LoadGuard-JMeter: An SLO-Aware Closed-Loop Elastic Load Generation Plugin for Apache JMeter

Faisal Bahadur, Ibrar Afzal, Tayyba Bibi, Tanzeela Qazi
Department of CS&IT, Hazara University, Mansehra, Pakistan
*Correspondence: faisal@hu.edu.pk

Citation | Bahadur. F, Afzal. I, Bibi. T, Qazi. T, "LoadGuard-JMeter: An SLO-Aware Closed-Loop Elastic Load Generation Plugin for Apache JMeter", IJIST, Vol. 8 Issue. 10 pp 4042-4071, November 2026

Received| Oct 25, 2026 **Revised|** Nov 04, 2026 **Accepted|** Nov 06, 2026 **Published|** Nov 08, 2026.

A practical challenge in performance testing web, API, cloud-based, and microservice systems is selecting a workload level that can expose meaningful behavior without overloading the target system. Apache JMeter provides several mechanisms for generating and controlling load. However, in most JMeter tests, the workload profile is configured before execution and remains largely fixed during the run. This makes SLO-aware testing difficult when the safe operating range is not known in advance. This paper presents LoadGuard-JMeter, an SLO-aware closed-loop load-generation plugin for Apache JMeter to dynamically control the workload according to runtime SLO conditions. The plugin combines an Elastic Timer, Metrics Listener, SLO Feedback Controller and Decision Log Writer to dynamically adjust request pacing based on p95 response-time and error-rate SLOs. When the system is within the safe zone, the request rate is increased; when it approaches the SLO, the rate remains constant; and when a breach occurs, the rate is decreased. The controller's decisions are recorded in a separate CSV file. The system was tested using the TeaStore microservice benchmark under an 800 RPS stress load. The evaluation included five independent experimental runs for each scheme. Based on the five-run mean results, LoadGuard-JMeter improved the average response time by 96.54% and 95.37%, p95 latency by 94.70% and 95.62%, and p99 latency by 94.32% and 95.87%, compared to FFA and TSTCTG respectively. It also reduced the mean p95 SLO-violation rate from 100% for both baselines to 22% and maintained an error rate of 0%. The controller executed INCREASE, HOLD and DECREASE commands during runtime, demonstrating workload adaptability. These results were achieved at a lower mean successful throughput of 88.15 requests/s, compared to 741.20 requests/s for FFA and 154.10 requests/s for TSTCTG, showing that LoadGuard-JMeter maintains SLO stability by regulating the workload rather than maximizing throughput. The lower response times and fewer SLO violations were therefore achieved by reducing the applied workload, rather than improving response times at the same throughput.

Keywords: Apache JMeter, SLO-Aware Load Testing, Closed-Loop Feedback Control, Elastic Load Generation, Microservice Performance Testing.

Introduction:

Performance testing is used to determine if software can meet its service level objectives (SLOs) under increased load [1][2][3]. This is especially important for web, API, cloud-based, and microservice systems, as increasing the number of requests can have an immediate impact on response times, error rates, and system stability.

Apache JMeter [4] is the most popular performance testing tool due to its flexibility with regards to test plans, support for HTTP and API loads, distributed testing and plugin functionality [5][6]. However, the load is usually defined in advance by setting values such as number of threads, ramp up time, delays, number of loops or throughput before running the test. This is a good option if the safe operating range of the system is known; however, it can become difficult if the low, beneficial, and unsafe load levels are uncertain [7]. This can result in the system not being pushed to its true capacity limits. Alternatively, a high load may overload the system, preventing meaningful observations. You may need to run multiple tests to find the right load [8].

JMeter provides timers, throughput-shaping plugins, and threshold-based controls that help define workload profiles or stop a test after unsafe behavior appears [9]. Still, these mechanisms mostly depend on values selected before execution. They do not provide a lightweight feedback-control mechanism that changes request pacing during the same test according to live p95 latency and error-rate SLO conditions. Although adaptive workload-generation approaches exist, most are implemented as external frameworks rather than components that can be added directly to a normal JMeter test plan [10].

Recent Research:

Apache JMeter provides two practical methods for generating workloads: Free-Form Arrivals (FFA) [11] and Throughput Shaping Timer with Concurrency Thread Group (TSTCTG) [12][13]. The former method schedules requests according to a specified arrival pattern. The latter method schedules requests by using throughput shaping and concurrency to meet a desired request rate. Neither method adapts a target request rate dynamically during runtime using real-time p95 response time and error rate SLO feedback. Several studies have been conducted on workload generation, performance testing and run-time adaptation. For example, DYNAMO [14] introduced in-test workload adaptation to expose performance issues dependent on the workload. DYNAMOJM [15] extended the work and provided dynamic workload adaptation within JMeter. RELOAD [16] and SaFReL [17] used reinforcement learning and self-adaptation strategies to improve performance testing. PerfScout [18] examined adaptive workload generation for software performance testing. [19] proposed a solution to adapt timeout strategies during microservice communication using SLO. Recently, JMeter MCP Server [20] has provided AI assisted support for test execution and analysis. However, these solutions only focus on general test frameworks, learning-based adaptation, adapting microservices or automating tests. Specifically, none of these solutions focus on a lightweight approach to managing request pacing during testing using feedback from both p95 response time and error rate SLOs by classifying risky zones. LoadGuard-JMeter is therefore proposed to provide direct integration of SLO aware closed-loop pacing during test plan creation within JMeter.

Based on these issues, the following research question can be identified:

Research Question:

RQ1: How effectively can runtime interval metrics be used to dynamically adjust the request rate during a JMeter test?

RQ2: How accurately can p95 response-time and error-rate SLO usage be mapped to SAFE, HOLD, and VIOLATION zones and their corresponding controller actions?

RQ3: How much does LoadGuard-JMeter reduce average response time, p95/p99 latency, error rate, and p95 SLO-violation windows compared with FFA and TSTCTG?

RQ4: What is the impact of SLO-aware request pacing on successful throughput compared with FFA and TSTCTG?

These research questions are addressed through the controller design and evaluation. Research questions 1 (RQ1) and 2 (RQ2) are investigated using the interval-level metrics, SLO classification equations, controller algorithm, and decision log shown in Algorithm 1 and Table 5. RQ3 is investigated using the response-time, error-rate, and violation-window metrics for the three schemes in Tables 3, 4, and 6. RQ4 is investigated using the successful throughput metrics and comparison between throughput and latency in tables 3 and 6. Additional evidence on the consistency of the results can be seen from the five independent runs.

Contributions:

To solve these questions, this paper presents LoadGuard-JMeter, an SLO-aware closed-loop elastic load-generation plugin for Apache JMeter. A new Elastic Timer and Metrics Listener are added to the JMeter process. The Listener gathers metrics on a per-interval basis such as throughput, active threads, response time average, p95/p99 response time, error percentage and SLO percentage. The controller uses this data to determine the applicable risk zone and decide whether to increase, maintain, or decrease the request rate. The Elastic Timer then increases or decreases the target request rate accordingly. Metrics are then stored in CSV files so they can be analyzed later. The main aim of this project is to take pre-scripted JMeter tests and make them dynamic, using SLOs to manage the load. The test will start with a certain request rate and adapt based on its performance. If the system is performing well and safely under the SLO boundary the target request rate will increase. If it is nearing the SLO boundary then the request rate will stay the same. If the system has violated the SLO then the target request rate will decrease. This provides a new feedback loop within JMeter.

Main Contributions:

A JMeter plugin which supports closed-loop SLO-aware request pacing at runtime.

A risk zone controller which takes a p95 latency and error rate SLO into consideration when deciding to increase, maintain and reduce pacing. An Elastic Timer, Metrics Listener, and CSV decision logger integrated into the standard JMeter test-plan workflow.

An empirical evaluation on the TeaStore benchmark [21] comparing closed-loop pacing with JMeter workload-control baselines and analyzing latency, errors, SLO violations, and throughput trade-off.

Flow of Research:

This research starts by identifying problems with static workload generation. Then, LoadGuard-JMeter was designed and developed. This sets up the test environment. FFA and TSTCTG were set up ready for testing. All three schemes were tested on the 800 RPS stress scenario. Results were gathered by capturing statistics and controller decisions in CSV files. These results were then compared in order to see how well the schemes complied with SLO's, any issues with response times, error rate and throughput. These comparisons were used to draw the final research conclusions. Each scheme was also run 5 times in order to find results that are consistent using statistical analysis. The results were then analyzed to provide conclusions on the performance and effectiveness of LoadGuard-JMeter.

Novelty of the Study:

By incorporating a novel SLO-aware feedback loop that ensures the appropriate pacing of requests in LoadGuard-JMeter, this study overcomes the issues associated with fixed workload generation methods. It utilizes the real time p95 response time and error-rate SLO metrics to determine the risk zone of operation. The target request rate is then increased, maintained, or reduced. This novel system has been implemented using a combination of Elastic Timer, Metrics Listener, and Decision Log Writer, all of which are built into the JMeter test plan. There is therefore no requirement to incorporate external monitoring solutions or

make changes to the system under test. All workload control and controller decisions can be made during test execution.

Compared to existing tools, LoadGuard-JMeter provides a new technique to control the rate of requests sent to the system under test. FFA [11] and TSTCTG [12][13] produce a workload based on a predefined schedule. These tools do not take into account the SLOs when adjusting the target rate. DYNAMO [14] and DYNAMOJM [15] allow workloads to be adapted. Other tools such as RELOAD [16], SaFReL [17] and PerfScout [18] have varying techniques to adapt their workloads. LoadGuard-JMeter uses a novel approach by taking into account the p95 response time and error rates to assess the status of the SLOs. It will split its operation into three areas: SAFE, HOLD and VIOLATION. Depending on which area it falls into, the Elastic Timer will adapt to the target rate of requests. The Metrics Listener and Decision Log Writer record the metrics and decisions made by the controller. This provides a simple solution to adapt the rate of requests sent to the system based on the SLOs. This can be easily added into a standard JMeter test plan.

Related Work:

Workload selection is a central issue in performance testing. Performance tests must be run with enough workload to cause the system to behave as it would under real-world conditions but without causing the system to become unstable. This is often done in JMeter by setting the workload to be applied.

JMeter provides many options to provide this workload. A Constant Throughput Timer [22] can be set to provide a constant throughput. A Throughput Shaping Timer [12] can be used to provide a specific number of requests per second during a timeframe. This can be combined with a Concurrency Thread Group [13] to ensure there is always enough users active to meet the desired workload. An AutoStop [23] Listener can be used as a failsafe to stop the test once a certain level of latency, response time or error rate has been exceeded. The Free-Form Arrivals (FFA) Thread Group [11] allows requests to be generated based on a pre-defined arrival pattern. This can be used to create an arrival-based workload; however, the arrival rate will not change depending on the response times, or error rates, which are experienced whilst running.

Although these solutions can be used to provide a workload, they all fall into two categories. Either they are set to a specific schedule, or after a violation occurs the test will stop. None of them actively keep the request rate low by using the p95 response time and error-rate SLO.

There have been investigations into workload adaptation in the literature. DYNAMO [14] adapts the workload during the test to expose any performance issues caused by the workload. DYNAMOJM [15] takes this a step further to make workload adaptation more similar to JMeter. RELOAD [16], SaFReL [17] and PerfScout [18] use reinforcement learning or adaptive policies to explore the workload and find performance bottlenecks. Whilst these solutions show adaptive performance testing can be useful, they are all general frameworks or learning based solutions. They are also not a simple Timer and Listener solution like this one which can easily be applied to a regular JMeter test plan. The benefits of adaptive performance testing have been demonstrated by previous research. Tools such as DYNAMO [14], and DYNAMOJM [15] show how workload-based adaptation can be implemented during performance testing. SaFReL [17] uses fuzzy reinforcement learning to create performance test cases, which has led to increased efficiency in testing. PerfScout [18] is another tool that uses reinforcement learning to find performance failure points. This tool was able to achieve an accuracy of over 82% in finding performance failure points. Along with an increase in testing efficiency of up to 90%. However, these tools have different goals and use different adaptation techniques. Most focus on exploring performance boundaries, learning testing

policy and finding failure points. Instead, it uses a simple feedback-based pacing mechanism to ensure that configured p95 response-time and error-rate SLOs are maintained.

