, Renton, WA, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[15] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013

[16] Google Cloud. [n. d.]. Guidelines for Load Testing Backend Services with Application Load Balancers. https://docs.cloud.google.com/load-balancing/docs/backend-service-load-testing. Accessed: 2026-06-17.

[17] Grafana Labs. [n. d.]. Constant arrival rate. https://grafana.com/docs/k6/latest/using-k6/scenarios/executors/constant-arrival-rate/. Accessed: 2026-06-26.

[18] Grafana Labs. 2026. Create tests from recordings. https://grafana.com/docs/k6/latest/using-k6/test-authoring/create-tests-from-recordings/. Grafana k6 documentation. Accessed: 2026-06-21.

[19] Hyperfoil Project. 2023. Hyperfoil Performance Testing — Overview. https://hyperfoil.io/docs/overview/.

[20] Inditex Tech. 2025. How Inditex Handles Massive Traffic Spikes: Building a Custom Load Test Platform. https://medium.com/@InditexTech/how-inditex-handles-massive-traffic-spikes-building-a-custom-load-test-platform-be9ffe3ede59.

[21] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. 2023. SelfTune: Tuning Cluster Managers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*. USENIX Association, Boston, MA, 1097–1114. https://www.usenix.org/conference/nsdi23/presentation/karthikeyan

[22] Marios Kogias, Stephen Mallon, and Edouard Bugnion. 2019. Lancet: A self-correcting Latency Measuring Tool. In *USENIX Annual Technical Conference (USENIX ATC '19)*. USENIX Association, Renton, WA, 881–896. https://www.usenix.org/conference/atc19/presentation/kogias-lancet

[23] Mingyu Liang, Yu Gan, Yueying Li, Carlos Torres, Abhishek Dhanotia, Mahesh Ketkar, and Christina Delimitrou. 2023. Ditto: End-to-End Application Cloning for Networked Cloud Services. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 222–236. https://doi.org/10.1145/3575693.3575751

[24] Shengkai Lin, Shizhen Zhao, Peirui Cao, Xinchi Han, Quan Tian, Wenfeng Liu, Qi Wu, Donghai Han, and Xinbing Wang. 2025. ONCache: A Cache-Based Low-Overhead Container Overlay Network. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '25)*. USENIX Association, Philadelphia, PA, 979–998. https://www.usenix.org/conference/nsdi25/presentation/lin-shengkai

[25] John D. C. Little and Stephen C. Graves. 2008. Little's Law. In *Building Intuition: Insights From Basic Operations Management Models and Principles*, Dilip Chhajed and Timothy J. Lowe (Eds.). Springer US, Boston, MA, 81–100. https://doi.org/10.1007/978-0-387-73699-0_5

[26] Locust Cloud. 2022. Closed vs Open Workload Models in Load Testing. https://locust.cloud/blog/open-vs-closed-workload-models/.

[27] Locustio. [n. d.]. Locustio/locust: Write scalable load tests in plain Python. https://github.com/locustio/locust

[28] Microsoft Azure Architecture Center. 2026. Architecture Strategies for Performance Testing. https://learn.microsoft.com/en-us/azure/well-architected/performance-efficiency/performance-test. Accessed: 2026-06-17.

[29] NVIDIA Corporation. 2026. Load Generator Options Reference. https://docs.nvidia.com/aiperf/benchmark-modes/load-generator-options-reference. AIPerf Documentation, Latest v0.9.0. Accessed: 2026-06-21.

[30] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2024. Graph Neural Network-Based SLO-Aware Proactive Resource Autoscaling Framework for Microservices. *IEEE/ACM Transactions on Networking* 32, 4 (2024), 3331–3346. https://doi.org/10.1109/TNET.2024.3393427

[31] ProcessOne. [n. d.]. Tsung. https://github.com/processone/tsung

[32] Kritwara Rattanaopas and Phantiga Wattanakul. 2024. HTTP/2 Web Performance Characteristics with Link-aggregation Mode Under DDoS Attack. In *International Electrical Engineering Congress (iEECON '24)*. 1–4. https://doi.org/10.1109/iEECON60677.2024.10537969

[33] Red Hat Performance Blog. 2020. Your Load Generator is Probably Lying to You — Take the Red Pill and Find Out Why. https://redhatperf.github.io/post/coordinated-omission/.

[34] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, et al. 2020. MLPerf Inference Benchmark. In *International Symposium on Computer Architecture (ISCA '20)*. 446–459. https://doi.org/10.1109/ISCA45697.2020.00045

[35] Robert Ricci, Eric Eide, and The CloudLab Team. 2014. Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications. *USENIX ;login:* 39, 6 (Dec. 2014). https://www.usenix.org/publications/login/dec14/ricci

[36] Y. Rubner, C. Tomasi, and L.J. Guibas. 1998. A metric for distributions with applications to image databases. In *IEEE International Conference on Computer Vision (ICCV '98)*. 59–66. https://doi.org/10.1109/ICCV.1998.710701

[37] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open Versus Closed: A Cautionary Tale. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*. USENIX Association, San Jose, CA, 239–252. https://www.usenix.org/conference/nsdi-06/open-versus-closed-cautionary-tale

[38] Shopify Engineering. 2023. Performance Testing At Scale—for BFCM and Beyond. https://shopify.engineering/scale-performance-testing.

[39] Gagan Somashekar, Karan Tandon, Anush Kini, Chieh-Chun Chang, Petr Husak, Ranjita Bhagwan, Mayukh Das, Anshul Gandhi, and Nagarajan Natarajan. 2024. OPPerTune: Post-Deployment Configuration Tuning of Services Made Easy. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)*. USENIX Association, Santa Clara, CA, 1101–1120. https://www.usenix.org/conference/nsdi24/presentation/somashekar

[40] Spotify Engineering. 2013. Incident Management at Spotify. https://engineering.atspotify.com/2013/6/incident-management-at-spotify.

[41] Spotify Engineering. 2023. Load Testing for 2022 Wrapped. https://engineering.atspotify.com/2023/3/load-testing-for-2022-wrapped.

[42] Gil Tene. [n. d.]. giltene/wrk2. https://github.com/giltene/wrk2

[43] Tsenart. [n. d.]. TSENART/Vegeta: HTTP Load Testing Tool and library. it's over 9000! https://github.com/tsenart/vegeta

[44] Tatsuhiro Tsujikawa. 2015. Nghttp2.org. https://nghttp2.org/

[45] Twitter. 2018. Iago, A Load Generator. https://github.com/twitter/iago2.

[46] Uber Engineering. 2022. Introducing Shadower: A Minimalistic Load Testing Tool. https://www.uber.com/us/en/blog/introducing-shadower-a-minimalistic-load-testing-tool/.

[47] Cheng Zhang, Xiangming Wen, Luhan Wang, Zhaoming Lu, and Lu Ma. 2018. Performance Evaluation of Candidate Protocol Stack for Service-Based Interfaces in 5G Core Network. In *IEEE International Conference on Communications Workshops (ICC Workshops '18)*. 1–6. https://doi.org/10.1109/ICCW.2018.8403675

[48] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. 2016. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *International Symposium on Computer Architecture (ISCA '16)*. 456–468. https://doi.org/10.1109/ISCA.2016.47