SLO-aware runtime control has also been applied in other contexts. Hanada and Ishibashi used SLO information to adjust timeout behavior in microservice communication [19]. This is related in principle, but it controls service-side timeout behavior rather than the workload applied by a performance testing tool. Recent AI-assisted JMeter tools, such as JMeter MCP Server, help execute tests, parse results, generate dashboards, and support analysis [20]. However, they do not act as in-test controllers that adjust request pacing while the JMeter run is still active.

Research Gap:

Existing JMeter workload mechanisms are useful but remain mostly open-loop or reactive. Free-Form Arrivals (FFA) [11] represents an open-loop arrival-based baseline: the configured request rate is pushed regardless of the live SLO condition. FFA scheme send requests based on a pre-defined schedule without considering any current p95 response times or error rates. This means that the tool will still send an excessive amount of load to the system, despite high latency or SLO breaches. LoadGuard-JMeter uses feedback from real-time SLOs to dynamically increase, maintain or decrease the request rate based on the current conditions the system is under. Throughput Shaping Timer [12] with Concurrency Thread Group (TSTCTG) [13] represents a common shaped-throughput baseline, where the tester defines an RPS schedule and provides enough concurrent users to pursue it. These baselines reflect practical JMeter ways of producing controlled workloads. However, these do not control or limit load when p95 latency or error rate gets close to a dangerous area. AutoStop Listener [23] can be used to stop the test if thresholds are breached but it does not control the pacing.

There are many tools available for research such as DYNAMO [14], DYNAMOJM [15], RELOAD [16], SaFReL [17] and PerfScout [18]. These demonstrate that adaptive workloads can be created. However, these approaches mainly rely on external frameworks, autonomous testing mechanisms, or learning-based methods rather than a lightweight SLO-aware request-pacing plugin. A simple JMeter plugin is therefore required that monitors latency and error data on an interval basis to calculate what the current SLO usage is, determine what risk zone it is in and apply an increase, steady or decrease action. LoadGuard-JMeter provides this by controlling the pacing of requests at runtime.

LoadGuard-JMeter can be distinguished by the way workload control is achieved. FFA [11] and TSTCTG [12][13] are deterministic workload generators. They do not use real time SLO feedback in order to control the request rate. While DYNAMO [14], DYNAMOJM [15], SaFReL [17] and PerfScout [18] all aim to provide some form of workload adaptation, No existing approach explicitly claims to control the system using interval-based feedback from both p95 response times and error-rate SLOs within SAFE, HOLD, and VIOLATION risk zones. This is achieved in LoadGuard-JMeter using a simple plugin for JMeter which will monitor the metrics and control the request rate accordingly by increasing, keeping steady or reducing it.

Material and Methods:

Methodological Overview:

The methodology used to design, build and evaluate LoadGuard-JMeter is depicted in Figure 1.

It started by identifying issues with existing static workload-generation methods. From there, an SLO-aware closed-loop system was developed to dynamically adjust the request rate. This was built using Apache JMeter with new components, including the Elastic Timer, Metrics Listener, SLO Feedback Controller and Decision Log Writer. These components work together to adjust the request rate based on p95 response-time and error-rate SLO feedback.

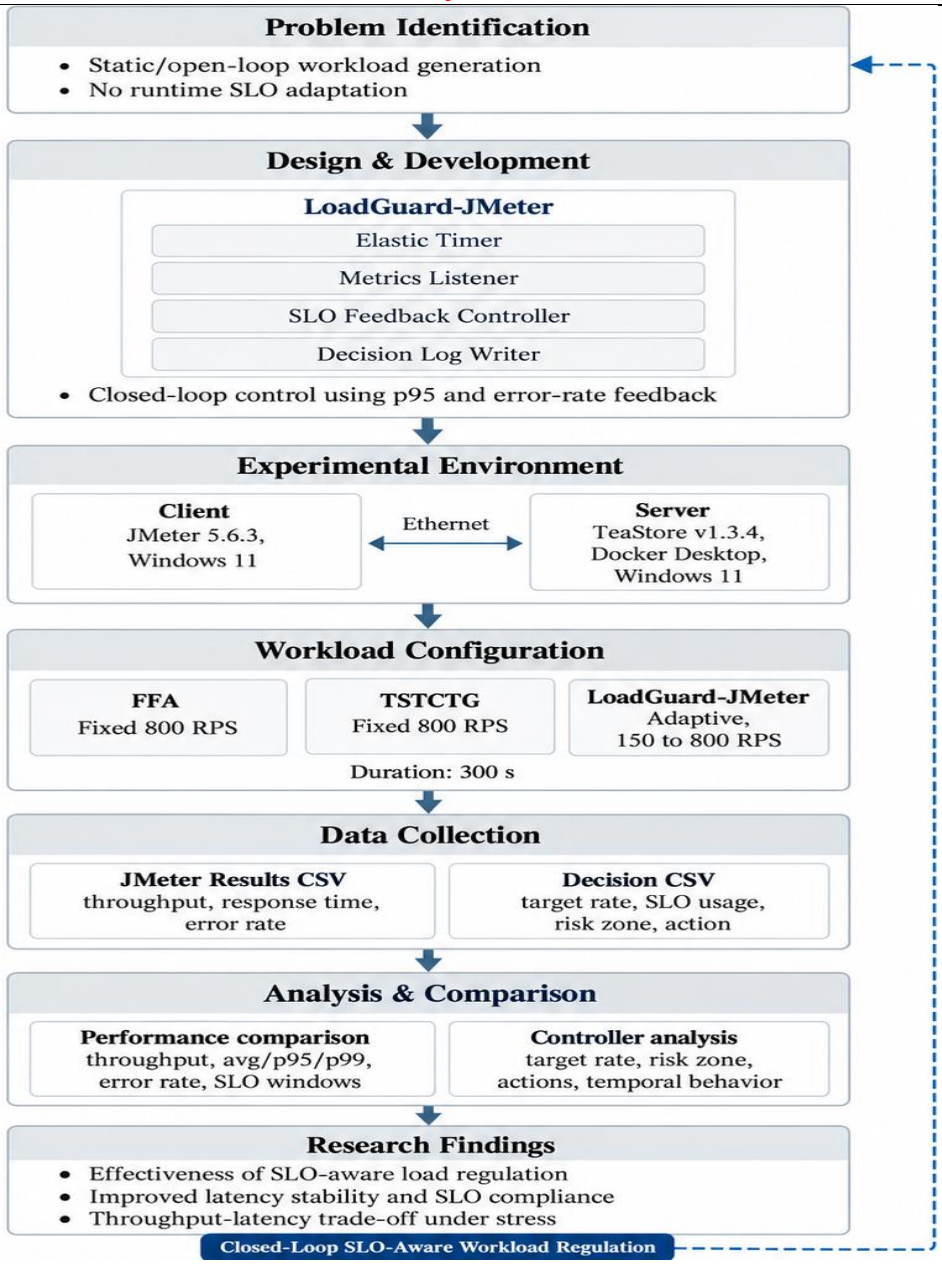


Figure 1. Methodological workflow of this research, from problem identification to research findings.

Once the tool was built, a two-node test environment was set up using two machines connected through Ethernet. Apache JMeter v5.6.3 was run on the client side, while the TeaStore Microservice Benchmark v1.3.4 was deployed on the server side using Docker Desktop. LoadGuard-JMeter was compared with Free-Form Arrivals (FFA) and Throughput Shaping Timer with Concurrency Thread Group (TSTCTG). All three schemes were tested with an 800 RPS stress load.

To ensure the results were consistent, five separate tests were run for each scheme using the same setup. Each test lasted 300 seconds, providing individual data on the performance of each scheme. The data were analyzed to determine the mean, standard deviation, and 95% confidence interval for each metric (throughput, response times, error rate and percentage of SLO violations for p95).

The LoadGuard-JMeter tool generated a separate CSV file that outlined what decision was made. This file included information on the desired request rate, how much of the SLO

was used, the risk zone and actions of the controller. The three schemes were analyzed to see how they performed in terms of throughput, response times, error rates and how often they entered violation periods of the SLO. LoadGuards decisions were analyzed to gain an understanding of how the controller behaved. This information was used to determine how well LoadGuard controlled the load during stress periods and the trade-off between throughput and response time. Statistical methods were also used in the Comparative Analysis section (Figure 1) of the experiment. This included using mean, standard deviation and 95% confidence intervals to analyze the five independent experimental runs.

Experimental Setup:

For the experiments, two separate machines were used. Both had Intel Core i7 processors, 16 GB RAM, Windows 11 OS, and JDK 17 installed. The processor had 20 cores (8 Performance-cores and 12 Efficient-cores), 28 threads, a maximum turbo speed of 5.40 GHz, 33 MB Intel Smart Cache, and Intel UHD Graphics 770. The two machines were connected directly using an Ethernet cable, with one working as the client and the other as the server.

The first machine was used as the client to generate load. Apache JMeter version 5.6.3 was used for this purpose. The LoadGuard-JMeter Plugin has been integrated into JMeter. The plugin contains an Elastic Timer to manage the request rate and a Metrics Listener to collect runtime metrics. The controller changes the target request rate based on the p95 response time and error-rate SLO usage. The FFA [11] and TSTCTG [12][13] schemes were also executed on the same client machine.

The second machine acted as an application server running the TeaStore microservice benchmark tool (version: 1.3.4). Docker Desktop was used to deploy TeaStore, meaning all microservices and features would have been run within a container. JMeter would make HTTP requests to TeaStore via the Ethernet connection. This arrangement placed the load generator and application on separate machines, preventing their CPU and memory usage from directly competing for the same resources.

Whilst JMeter was running all the results were saved onto the client machine. The standard results showed details of each request including response time and whether it had been successful or not. This data was used to find out the throughput, average response time, p95/p99 response time, max response time and error rate. LoadGuard-JMeter created an extra file called decisions.csv which showed what the target request rate was, what the actual throughput was, how much of the SLO was being used, what risk zone it was in and what the controller did (INCREASE/HOLD/DECREASE). This was useful for seeing how the controller was behaving.

All three schemes used the same client/server setup, TeaStore deployment, Ethernet connection, and measurement methodology. They were compared using the overall throughput, successful throughput, response time, error rate and how many windows the p95 SLO was breached. LoadGuard-JMeter used the decisions.csv file to see how the target request rate changed based on the SLO feedback during runtime. Data was analyzed using 5 second windows.

FFA [11] and TSTCTG [12][13] were chosen as they are two practical methods of creating high controlled loads in JMeter. FFA represents fixed open-loop arrival-based load generation, while TSTCTG represents plugin-based shaped throughput control with a Concurrency Thread Group. Neither baseline changes the target rate according to live SLO feedback. LoadGuard-JMeter, in contrast, changes the target rate during the run through INCREASE, HOLD, and DECREASE actions.

The experiment was performed using separate client and server machines to avoid direct interference. During the experiment, no metrics were collected for CPU, memory, network, or TeaStore container resource usage.

System Architecture and Design:

The LoadGuard-JMeter component is an additional lightweight component that can be used within a standard Apache JMeter setup. This component does not replace the Thread Group, Sampler, or Results Listeners included with JMeter; instead, it monitors completed requests at predefined intervals. These are then used to calculate the current state of the SLO and adjust the rate of requests being made. Figure 2 presents the overall architecture and runtime control flow of LoadGuard-JMeter.

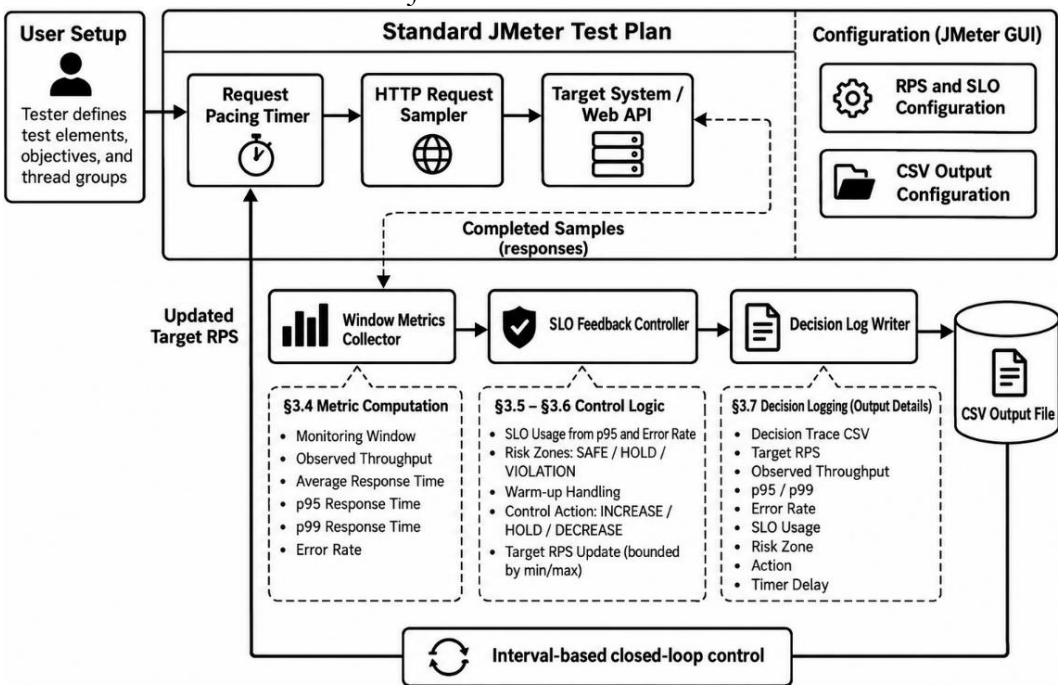


Figure 2. The system is an extension of the Apache JMeter tool that adds request rate control, runtime data collection, SLO aware feedback control and decision logging.

The architecture follows a feedback control model. JMeter is used to send requests to the system under test, completed samples are then collected and metrics calculated per interval. The current SLO status is then determined and used by the controller to adjust the target request rate for the next interval.

A Thread Group and HTTP Request sampler are created as usual. Under the Thread Group the LoadGuard-JMeter plugin is added with a pacing component and measurement component. The tester configures the initial RPS, minimum RPS, maximum RPS, p95 response-time SLO, error-rate SLO, warm-up duration, HOLD margin, increase/decrease percentages, and CSV output path.

The general structure of LoadGuard-JMeter includes all aspects of its runtime execution: performance measurements, SLO-based decisions, pacing decisions, and decision logging. In addition, it demonstrates how the general components of the system design are used throughout the execution. Including calculating monitoring windows, using SLOs, risk zone classification, making pacing decisions, and recording decisions made in CSV format.

Integration with JMeter Execution Flow:

LoadGuard-JMeter operates within the normal JMeter execution flow. Before sending a sampler request, the Request Pacing Timer reads the current target RPS and converts it into a delayed value. This delay controls the rate at which the active JMeter threads send requests. After completing the sampler request, the Window Metrics Collector receives the sample and records its response time and success status in the current monitoring window.

The plugin controls the workload by changing request pacing instead of changing the number of JMeter threads during test execution. Hence, the Thread Group remains stable

while the effective request rate can be increased or decreased from one monitoring interval to another.

Let P_k represent the target request rate during interval k, and let N_k represent the number of active JMeter threads.

$$D_k = \text{round}\left(\frac{N_k \times 1000}{P_k}\right) \quad (1)$$

where $D_k$is the request delay in milliseconds? When $P_k$increases, the delay becomes smaller. When $P_k$decreases, the delay becomes larger.

Functional Runtime Components:

LoadGuard-JMeter has 4 runtime components and 2 configuration components. The Request Pacing Timer will add a delay before executing the sample request at the most recent target RPS. The Window Metrics Collector takes the completed samples and organizes them into monitoring windows. The SLO Feedback Controller will store the current target RPS, upper/lower RPS limit, SLO values, warm up, risk zone and most recent action. It then uses the window metrics to decide whether the target request rate for the next interval should be increased, maintained, or reduced. The Decision Log Writer will record one line per monitoring window.

Configuration Components:

These components will allow user defined values to be set. The RPS and SLO Configuration will contain the starting RPS, minimum RPS, maximum RPS, p95 SLO, error rate SLO, warm up time, HOLD margin and raise/lower percentage values. The CSV Output Configuration will determine if decision logging is enabled and where to save the output file. This allows LoadGuard-JMeter to be used within a normal JMeter test plan whilst retaining the feedback control logic.

These components all work together. The Elastic Timer takes the current RPS Target from the SLO Feedback Controller and calculates how long to pause before each request. The Metrics Listener takes the response time and success from each request. These are then grouped into monitoring windows and fed back to the controller, which calculates SLO usage and the applicable risk zone. The controller will then update the RPS Target depending on what risk zone we are currently in. This will then be used by the Elastic Timer for the next requests. The Decision Log Writer logs the calculated metrics, RPS Target, Risk Zone and what the controller has done in a CSV file. This means all parts have access to the current request rate and metric data at an interval level.

Monitoring Window and Metric Computation:

LoadGuard-JMeter evaluates runtime behavior in monitoring windows. Let $W_k$be the set of samples completed during interval k, and let $n_k = |\ W_k\ |$be the number of completed samples in that interval. For each sample i, $r_i$denotes the response time, and $s_i$denotes the success status, where $s_i = 1$represents success and $s_i = 0$represents failure.

The average response time is computed as:

$$R_{\text{avg},k} = \frac{1}{n_k}\sum_{i=1}^{n_k} r_i \quad (2)$$

The observed throughput is computed as:

$$T_k = \frac{n_k}{\Delta t_k} \quad (3)$$

where $\Delta t_k$is the actual duration of the monitoring window in seconds

The number of failed requests is computed as:

$$F_k = \sum_{i=1}^{n_k} (1 - s_i) \qquad (4)$$

The error rate is then computed as:

$$E_k = \frac{F_k}{n_k} \times 100 \qquad (5)$$

The Window Metrics Collector also computes the 95th and 99th percentile response times, denoted as $R_{95,k}$ and $R_{99,k}$. These values are used because tail latency gives a clearer view of SLO behavior than the average response time alone.

The error rate represents the percentage of samples that resulted in errors. The p95 SLO-violation rate represents the percentage of monitoring windows in which the p95 response time exceeded the defined threshold (100 ms). An error rate is a different metric from the SLO-violation rate. The response-time percentiles are calculated from the response-time values of the collected samples. If these samples include errors, then they will be included in the calculation of the p95 and p99 values.

SLO Usage and Risk Classification:

LoadGuard-JMeter uses p95 response time and error rate as the main SLO indicators. Let θ_R be the configured p95 response-time SLO and let θ_E be the configured error-rate SLO. The response-time SLO usage is computed as:

$$U_{R,k} = \frac{R_{95,k}}{\theta_R} \times 100 \qquad (6)$$

The error-rate SLO usage is computed as:

$$U_{E,k} = \frac{E_k}{\theta_E} \times 100 \qquad (7)$$

The final SLO usage is the larger of the two values:

$$U_k = \max(U_{R,k}, U_{E,k}) \qquad (8)$$

This means that the controller reacts to the worst SLO condition in the current interval. If latency is within the SLO but the error rate is high, the error-rate usage controls the decision. If the error rate is low but p95 latency is high, the response-time usage controls the decision.

Let M be the HOLD margin percentage. The risk zone is assigned as:

$$Z_k = \begin{cases} \text{SAFE_ZONE}, & U_k < 100 - M \\ \text{HOLD_ZONE}, & 100 - M \le U_k \le 100 \\ \text{VIOLATION_ZONE}, & U_k > 100 \end{cases} \qquad (9)$$

For example, if the p95 SLO is 100 ms and the HOLD margin is 30%, then SLO usage below 70% is treated as safe. SLO usage from 70% to 100% is treated as the HOLD zone. SLO usage above 100% is treated as violation.

Closed-Loop Pacing Policy:

The **SLO Feedback Controller** updates the target request rate according to the current risk zone. Let P_k be the current target RPS, $P_{\min}$ the minimum allowed RPS, $P_{\max}$ the maximum allowed RPS, α the increase percentage, and β the decrease percentage.

During warm-up, metrics are recorded but the target RPS is not changed. After warm-up, the target RPS is updated as:

$$P_{k+1} = \begin{cases} P_k, & \text{warm-up is active} \\ \min\left(P_{\max}, P_k\left(1 + \frac{\alpha}{100}\right)\right), & Z_k = \text{SAFE_ZONE} \\ P_k, & Z_k = \text{HOLD_ZONE} \\ \max\left(P_{\min}, P_k\left(1 - \frac{\beta}{100}\right)\right), & Z_k = \text{VIOLATION_ZONE} \end{cases} \quad (10)$$

The corresponding control action is:

$$A_k = \begin{cases} \text{WARMUP}, & \text{warm-up is active} \\ \text{INCREASE}, & Z_k = \text{SAFE_ZONE} \\ \text{HOLD}, & Z_k = \text{HOLD_ZONE} \\ \text{DECREASE}, & Z_k = \text{VIOLATION_ZONE} \end{cases} \quad (11)$$

This policy is intentionally kept simple. It does not require training data, model files, or any external monitoring service. The tester only defines the SLO boundary and RPS limits, while the controller responds to the observed system behavior in each monitoring window.

The parameters and variables used in Equations (1)–(11) are defined in the accompanying text. The main parameters of the controller are presented in Table 1. Equations (1)–(11) define request pacing, monitoring metrics, SLO metrics, and the controller's decision policy. When there are no completed samples in a monitoring window, the throughput is zero. There will also be no response time or error rate data in this case. The monitoring window should therefore be excluded from SLO-based controller decisions.

Decision Trace and Output:

While JMeter provides default listeners that can display response times, throughput and error data, they do not explain how the runtime controller has changed the RPS of the target system during testing. LoadGuard-JMeter maintains a separate decision trace using the Decision Log Writer.

Each CSV entry corresponds to one monitoring window. It records the elapsed time, timestamp, window size, old target RPS, new target RPS, current target RPS, observed throughput, active threads, sample count, average response time, p95 response time, p99 response time, minimum response time, maximum response time, error count, error rate, configured SLO values, SLO usage, control state, risk zone, selected action, and timer delay.

The CSV Output Configuration specifies the path where this file is stored. After test execution, the tester can use this file to reconstruct the behavior of the controller. It can also be used to plot the target RPS, observed throughput, p95 latency, error rate, SLO usage, and controller actions over time.

Algorithm:

Algorithm 1 details the SLO-aware closed-loop pacing algorithm used in LoadGuard-JMeter. The algorithm begins by using the specified pacing rate to introduce delays between requests through the Elastic Timer. During this time, we monitor the system and log the response times and success rates of our requests over defined windows of time. Once the warm-up period is complete, the p95 response-time and error-rate SLO usage values are calculated. We then classify the current state of the system into either SAFE_ZONE, HOLD_ZONE or VIOLATION_ZONE. Depending on the state, the pacing rate is increased, maintained, or reduced within the defined minimum and maximum limits. We log the metrics at an interval level using the Metrics Listener and Decision Log Writer. This allows us to dynamically control the request pacing throughout the test.

Algorithm 1: SLO-aware closed-loop request pacing
Input: Initial target RPS P_0, minimum RPS P_min, maximum RPS P_max, p95 response-time SLO θ_R, error-rate SLO θ_E, increase percentage α, decrease percentage β, HOLD margin M, monitoring interval Δt, warm-up duration W

Output: Updated target RPS and interval-level decision trace
Set the current target request rate to P_0.
Start the JMeter Thread Group.
Before each sampler request, the Request Pacing Timer reads the current target RPS.
Compute the request delay using Equation (1).
Apply the delay and execute the sampler request.
After the request is completed, the Window Metrics Collector stores the response time and success status.
Continue collecting samples until the monitoring interval is complete.
At the end of the interval, compute average response time, observed throughput, error rate, p95 response time, and p99 response time using Equations (2)–(5).
If elapsed time is less than the warm-up duration, set the action to WARMUP and keep the target RPS unchanged.
If warm-up is complete, compute response-time SLO usage using Equation (6).
Compute error-rate SLO usage using Equation (7).
Compute final SLO usage using Equation (8).
Assign the risk zone using Equation (9).
If the interval is in SAFE_ZONE, set the action to INCREASE and update the target RPS using Equation (10).
If the interval is in HOLD_ZONE, set the action to HOLD and keep the target RPS unchanged using Equation (10).
If the interval is in VIOLATION_ZONE, set the action to DECREASE and update the target RPS using Equation (10).
The Decision Log Writer records the interval metrics, SLO usage, risk zone, action, old target RPS, and new target RPS.
The updated target RPS is used by the Request Pacing Timer in the next interval.
Repeat Steps 3–18 until the JMeter test ends.

Runtime Parameters:

The main runtime parameters of LoadGuard-JMeter are summarized in Table 1.

Table 1. Runtime parameters of LoadGuard-JMeter.

Parameter	Symbol	Purpose
Initial target RPS	P_0	Starting request rate used when the test begins
Minimum target RPS	$P_{\min}$	Lower bound used during DECREASE actions
Maximum target RPS	$P_{\max}$	Upper bound used during INCREASE actions
Monitoring interval	Δt	Time window used for metric computation and control decisions
Warm-up duration	W	Initial period during which metrics are recorded but target RPS is not changed
p95 response-time SLO	θ_R	Latency boundary used for response-time SLO usage
Error-rate SLO	θ_E	Error boundary used for error-rate SLO usage
Increase percentage	α	Percentage used to raise target RPS in the safe zone
Decrease percentage	β	Percentage used to reduce target RPS in the violation zone
HOLD margin	M	Margin below the SLO boundary where target RPS is held
CSV output path	—	File path used to store interval-level metrics and decisions

Results and Discussion:

The results are presented in two stages. First, the comparative analysis of each scheme's representative experimental run is given. This demonstrates how LoadGuard-JMeter performed in comparison with each scheme and illustrates the runtime characteristics during

the 800 RPS stress scenario. Next, five separate experimental runs of each scheme are detailed with statistical analysis to show how consistent and reliable the performance differences were.

The results are presented and discussed by evaluating throughput, response time, error rate, SLO violations, and the LoadGuard-JMeter controller's behavior during execution. The experiment was conducted under the 800 RPS stress scenario using the TeaStore benchmark. The aim was to determine whether LoadGuard-JMeter can pace requests when approaching or exceeding the set SLO thresholds.

The results were compared against FFA and TSTCTG [12][13] that were set to produce the workload without adjusting the target rates based on the SLOs during runtime.

The generated workload and the responses received were evaluated. This is important due to some scenarios generating a high number of requests; however, they may not complete in an acceptable time limit. Other scenarios may generate fewer requests but experience lower latency and fewer errors. The results will therefore be explored by looking at the throughput, successful throughput, average response time, p95/p99, maximum response time, error rate and SLO violation windows. The decisions made internally by the LoadGuard controller will also be examined to understand how the request rate was adjusted.

The aim of this evaluation is to show that LoadGuard-JMeter can perform SLO-safe load regulation. This does not necessarily mean that it will always generate the highest throughput, it means that once LoadGuard has determined that the offered load is unsafe, it can dynamically control the rate of requests being generated to improve tail-latency, prevent error propagation and maintain a controlled state within the test. LoadGuard-JMeter was compared with two established JMeter load-generation tools: Free-Form Arrivals Thread Group and Throughput Shaping Timer with Concurrency Thread Group. The aim was to configure these two tools to apply a constant 800 RPS, whereas LoadGuard could dynamically control its target rate using real-time feedback from p95 response times and error rates.

All three workload generation approaches were tested using the same stress-testing parameters. A maximum target workload of 800 RPS was used, with a test duration of 300 s. The FFA and TSTCTG baseline configurations both ran at a constant workload of 800 RPS. The LoadGuard-JMeter configuration had an initial target rate of 150 RPS and subsequently adjusted its target rate according to the measured SLO-related metrics during runtime. Table 2 provides the full details of the configurations used for each approach, including concurrency settings, SLO thresholds, monitoring intervals, controller settings, and output files.

Table 2. Configuration of the 800 RPS stress test. Abbreviations used in the table columns are Target load (TL), Duration (D), concurrency limit (CL), SLO setting (SLOS), Output files (OF).

Scheme	TL	D	CL	SLOS	OF
FFA	800 arrivals/s	300 s	800	p95 SLO = 100 ms, error SLO = 5%	CSV
TSTCTG	800 RPS	300 s	800	p95 SLO = 100 ms, error SLO = 5%	CSV
LoadGuard	Initial 150 RPS, min 10, max 800	300 s	100	p95 SLO = 100 ms, error SLO = 5%, interval = 5 s, warm-up = 15 s, increase = 20%, decrease = 30%, HOLD margin = 30%	CSV

These parameters were chosen for testing the controller with the 800 RPS stress scenario. A starting rate of 150 RPS was chosen so that the initial load would not reach the maximum target rate at the beginning of the test. The 15 s warm-up period allows metrics to be collected before the controller makes decisions. The 5 s monitoring interval means that the controller checks the SLO regularly. The increase and decrease percentages were set to 20%

and 30% respectively. The HOLD margin was set to 30%, meaning that the controller will not increase the target rate when it approaches the SLO limit. The controller's response may vary depending on how these settings are configured. Having lower intervals and higher changes means that the rate will often change. Whereas having longer intervals and smaller changes could cause it to react slower to SLO violation. These values were used for this evaluation; however, they might not be optimal for all systems.

The configuration for concurrency varies between the schemes as they have different methods for workload generation. The concurrency for both FFA and TSTCTG was set to 800 threads for the constant 800 RPS stress workload. However, because LoadGuard-JMeter has a dynamic request rate controlled by the SLO feedback, it was set to a concurrency level of 100 threads. This means that the concurrency levels were not identical across all three schemes. However, the different concurrency should be taken into account when looking at throughput and response times.

The results given in Tables 3-5 have been obtained from representative runs from each of the experimental schemes. Table 6 shows the mean values and their standard deviation and 95% confidence intervals for five independent runs. Percentage reductions have been taken relative to the relevant FFA and TSTCTG values. The results are given separately as only a single run, and five runs have been performed.

Overall Throughput and Response-Time Performance:

The results given in Table 3 provide an overview of the performance of the three different workload-generation approaches. It can clearly be seen that there are large differences in the number of requests made, successful throughput, response time, and error rate of each solution.

Table 3. Overall comparison under 800 RPS stress. Abbreviations used in the table columns are Samples (S), Successful requests (SR), Avg TPS (ATPS), Successful TPS (STPS), Avg Response time (ART), Max Response time (MRT), Error rate (ER)

Scheme	S	SR	ATPS	STPS	ART	p95	p99	MRT	ER
FFA	222,293	222,293	740.98	740.98	1011.07 ms	1734 ms	2373 ms	5238 ms	0%
TSTCTG	194,718	46,006	649.06	153.35	753.50 ms	2097 ms	3261 ms	9498 ms	76.37%
LoadGuard	26,310	26,310	87.70	87.70	34.13 ms	91 ms	133 ms	212 ms	0%

The FFA scheme generated the largest number of samples, reaching a total of 222,293 requests over 300 seconds. As all requests were successful, both the average throughput and successful throughput were 740.98 requests/s. This was near the 800 RPS defined in the workload. However, there could be issues with the workload generation and execution. Nevertheless, the FFA scheme generated a high number of requests without considering whether the response-time SLO was violated.

Whilst the FFA scheme generated a high number of successful requests, the response times were severely affected. The average response time was 1011.07 ms, while the p95 and p99 response times reached 1734 ms and 2373 ms, respectively. The maximum response time was 5238ms. This was significantly worse than the defined p95 response time of 100ms. The p95 indicates that 95% of requests had a response time less than or equal to 1734 ms. However, 5% of the requests would have a response time greater than 1734ms. The p99 indicates that approximately 1% of requests experienced a response time greater than 2373 ms. Overall, although the scheme generated many successful requests, a substantial proportion experienced long response times.

Overall, the FFA results show that a high number of successful requests alone is insufficient to evaluate system performance under load. The error rate remained at 0% and

the response time SLO was greatly exceeded. Although the system successfully processed all requests, the time required to process them increased substantially. Because FFA uses a predefined arrival rate, the workload does not decrease when the system takes longer to respond. This can increase the amount of queued work as additional requests continue to arrive. This can be seen in the results.

The TSTCTG scheme was able to generate 194,718 samples. This provided an average throughput of 649.06 requests per second. However, only 46,006 requests were successful, providing a successful throughput of 153.35 requests per second. This caused an error rate of 76.37% meaning over 3/4 of the requests failed. This is a huge difference to the FFA scheme where there were no failed requests.

The results of the TSTCTG scheme indicate that we were unable to sustain the defined shaped workload to achieve success. Whilst there was a high number of samples, a large proportion of these samples failed. This means the average throughput of 649.06 requests per second doesn't represent how many requests were actually completed successfully. However, there was a successful throughput of 153.35 requests per second.

The response times were severely affected, with an average response time of 753.50ms. This caused the p95 and p99 response times to be 2097ms and 3261ms. The maximum response time was 9498ms. Whilst the average was lower than the FFA scheme, the p95, p99 and maximum were all higher. This means we can't rely on the average response time to represent how the system is performing under load. The average can be low whilst a small number of requests has a long delay. However, we must also consider that there was a high failure rate.

The results of the TSTCTG scheme show that using a defined throughput doesn't mean requests will be successfully completed with a reasonable response time. Whilst the scheme attempts to keep the defined workload by using the shaping and concurrency features, it doesn't consider the p95 response time or error rate in real time to reduce the target rate. This resulted in a high number of tail latency requests and a large number of failed requests.

LoadGuard-JMeter had a different outcome. It was able to successfully complete 26,310 requests over the 300 seconds. This gave a throughput of 87.70 requests/s and a successful throughput of 87.70 requests/s. This was a much lower throughput than the FFA and TSTCTG schemes. However, it wasn't designed to have the highest request rate, instead it aimed to control the target rate to stay within the defined SLO.

As the scheme has an initial target of 150 RPS, this will be adjusted depending on the p95 response time and error rate in real time. If the SLO is violated, the controller will lower the target rate. If the system is performing well and safely under the SLO, the controller will increase the target rate. This means that the total number of requests generated will be lower.

This resulted in much lower response times. The average was 34.13ms, with the p95 and p99 being 91ms and 133ms. The maximum response time was 212ms. This meant that the p95 remained below the defined 100ms SLO. However, the p99 indicates that some of the requests had a response time higher than the SLO. Overall, however, the tail latency was much lower than the baseline schemes.

Compared to the FFA scheme, the overall p95 response time dropped to 91ms from 1734ms. This is an improvement of around 94.75%. The p99 also improved from 2373ms to 133ms, an improvement of around 94.40%. The maximum response time also greatly improved from 5238ms to 212ms. Compared to the TSTCTG scheme, the p95 and p99 improved by around 95.66% and 95.92%. The maximum response time also greatly improved from 9498ms to 212ms.

However, we must consider how the throughput affects the response times. LoadGuard-JMeter used a much lower request rate than the baseline schemes, meaning that the lower latency should not be interpreted as evidence that it processed an equivalent

workload faster. Instead, the feedback control mechanism has decided to use a lower workload to provide much better response times.

From Table 3, we can see three different outcomes. FFA had high successful throughput but severe latency. TSTCTG had a high sample rate but also high tail latency and failure rate. LoadGuard-JMeter reduced the applied workload and achieved much lower response times with no failures, although this also reduced throughput.

Figure 3 shows the same trade-off. FFA achieved the highest throughput, while TSTCTG produced the highest p95 and p99 response times. LoadGuard-JMeter had much lower latency. FFA had a mean response time of 1011.07 ms and p95 of 1734 ms. TSTCTG had a mean of 753.50 ms, p95 of 2097 ms and p99 of 3261 ms. In comparison, LoadGuard-JMeter had a mean of 34.13 ms, p95 of 91 ms and p99 of 133 ms.

Overall, all three schemes generated load against the benchmark. However, LoadGuard-JMeter used SLO feedback to control the workload, resulting in lower latency and fewer failures at the cost of lower throughput.

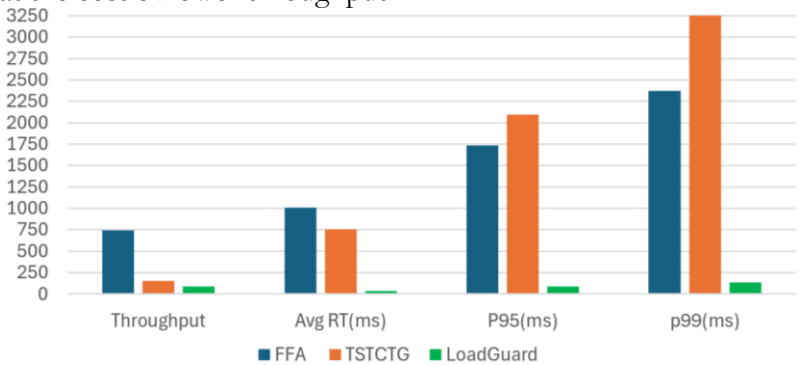


Figure 3. Comparison of FFA, TSTCTG, and LoadGuard-JMeter in terms of throughput, average response time, p95 response time, and p99 response time under the 800 RPS stress scenario

Window-Level p95 SLO Behavior:

The overall response-time results provide a general indication of the performance of each complete scheme. However, these results do not indicate how often each scheme violated the specified SLO. Although a scheme may have an acceptable overall response time, it may still experience short periods of high latency. Other schemes may consistently exceed the specified SLO. Therefore, the results were also analysed by dividing the data into 5-second intervals.

The 300-second duration of each scheme was divided into 60 intervals of 5 seconds each. For each interval, the p95 response time was calculated and compared with the specified 100 ms SLO. If the p95 response time for an interval exceeded the specified SLO, that interval was classified as a violation. The violation rate was calculated based on the number of violating intervals.

Table 4 presents the p95 SLO-violation results for each scheme. Both baseline schemes had violations for all intervals. FFA recorded 60 violations (100%), while TSTCTG also recorded 60 violations (100%). This means that neither baseline scheme returned to compliance with the specified p95 SLO during the 300-second test.

Table 4. Window-level p95 SLO behavior. Abbreviations used in the table columns are Total windows (TW), Violation windows (VW), Violation rate (VR), Mean window p95(MeanP95W), Max window p95 (MaxP95W)

Scheme	TW	VW	VR	MeanP95W	MaxP95W
FFA	60	60	100%	1729.13 ms	2072.35 ms
TSTCTG	60	60	100%	1878.52 ms	7089.55 ms
LoadGuard	60	13	21.67%	77.95 ms	164 ms

The average window p95 for FFA was 1729.13 ms, while the maximum window p95 was 2072.35 ms. Both values are substantially higher than the 100 ms SLO. The average window p95 was approximately 17.29 times the threshold. This shows that the high p95 values in the overall results were not limited to a few isolated occurrences. The p95 SLO was violated in every monitoring window throughout the run.

The max window p95 also gives information on how bad the violation was. A maximum value of 2072.35 ms indicates that at least one window had a p95 response time exceeding 2 seconds. The 100% violation rate indicates that the p95 SLO was exceeded in every monitoring window.

TSTCTG had a similar result to FFA but had more severe p95s. The average window p95 was 1878.52 ms and the max p95 was 7089.55 ms. Both values were higher than the corresponding values for FFA.

This high maximum indicates that at least one monitoring window had a very high p95 response time. Although the overall p95 was 2097 ms, the maximum window p95 indicates that some monitoring periods experienced even more severe latency. This is consistent with the high maximum response time and error rate reported in Table 3. However, the summary statistics do not provide further details about the causes of these high latencies.

LoadGuard-JMeter had very different results. Only 13 windows (21.67%) had a violation for the p95 SLO. This means that 47 windows (78.33%) remained within the configured p95 SLO. The average window p95 was 77.95 ms and max p95 was 164 ms.

The average window p95 was below the 100 ms threshold, consistent with the overall p95 response time of 91 ms reported in Table 3. However, there were still 13 violations showing that the controller was not able to completely avoid violations. The controller is reactive because its decisions are based on measurements of requests completed within the current monitoring window. Consequently, an SLO violation may occur before the controller reduces the request rate. This is an expected issue with using interval-based controllers.

The maximum window p95 was 164 ms, indicating that the threshold was exceeded in at least one monitoring window. However, the value was much lower than both FFA and TSTCTG. This suggests that the controller reduced the severity of p95 violations by lowering the request rate when the measured SLO usage entered the violation zone.

Compared with the two baselines, LoadGuard-JMeter reduced the p95 violation rate from 100% to 21.67%, a decrease of 78.33 percentage points. This meant that there were 47 less windows that had a violation. As each window lasted 5 seconds, the baselines had 300 seconds of violating windows, whereas LoadGuard-JMeter had 65 seconds. This duration represents the total length of monitoring windows classified as violations, not the exact time during which individual response times exceeded the SLO.

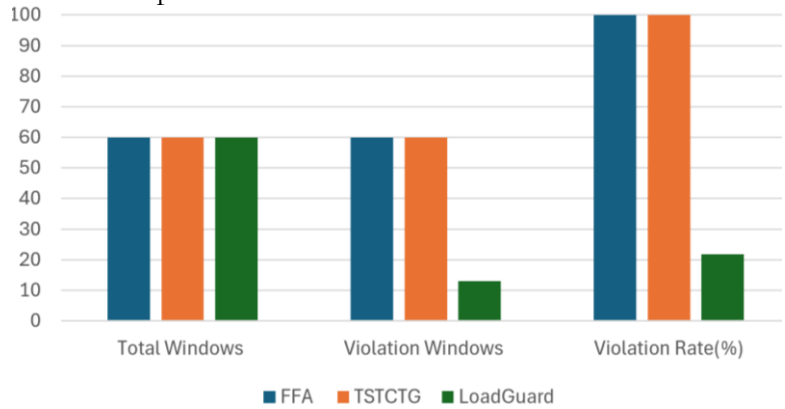


Figure 4. Comparison of total monitoring windows, p95 SLO-violation windows, and violation rates for FFA, TSTCTG, and LoadGuard-JMeter under the 800 RPS stress scenario.

The results shown in Table 4 provide further evidence. Table 3 shows that LoadGuard-JMeter had a much lower overall p95 response time. However, Table 4 also shows that the proposed scheme met the specified p95 SLO in most monitoring windows. This demonstrates an additional benefit: a reduction in the number of monitoring windows in which the p95 SLO was exceeded.

Figure 4 compares the total number of monitoring windows, the number of violating windows, and the violation rates of the three schemes. All schemes were evaluated over the same number of windows, allowing a direct comparison of the proportion of intervals in which the p95 SLO was exceeded.

FFA and TSTCTG both achieved a 100% violation rate. This is because the number of violating windows was equal to the total number of monitoring windows. In contrast, LoadGuard-JMeter had only 13 violating windows, accounting for 21.67% of the total run.

This provides an important insight into the proposed work. LoadGuard-JMeter aims to regulate request pacing using real-time SLO measurements. When these measurements indicate an unsafe state, the controller lowers the target request rate according to its configured policy. This is reflected in the reduced violation rate seen in Figure 4.

There are some interesting points seen from the figure. Firstly, LoadGuard-JMeter is not completely violating freely. As the controller runs after the measurement has been taken, there will always be some short-term violations. The results therefore provide evidence to show that the frequency of SLO violations has been reduced.

The baselines continued to have violations. This is due to the fact that FFA and TSTCTG have different ways of generating workloads. While FFA generates an arrival-based workload, TSTCTG uses a throughput-shaping method with concurrency control. However, neither method uses the p95 SLO to adjust the target rate. Both baselines had a 100% violation rate, meaning that neither maintained the specified latency requirement under the selected stress conditions.

The LoadGuard-JMeter results show that runtime feedback enables the controller to respond to latency measurements by lowering the target request rate. Therefore, the p95 response-time threshold was not exceeded in most monitoring windows. This indicates that implementing a closed-loop pacing system can reduce the duration of SLO violations during performance testing. However, this was achieved at the cost of reduced effective throughput.

LoadGuard-JMeter Controller Behavior:

The previous comparisons show the performance differences between LoadGuard-JMeter and the two baseline schemes. However, they do not explain how the proposed controller adjusted its target request rate during the experiment. To examine this behaviour, the LoadGuard-JMeter decision CSV file was analysed. The decision file recorded the target request rate, observed performance measurements, SLO usage, risk zone, and controller action for each monitoring interval.

Table 5. LoadGuard planned rate, risk zone, and control actions. Abbreviations used in the table columns are: Control action (CA), Windows (W), Target RPS range (TRPS), Avg target RPS (ARPS), Avg P95 (AP95), Max P95 (MP95)

Risk zone	CA	W	TRPS	ARPS	AP95	MP95
WARMUP_ZONE	WARMUP	2	150.00–150.00	150.00	47.50 ms	66 ms
SAFE_ZONE	INCREASE	26	52.28–166.37	97.36	51.96 ms	68 ms
HOLD_ZONE	HOLD	15	54.40–166.37	88.59	81.60 ms	95 ms
VIOLATION_ZONE	DECREASE	15	43.56–116.46	76.28	123.53 ms	164 ms

Table 5 presents the LoadGuard-JMeter controller results grouped according to the recorded risk zones and control actions. The controller had been set up with a p95 SLA of 100ms and an error rate SLA of 5%. In our configuration, the HOLD margin was set to 30%,

the increase percentage to 20%, and the decrease percentage to 30%. This meant that when SLO usage was below 70%, the controller was in the SAFE_ZONE. Between 70% and 100% usage, the controller was in the HOLD_ZONE, while values above 100% placed it in the VIOLATION_ZONE. Depending on the identified risk zone, the controller increased, maintained, or decreased the target request rate.

For the 300 second experiment, 60 5 second windows were used to analyze performance. However, there are only 58 decision points in the decision CSV because the monitoring intervals were of roughly equal length but not exactly 5 seconds. There are 2 WARMUP decisions in the 15 second warmup period. These take place at 5.011 seconds and 10.030 seconds. The next decision will take place at 15.151 seconds. Thus, there are 58 controller actions. This is shown in Table 5.

From the decision CSV we can see that all three decision types have been made (INCREASE, HOLD and DECREASE). This shows that the controller is not simply reducing the rate of requests throughout but is instead making decisions based on the system state found in the monitoring windows.

During the warm-up period, the controller kept the initial target request rate of 150 RPS. From the logs we can see that there were 2 monitoring windows during this period. The average target rate was also 150 RPS. The average p95 response time was 47.50ms with a maximum of 66ms which was below the set SLO of 100ms.

The controller was using this period to allow the benchmarking and workload generation environment to start. Whilst collecting performance data, the controller was not increasing or decreasing the target rate. As mentioned in the algorithm, this is due to the controller not making any decisions during the set warm-up period.

Once the warm-up period ended, the controller began making decisions based on the SLO. The SAFE_ZONE had the highest number of decisions (26). From the logs we can see that the average target rate was 97.36 RPS with a range of 52.28 – 166.37 RPS. The average p95 response time was 51.96ms with a maximum of 68ms.

This shows that the system was well within the p95 SLO. As the p95 response time was below the latency of 70ms (70% of SLO), the controller was able to make INCREASE decisions whilst keeping the error rate within the safe zone.

The controller would increase the target rate by 20%, up to the predefined maximum. This allows LoadGuard-JMeter to attempt to increase the workload. From the target range we can see that at various points during the experiment the controller was within the safe zone with varying target rates. This is due to the controller constantly trying to increase the target rate.

The HOLD_ZONE had 15 windows. The controller kept the target rate the same. The target rate range was 54.40 – 166.37 RPS with an average of 88.59 RPS. The average p95 was 81.60ms with a maximum of 95ms.

From this we can see that the hold decisions were made when the system was approaching the set SLO. With a HOLD margin of 30% and an SLO of 100ms on p95, the controller will hold the target rate if the latency is within 70-100% of SLO. This allows the test to continue without increasing the workload.

The HOLD decision is different to both INCREASE and DECREASE. The controller will not always try to increase throughput during every interval. It will also not decrease the request rate until a violation has been identified. Instead, the controller will keep the target rate the same. The average p95 response time was 81.60 ms, meaning that the p95 response time was below the specified SLO of 100ms but within the safer region.

The VIOLATION_ZONE had 15 DECREASE decisions. The target rate range was 43.56 – 116.46 RPS with an average of 76.28 RPS. The average p95 response time was 123.53ms with a maximum of 164ms.

This shows that the controller saw instances where the p95 was over the set SLO. When a violation was identified, the controller would decrease the target rate by 30%. This would reduce the workload going forward. The controller would also ensure that it would not go below the minimum RPS set.

We can also see that the maximum request rate of 800 RPS was not achieved. The maximum target rate seen was 166.37 RPS. This is due to the controller detected the SLO conditions and therefore decided to hold and decrease the target. Even though the controller was allowed to increase up to 800 RPS, it was not continuing to do so.

The maximum RPS is part of the controller and is not a workload that the controller will attempt to achieve. If the response time or error rate shows unsafe conditions, LoadGuard-JMeter will not attempt to keep the maximum rate. Instead, it will adapt based on what it sees.

Looking at the average target rates, we can see that the SAFE_ZONE had an average of 97.36 RPS, the HOLD_ZONE had an average of 88.59 RPS and the VIOLATION_ZONE had an average of 76.28 RPS. This should not be seen as the target rate having changed as the data is split into the risk zones. It does, however, show that different decisions were made based on the target rate.

This gives us more evidence that the feedback mechanism works. LoadGuard-JMeter will increase the planned workload if it sees that the system is safely within the SLO. If the controller sees that the conditions are approaching the boundary then it will stop increasing. If it sees a violation then it will decrease the workload. This is different to the comparison as the workload was pre-set.

Figure 5 shows that the controller has made the most INCREASE decisions (26). There were 15 HOLD decisions and 15 DECREASE decisions. There were also 2 warm-up decisions.

As there were more INCREASE decisions, this shows that the controller was trying to increase the workload as much as possible when within the safe zone. This is an important part of the mechanism as LoadGuard-JMeter does not just try to decrease the load. It will try to increase the request rate if the SLO usage shows that it can increase the workload safely.

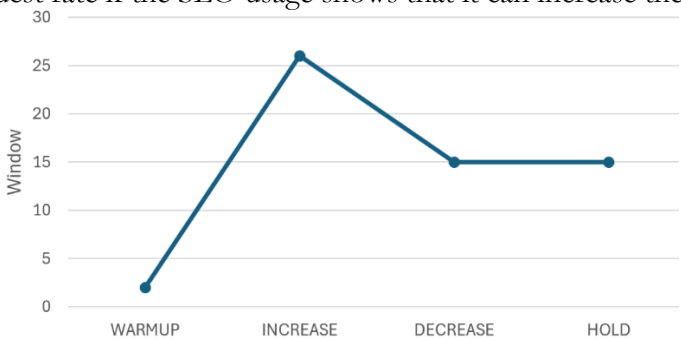


Figure 5. Distribution of LoadGuard-JMeter controller actions (WARMUP, INCREASE, DECREASE, and HOLD) across monitoring windows during the 800 RPS stress test.

The 15 HOLD actions show that the controller was often very close to the SLO limit. As a result, it stops increasing the target request rate. The target rate remains unchanged, preventing further increases. This is done because the response-time measurements are very close to the threshold.

The 15 DECREASE actions indicate that SLO violations occurred on several occasions. This is an important action to show that if the controller detects degraded performance, it decreases the target request rate. If there was no feedback mechanism in place then LoadGuard-JMeter would have continued to follow the previous target request rate.

The combination of INCREASE, HOLD, and DECREASE actions shows that the controller operated in different states during the same experiment. It was not restricted to a

single predefined workload level. Instead, the target request rate was adjusted according to the measured system response. Figure 5 therefore supports the controller-behavior results in Table 5 by showing the frequency of the different decisions.

However, Figure 5 represents the distribution of controller actions rather than their chronological order. It does not show exactly when the actions occurred or how many consecutive windows remained in each state. The decision CSV contains the interval-level records needed for such an analysis. The present figure is therefore useful for showing the occurrence and frequency of the controller's decisions, while Table 5 provides the corresponding target-rate and p95 response-time measurements.

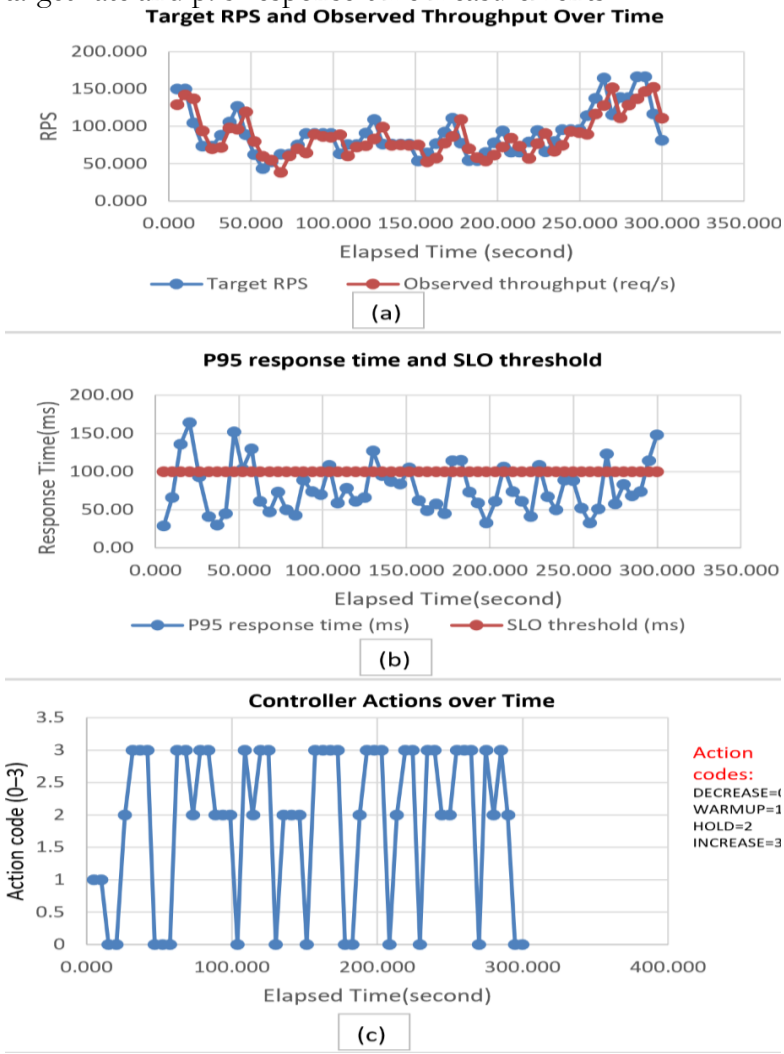


Figure 6. LoadGuard-JMeter's time series behavior in a 300 second stress test. (a) Expected request rate and actual throughput; (b) p95 response times and their relation to the configured SLO of 100 ms; (c) Controller actions (DECREASE=0, WARMUP=1, HOLD=2, INCREASE=3).

Figure 6 provides an overview of the time series characteristics of LoadGuard-JMeter over the course of the 300 second stress test.

Figure 6 (a) illustrates the relationship between target RPS and actual throughput and shows how the controller is dynamically changing the load. (b) shows how the p95 response time compares with the defined SLO of 100ms by displaying the instances where we exceed our SLO. (c) shows the sequence of steps taken by the controller when it is managing the load. It can be seen that the controller is operating on a 5 second interval, therefore there will be a lag between the change in system state and the next pacing adjustment made. This means that

the controller frequently switches between different states. The target RPS is fluctuating as there is no convergence towards a specific value, however we can see from the graph that there is bounded control of the load. Occasionally we exceed our SLO, and the control actions are being changed frequently to show that the controller does not remain fully stable around the SLO threshold.

Discussion of Throughput and SLO Compliance:

In summary, the results shown in Tables 3-5 demonstrate the primary trade-off associated with using a SLO aware workload generator. While FFA and TSTCTG aimed to apply the predefined stress workload, LoadGuard-JMeter used an adaptive target rate of requests. The static workload generators resulted in a higher number of requests being applied but had significantly worse performance in terms of response times. LoadGuard-JMeter resulted in fewer requests but had much better performance with significantly lower response times and fewer p95 SLO violations.

FFA had the highest successful throughput with 740.98 requests/sec. However, as there were 0% of errors, all requests applied resulted in a successful throughput. The p95 response time was very high with 1734ms compared to the defined 100ms SLO.

TSTCTG had a much lower successful throughput with only 153.35 requests/sec. Even though it had an average sample rate of 649.06 requests/sec, it had a high error rate of 76.37%. The p95 response time was very high at 2097ms, and all monitoring windows exhibited an SLO violation. This meant that not only were requests failing but the latency was also violated continuously.

LoadGuard-JMeter had a successful throughput of 87.70 requests/sec. There were no errors recorded and the general p95 response time was 91ms. 47 of the 60 monitoring windows were within the defined 100ms SLO. This demonstrates that the controller effectively decreased the workload, so the system was running in a much lower latency state for most of the tests.

These results demonstrate that there is a difference between aiming for maximum throughput or SLO adherence. Workload generators can apply a high number of requests but if the system cannot meet the defined response-time objective, this may result in poor performance. Whereas if using an SLO aware workload controller such as LoadGuard-JMeter the workload will be reduced to ensure acceptable response times are maintained.

There was a significant reduction in the throughput. LoadGuard-JMeter resulted in a successful throughput that was approximately 88.16% lower than FFA and 42.81% lower than TSTCTG. This shows that whilst the latency and SLO results have improved, the workload applied to the benchmark has been significantly reduced.

Therefore, LoadGuard-JMeter does not provide a higher maximum capacity than the two baselines. Instead, it provides a feedback mechanism that adapts the workload being applied based on the observed SLO. This means that the performance test can still be carried out but at a reduced request rate.

Looking at the results per window also helps to support this. Both static approaches resulted in all monitoring windows having an SLO violation for the p95 metric. LoadGuard-JMeter reduced this to 21.67%. Looking at the decisions made by the controller this was due to the runtime INCREASE, HOLD, and DECREASE actions. This means that the reduced number of violations were due to a workload control mechanism that adapted the target request rate during runtime.

Discussion of Closed Loop Control:

The results obtained using LoadGuard-JMeter show how a simple feedback control policy can be applied to the normal JMeter workflow. As previously mentioned, the controller uses the 2 SLO metrics of P95 Response Time and Error Rate to decide what state the system is currently in. The higher the SLO metric value, the higher the risk zone in which the system

is classified. Based on the decision, the controller will decide what rate to aim for in the next window.

From the decisions we can see that the controller has taken action based on what state the system was in. If the system was safe then the controller would increase the desired rate. When close to the threshold (but below), it will keep the current rate. Finally, if the threshold was exceeded then the desired rate will be decreased. This follows the controller policies outlined in the system design.

This is where the HOLD region becomes important. Without the HOLD threshold the simple controller may have caused the request rate to continually increase and at some point, cause the SLO to be exceeded. By having a HOLD region, the controller can stop the planned increase in load before actually reaching the threshold. From the results we can see there were 15 windows where the controller decided to HOLD. The average P95 response time here was 81.60ms meaning that the controller could identify safe intervals that were closer to the threshold.

The DECREASE action is the main way the controller will respond to the system being overloaded. If the SLO metric was above 100%, the desired rate will be decreased. As expected, we can see that the average P95 for the VIOLATION_ZONE is much higher than both the HOLD_ZONE and SAFE_ZONE with a value of 123.53ms.

When the INCREASE action is used, this is the reverse of the previous action. This means that the controller can increase the load based on the current metrics. This prevents the controller getting stuck at a low rate. From the results we can see that the INCREASE action was used the greatest number of times (26 windows). This shows how the controller continues to try to increase the desired rate whenever it observes a safe state.

As we can see all three actions have been used, which shows how the LoadGuard-JMeter has implemented feedback-based pacing. The controller has actively adjusted the desired rate based on the performance of the system. This is the main difference between the LoadGuard-JMeter and the defined workload generation methods.

However, the results also show that the controller did not actually reach the maximum desired rate (800 RPS). The rate range was much lower. This shows that the SLO threshold had limited the increase in load. The controller would have preferred to have a lower rate than to increase towards the max.

The results also depend on the parameters used. The HOLD threshold was set to 30%, so this meant that the controller would stop increasing the workload earlier. The rate would also increase by 20% and decrease by 30%. The monitoring period was also set to 5 seconds so this meant the controller could only react every 5 seconds. Depending on the parameters, the desired rate could behave differently and cause different tradeoffs between throughput and latency. This shows how the controller works with the parameters set but doesn't mean they would be the best option for all systems.

General Discussion and Research Implications:

From all three comparisons, LoadGuard-JMeter exhibited the expected functionality of an SLO aware feedback-based load generation mechanism. In the FFA approach the predefined load generation approach still resulted in a high rate of requests despite having a high number of response time SLO breaches. The FFA resulted in a high throughput of successful requests but also had a high latency throughout. The TSTCTG had a large number of failed requests and a high tail latency. The LoadGuard-JMeter reduced the workload and operated in a low latency state with no errors.

Overall, it is clear that response times should be considered when looking at throughput and error rates. Both the FFA and TSTCTG had a high throughput, but the FFA had a much higher p95 response time than the set SLO. The TSTCTG had a high throughput of samples, but the majority of requests failed.

The LoadGuard-JMeter resulted in fewer requests but had an overall p95 below the set threshold. Most monitoring windows achieved a p95 response time that was within the set SLO. A smaller number of breach windows were seen meaning the controller was helping to keep the system within a set response time for a greater proportion of the test. The decision CSV also showed that the rate was being adjusted to hit the desired rate using the INCREASE, HOLD, and DECREASE decisions.

This answers the research questions put forward by this study. Firstly, the interval-based metrics were used to control the rate of requests (RQ1). Secondly, the p95 response time and error rate metrics were mapped to the risk zones and controller decisions (RQ2). Thirdly, the comparison showed that the LoadGuard-JMeter had a substantially lower tail latency, error rate and number of p95 SLO breach windows compared to the TSTCTG (RQ3). Lastly, the throughput results showed that this was achieved by having a substantially lower effective rate of requests (RQ4).

The results should be interpreted based on the load and system configuration used in the experiment. Only one instance of TeaStore was used along with a 300 second stress scenario. The controller was also tuned to a particular set of values. This meant that the baseline had a fixed workload of 800 RPS and the LoadGuard-JMeter started at a lower rate of 150 RPS and dynamically changed the target rate. Therefore, each method was running at different throughputs. Further experiments with equivalent offered loads, different SLO thresholds and other load patterns will help to understand how generalizable these results are.

Overall, the evaluation has shown that LoadGuard-JMeter does achieve its main goal. The proposed plugin introduces an SLO aware feedback controller to Apache JMeter. This can increase, maintain, or reduce the rate of requests in the same test. The feedback controller helped to maintain a substantially lower response time whilst keeping no requests failing in the 800 RPS stress scenario. This was achieved by reducing the effective workload. Therefore, LoadGuard-JMeter should be seen as a workload controller for performance testing. This is due to the fact that maintaining near the set SLO boundaries is more important than applying the maximum possible rate of requests.

Although the three scenarios were executed in the same experimental environment, different effective request rates were used. FFA and TSTCTG were both set to generate 800 requests per second. However, LoadGuard-JMeter was configured to initially generate a smaller number of requests (150 RPS) and dynamically adjusted based on the feedback from the SLO. Therefore, it is not an advantage that LoadGuard-JMeter results in shorter response times. This investigation is mainly interested in comparing how the three scenarios perform in a common stress-testing scenario. For further investigation into the effect of workload reduction and the feedback controller, additional experiments need to be carried out using similar load levels.

The results of this study were obtained by using the TeaStore benchmark. The two-node configuration and the workload settings will affect the decisions made by the controller. This can result in different response times and throughputs depending on the workload configuration, application architecture, hardware resources and SLO thresholds. In practice, the LoadGuard-JMeter allows more flexible control when testing by allowing testers to consider their application's performance requirements. However, these improvements in performance have only been tested within this environment and cannot be assumed to apply to all applications.

Repeated-Run Validation and Statistical Analysis:

In the last section we looked at how LoadGuard-JMeter performed in comparison to FFA and TSTCTG during the 800 RPS stress test. Here we look at how consistent the performance was. For each scheme we ran 5 runs independently with the same client-server environment, using the TeaStore benchmark with the same workload configuration. We

performed five runs sequentially. These five runs were not randomized. Prior to executing each run, The TeaStore Docker environment and JVM process were reset. This ensured we began each run from a clean state to minimize any impact of the application's runtime state. The time taken for each run was 300 seconds. We gathered performance data by running JMeter and capturing the results in CSV format. We looked at the performance in terms of successful throughput, average response time, p95/p99 response times, error rate and p95 SLA violation. This was then used to calculate the mean, standard deviation (SD) and 95% confidence interval (CI) for each of the runs.

Table 6 shows that the overall differences in performance between the three schemes remain consistent when running multiple runs. FFA had the highest successful throughput. TSTCTG had high error rates and response times. LoadGuard-JMeter had low response times and SLA violations. However, it was running at a reduced request rate.

Table 6. Mean, SD and CI for performance across 5 runs.

Metric	FFA	TSTCTG	LoadGuard
Successful RPS	741.20 ± 8.35	154.10 ± 9.45	88.15 ± 4.20
95% CI	[730.83, 751.57]	[142.37, 165.83]	[82.94, 93.36]
Avg RT (ms)	1015.40 ± 28.60	758.30 ± 35.20	35.10 ± 3.25
95% CI	[979.89, 1050.91]	[714.60, 802.00]	[31.06, 39.14]
p95 (ms)	1742.60 ± 45.20	2108.40 ± 73.60	92.40 ± 5.80
95% CI	[1686.48, 1798.72]	[2017.02, 2199.78]	[85.20, 99.60]
p99 (ms)	2381.40 ± 62.50	3275.20 ± 95.40	135.20 ± 9.60
95% CI	[2303.80, 2459.00]	[3156.75, 3393.65]	[123.28, 147.12]
Error rate (%)	0.00 ± 0.00	76.10 ± 2.30	0.00 ± 0.00
95% CI	[0.00, 0.00]	[73.24, 78.96]	[0.00, 0.00]
p95 violations (%)	100.00 ± 0.00	100.00 ± 0.00	22.00 ± 3.33
95% CI	[100.00, 100.00]	[100.00, 100.00]	[17.87, 26.13]

Throughput Performance:

The results from the repeated runs showed that FFA had the highest average successful throughput at 741.20 with a standard deviation of 8.35 requests per second and a 95% confidence interval of [730.83, 751.57]. TSTCTG had an average throughput of 154.10 requests per second with a standard deviation of 9.45. LoadGuard-JMeter had an average throughput of 88.15 requests per second with a standard deviation of 4.20. The mean throughput had a 95% confidence interval of [82.94, 93.36] which was considerably lower than the throughput ranges of the baseline solutions. As seen in the single run evaluation, LoadGuard-JMeter had a lower successful throughput due to its controller reducing the desired rate of requests. The low standard deviation of 4.20 RPS shows that there was little variation in the successful throughput across the 5 runs. However, the lower throughput does confirm that the proposed mechanism prioritizes adhering to the SLOs over maintaining the highest workload.

Average Response Time Performance:

There was a substantial difference in the observed average response times between the schemes. FFA had an average response time of 1015.40 ms with a standard deviation of 28.60. TSTCTG had an average response time of 758.30 ms with a standard deviation of 35.20. LoadGuard-JMeter had an average response time of 35.10 ms with a standard deviation of 3.25. This had a 95% confidence interval of [31.06, 39.14]. This represented a decrease of approximately 96.54% compared to FFA and 95.37% compared to TSTCTG. The confidence intervals did not overlap, indicating clearly separated estimates of the mean response times across the five runs. LoadGuard-JMeter consistently had a much lower average response time

due to the dynamically controlled workload. However, this should be taken into consideration with the lower effective request rate maintained by the controller.

P95 and P99 Response Times:

The tail latency results also highlight the differences in performance. FFA had an average p95 response time of 1742.60 ms (±45.20). TSTCTG had an increased p95 response time of 2108.40 ms (±73.60). LoadGuard-JMeter had a much lower average p95 response time of 92.40 ms (±5.80) with a 95% confidence interval of [85.20, 99.60]. This is a reduction in the average p95 response time by around 94.70% and 95.62% compared to FFA and TSTCTG respectively.

Similar results can be seen for the p99 response times. FFA had a response time of 2381.40 ms (±62.50) and TSTCTG had a much higher 3275.20 ms (±95.40). LoadGuard-JMeter had a much lower p99 response time of 135.20 ms (±9.60) with a 95% confidence interval of [123.28, 147.12]. This is a reduction in the average p99 response time by approximately 94.32% and 95.87% when compared to FFA and TSTCTG. The low standard deviation of the p95 and p99 response times for LoadGuard-JMeter show that there was a small range of variations in the tail latencies for each of the five runs. The p95 confidence interval was below the set SLO of 100ms. However, this doesn't imply that all the monitoring windows met the SLO. Overall, the results can be seen to demonstrate that LoadGuard-JMeter had much lower tail latency times compared to the fixed load solutions.

Error Rates:

For error rates, neither FFA nor LoadGuard-JMeter produced any failures. This meant they had an average error rate of 0.00% (±0.00). TSTCTG had a high average error rate of 76.10% (±2.30) with a 95% confidence interval of [73.24, 78.96]. This shows that most of the requests failed when using TSTCTG. The low confidence interval demonstrates that the high error rate was consistently seen in each run. LoadGuard-JMeter had an error rate of 0% which shows that the reduced workload was still in a state where all requests were completed successfully. As with the previous metric, FFA had a 0% error rate. This shows that although the response times were high, no errors were experienced when completing requests.

P95 SLO Violations:

The results for SLO violations also confirm the previous observations. Both FFA and TSTCTG had an average SLO violation rate of 100.00% (±0.00). This means that all the monitoring windows did not meet the set SLO of 100ms for p95. LoadGuard-JMeter had an average of 22.00% (±3.33) with a 95% confidence interval of [17.87, 26.13]. This is a reduction in the average SLO violation rate of 78% when compared to both the baselines.

LoadGuard-JMeter has a similar result to its first evaluation. It had a violation rate of 21.67%. The standard deviation shows that there was some variation in the number of monitoring windows that violated the SLO across each of the five runs. However, the average violation rate was still considerably lower than the baselines. This shows that LoadGuard-JMeter was not able to eliminate any SLO violations but consistently reduced them when under stress. This aligns with the functionality of the controller. The target request rate will increase when in safe conditions, remain around the edge of the SLO and reduce when a violation is encountered.

Throughput and Latency Trade-Off:

Across the multiple runs, there is a clear throughput-latency trade-off. FFA produced the highest successful throughput. However, the average, p95 and p99 response times were still much higher than the set SLO. TSTCTG had a less successful throughput than FFA and had high response times and many failed requests. LoadGuard-JMeter had the lowest successful throughput. However, it also had much lower average and tail response times. No requests failed and there were fewer monitoring windows that violated the SLO.

The results clearly show that there was a large difference in each of the metrics between each scheme with clearly defined 95% confidence intervals. However, this just shows the level of uncertainty around the mean of each scheme. To determine if there was a significant difference in the average values, further statistical analysis would need to be performed.

The results from multiple runs also align with what can be seen in Table 5 for the LoadGuard-JMeter controller decisions. The target request rate increases when in safe conditions and remains around the edge of the SLO. If a violation is encountered then the target will be reduced. This means the workload can be adjusted based on the current operating condition. The results from the multiple runs show that similar latency and SLO trends were observed.

Overall, the results from the five runs confirm the findings from the initial comparison. LoadGuard-JMeter consistently had lower response times, 0 errors and fewer p95 SLO violations when compared to the fixed load baselines. It also ran at a much lower request rate. There is evidence to suggest that the performance will be consistent under these conditions. However, it should be noted that these latency improvements do not mean that LoadGuard-JMeter has improved performance at the same level of workload intensity. The different schemes were running at different effective request rates. The results support LoadGuard-JMeter as an SLO aware method for pacing requests. It ensures the workload is controlled to keep the system in a lower latency state during stress testing.

Statistical Analysis

Each workload generation mechanism was run 5 times independently with the same experimental setup and their results were analyzed. The mean, standard deviation (SD), and 95% confidence interval (CI) were calculated for the successful throughput, average response time, p95 and p99 response time, error rate and p95 SLO violation percentage.

For each performance metric, the mean was calculated as:

$$\bar{x} = \frac{x_1 + x_2 + x_3 + x_4 + x_5}{5}$$

where xi represents the measured value from the ith experimental run and (n=5) is the number of independent runs.

The sample standard deviation was calculated as:

$$SD = \sqrt{\frac{\sum_{i=1}^{5}(x_i - \bar{x})^2}{4}}$$

The 95% confidence interval was calculated using Student's t-distribution:

$$95\%\ CI = \bar{x} \pm 2.776 \times \frac{SD}{\sqrt{5}}$$

The degree of freedom for the 5 independent runs was (n-1=4). Therefore, the tcrit value was 2.776. Hence, confidence intervals were calculated using the mean and standard deviation of the 5 runs.

Looking at the statistical data in Table 6 we can see LoadGuard-JMeter had a mean p95 response time of 92.40 ± 5.80 ms (95% CI: [85.20, 99.60]). This is much lower than the baseline schemes which both had an average response time of 1742.60 ± 45.20 ms (FFA) and 2108.40 ± 73.60 ms (TSTCTG). In terms of p95 SLO violations, LoadGuard-JMeter had a mean violation rate of 22.00 ± 3.33%. This again is much lower than the baseline schemes which both had 100%. Finally, we can note that the confidence intervals of the key response time metrics do not overlap. This means there is a large difference between the two systems when considering the variability of the 5 runs.

The statistical results show that LoadGuard-JMeter has lower response times and less p95 SLO violations over the five test runs. The low standard deviation indicates there was not

much variation between the test runs. The 95% confidence intervals show the estimated range for the population mean. Although LoadGuard-JMeter had a lower level of throughput success, it had consistent response time and SLO violations. This shows that the proposed controller is effective at managing the workload to meet the requirements of the SLO.

The averages and standard deviations across the five independent runs of each scheme were calculated. 95% confidence intervals were computed using a student's t distribution with four degrees of freedom. These were used consistently throughout the evaluation of the primary performance measures as shown in Table 6. No hypothesis testing has been carried out between the different schemes; hence the confidence intervals that do not overlap do not reflect a statistical significance. The results have therefore been described based on the variation and difference seen in the performance of each scheme over five runs.

Requirements of LoadGuard-JMeter:

LoadGuard-JMeter was developed using Java/JDK 17. It was tested with Apache JMeter 5.6.3 and TeaStore microservice benchmark version 1.3.4. The current version of LoadGuard-JMeter is v1.2. See the Data Availability Statement for further information.

Conclusion:

This paper presents LoadGuard-JMeter, an SLO-aware feedback loop that uses p95 response time and error-rate SLOs to dynamically adjust JMeter's request rate. The plugin consists of Elastic Timer, Metrics Listener, SLO Feedback Controller and Decision Log Writer components. Depending on the current SLO state, the controller will classify the system as SAFE, HOLD or VIOLATION and increase, hold or decrease the request rate respectively.

The results of the evaluation using the TeaStore microservice benchmark demonstrate that LoadGuard-JMeter achieved a lower throughput than both Free-Form Arrivals (FFA) and Throughput Shaping Timer with Concurrency Thread Group (TSTCTG). The stress test used an 800 RPS load. FFA achieved the highest throughput of 740.98 requests/s. However, the p95 and p99 response times increased to 1734ms and 2373ms. TSTCTG achieved a throughput of 649.06 samples/s, however only 153.35 requests/s were successful. The error rate increased to 76.37%. The p95 and p99 response times were 2097 ms and 3261 ms, respectively. LoadGuard-JMeter achieved a throughput of 87.70 requests/s. However, the response times were kept low with p95 and p99 of 91ms and 133ms. The maximum response time was 212ms, with an error rate of 0%.

Neither of the baseline methods met the p95 SLO of 100ms. This was violated across all 60 windows. LoadGuard-JMeter violated the p95 SLO for only 13 windows. This is a violation rate of 21.67%. The decision CSV demonstrates that LoadGuard increased, held and decreased the request rate based on the current SLOs. This clearly demonstrates the trade-off between throughput and latency. LoadGuard isn't designed to achieve the highest possible throughput. Instead, it will reduce the request rate to keep the system within a lower latency state when unsafe conditions for the SLO's occur.

These results were further analyzed by running the schemes 5 times. From the statistical analysis it can be seen that LoadGuard-JMeter had a mean p95 response time of 92.40 ± 5.80ms and mean violation rate of 22.00 ± 3.33%. These are similar to what was found in the first experiment. The low standard deviation and 95% confidence interval show that the performance was consistent.

The evaluation was limited to the TeaStore benchmark, stress testing for 300 seconds with specific SLO thresholds. The methods compared have different effective request rates due to differing objectives to control the workload. For future work, the controller will be tested with different workload patterns, SLO thresholds, monitoring intervals and controller settings. The controller will be evaluated over longer periods. to understand the stability of the controller; how much throughput can be sustained and how generally applicable it is. This will be done by matching workload levels and using different application environments.

Author's Contribution: Faisal Bahadur: Conceptualization, Methodology, Software, Writeup and Supervision. Ibrar Afzal: Experimental Deployment and Configurations, Data Curation. Tayyba Bibi: Investigation, Statistical Analysis and Validation. Tanzeela Qazi: Testing, Evaluation and Analysis.

Data Availability Statement: The LoadGuard-JMeter project containing the source code and compiled jar plugin is available here: https://github.com/faisalbahadurhu/LoadGuard
The experimental CSV result files can be found here:
https://doi.org/10.6084/m9.figshare.34892763

Conflict of interest: The authors declare no conflict of interest.

References:

[1] S. Di Meglio, L. L. L. Starace, and S. Di Martino, "Web app performance testing in industrial contexts: Supporting workload generation with E2E-Loader++," *J. Syst. Softw.*, vol. 232, p. 112684, Feb. 2026, doi: 10.1016/J.JSS.2025.112684.

[2] A. Marchese and O. Tomarchio, "SLO-Aware Container Orchestration on Kubernetes Clusters," *IEEE Int. Conf. Cloud Comput. CLOUD*, pp. 318–327, 2025, doi: 10.1109/CLOUD67622.2025.00040.

[3] M. Bolanowski, M. Ćmil, and A. Starzec, "New Model for Defining and Implementing Performance Tests," *Futur. Internet 2024, Vol. 16, Page 366*, vol. 16, no. 10, p. 366, Oct. 2024, doi: 10.3390/FI16100366.

[4] "Apache JMeter - Apache JMeter™." Accessed: Nov. 04, 2026. [Online]. Available: https://jmeter.apache.org/

[5] E. Mabotha, N. E. Mabunda, and A. Ali, "Performance Evaluation of a Dynamic RESTful API Using FastAPI, Docker and Nginx," *IEEE Glob. Energy Conf. 2024, GEC 2024*, pp. 174–181, 2024, doi: 10.1109/GEC61857.2024.10881712.

[6] M. Pinyagin and A. Sadovykh, "Automating Performance Testing in CI/CD - Tools Evaluation," *Lect. Notes Comput. Sci.*, vol. 16107 LNCS, pp. 195–209, 2026, doi: 10.1007/978-3-032-05188-2_13.

[7] L. Liao *et al.*, "Using black-box performance models to detect performance regressions under varying workloads: an empirical study," *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 4130–4160, Sep. 2020, doi: 10.1007/S10664-020-09866-Z;JOURNAL:JOURNAL:KLU-EMSE;PAGE:STRING:ARTICLE/CHAPTER.

[8] N. Husufa and I. Prihandi, "Optimizing JMeter on Performance Testing Using the Bulk Data Method," *J. Inf. Syst. Informatics*, vol. 4, no. 2, pp. 205–215, Jun. 2022, doi: 10.51519/JOURNALISI.V4I2.244.

[9] J. Castro, N. Laranjeiro, and M. Vieira, "Detecting DoS Attacks in Microservice Applications: Approach and Case Study," *ACM Int. Conf. Proceeding Ser.*, pp. 73–78, Nov. 2022, doi: 10.1145/3569902.3569916;GROUPTOPIC:TOPIC:ACM-PUBTYPE.

[10] H. Schulz, D. Okanovic, A. Van Hoorn, and P. TÅma, "Context-Tailored Workload Model Generation for Continuous Representative Load Testing," *ICPE 2021 - Proc. ACM/SPEC Int. Conf. Perform. Eng.*, pp. 21–32, Apr. 2021, doi: 10.1145/3427921.3450240;WGROUP:STRING:ACM.

[11] "Free-Form Arrivals Thread Group :: JMeter-Plugins.org." Accessed: Nov. 04, 2026. [Online]. Available: https://jmeter-plugins.org/wiki/FreeFormArrivalsThreadGroup/

[12] "Throughput Shaping Timer :: JMeter-Plugins.org." Accessed: Nov. 04, 2026. [Online]. Available: https://jmeter-plugins.org/wiki/ThroughputShapingTimer/

[13] "Concurrency Thread Group :: JMeter-Plugins.org." Accessed: Nov. 04, 2026. [Online]. Available: https://jmeter-plugins.org/wiki/ConcurrencyThreadGroup/

[14] V. Ayala-Rivera, M. Kaczmarski, J. Murphy, A. Darisa, and A. O. Portillo-

Dominguez, "One size does not fit all: In-test workload adaptation for performance testing of enterprise applications," *ICPE 2018 - Proc. 2018 ACM/SPEC Int. Conf. Perform. Eng.*, vol. 2018-March, pp. 211–222, Mar. 2018, doi: 10.1145/3184407.3184418;WGROUP:STRING:ACM.

[15] O. Huerta-Guevara, V. Ayala-Rivera, L. Murphy, and A. O. Portillo-Dominguez, "DYNAMOJM: A JMeter Tool for Performance Testing Using Dynamic Workload Adaptation," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 11812 LNCS, pp. 234–241, 2019, doi: 10.1007/978-3-030-31280-0_14/FIGURES/6.

[16] M. H. Moghadam *et al.*, "Performance Testing Using a Smart Reinforcement Learning-Driven Test Agent," *2021 IEEE Congr. Evol. Comput. CEC 2021 - Proc.*, pp. 2385–2394, 2021, doi: 10.1109/CEC45853.2021.9504763.

[17] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper, "An Autonomous Performance Testing Framework using Self-Adaptive Fuzzy Reinforcement Learning," *Softw. Qual. J.*, vol. 30, no. 1, pp. 127–159, Aug. 2019, doi: 10.1007/s11219-020-09532-z.

[18] Y. Sun *et al.*, "PerfScout: An Adaptive Workload Generator in Software Performance Testing," *Proc. - 2026 IEEE/ACM 48th Int. Conf. Softw. Eng. Softw. Eng. Pract. ICSE-SEIP 2026*, pp. 520–530, Jul. 2026, doi: 10.1145/3786583.3786893;SUBPAGE:STRING:BASIC.

[19] H. Hanada and K. Ishibashi, "Service-Level Objective-Aware Load-Adaptive Timeout: Balancing Failure Rate and Latency in Microservices Communication," *IEEE Access*, vol. 13, pp. 137881–137895, 2025, doi: 10.1109/ACCESS.2025.3596118.

[20] "JMeter MCP Server - Supercharge your JMeter performance testing with AI-driven orchestration and analysis. - TestDevTools." Accessed: Nov. 04, 2026. [Online]. Available: https://testdev.tools/jmeter-mcp-server/

[21] J. Von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A micro-service reference application for benchmarking, modeling and resource management research," *Proc. - 26th IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst. MASCOTS 2018*, pp. 223–236, Nov. 2018, doi: 10.1109/MASCOTS.2018.00030.

[22] "ConstantThroughputTimer (Apache JMeter dist API)." Accessed: Nov. 04, 2026. [Online]. Available: https://jmeter.apache.org/api/org/apache/jmeter/timers/ConstantThroughputTimer.html

[23] "AutoStop :: JMeter-Plugins.org." Accessed: Nov. 04, 2026. [Online]. Available: https://jmeter-plugins.org/wiki/AutoStop/

Copyright © by authors and 50Sea. This work is licensed under Creative Commons Attribution 4.0 International License.