PerfScout: An Adaptive Workload Generator in Software Performance Testing

Yongqian Sun
Nankai University
Tianjin, China
sunyongqian@nankai.edu.cn

Qingliang Zhang
Nankai University
Tianjin, China
zhangqingliang@mail.nankai.edu.cn

Xiao Xiong
Nankai University
Tianjin, China
xiongxiao@mail.nankai.edu.cn

Mengyao Li
Nankai University
Tianjin, China
limengyao@mail.nankai.edu.cn

Yimin Zuo
Nankai University
Tianjin, China
2213023@mail.nankai.edu.cn

Shenglin Zhang*
Nankai University
Tianjin, China
zhangsl@nankai.edu.cn

Xidao Wen
BizSeer
Beijing, China
wenxidao@bizseer.com

Wenwei Gu
Nankai University
Tianjin, China
wwgu@nankai.edu.cn

Huandong Zhuang
Huawei Cloud
Chengdu, China
zhuanghuandong@huawei.com

Bowen Deng
Huawei Cloud
Chengdu, China
dengbowen10@huawei.com

Ruiyuan Wan
Huawei Cloud
Chengdu, China
wanruiyuan@huawei.com

Dan Pei
Tsinghua University
Beijing, China
peidan@tsinghua.edu.cn

Abstract

Effective performance testing is essential for ensuring the reliability of large-scale software systems under varying load conditions. However, conventional workload generation techniques often rely on static thresholds and rule-based heuristics, which lack adaptability and generalizability across diverse systems. This paper presents *PerfScout*, a reinforcement learning–based framework for adaptive workload generation that automates the performance testing process. *PerfScout* integrates three components: dynamic breaking point identification using SPOT, local stationarity assessment via ADF and KPSS tests, and a policy optimization module based on Proximal Policy Optimization (PPO). By continuously observing key performance indicators and adapting the testing workload in real time, *PerfScout* enhances the automation of performance testing while improving testing efficiency. *PerfScout* has been deployed in *Huawei Cloud*, and experimental results on two datasets collected from its industrial environment demonstrate that *PerfScout* achieves over 82% accuracy in breaking point identification and improves testing efficiency by up to 90%, underscoring its practical applicability and effectiveness.

*Shenglin Zhang is the corresponding author.

ICSE-SEIP '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2426-8/2026/04
https://doi.org/10.1145/3786583.3786893

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

performance testing, workload generation, reinforcement learning, breaking point identification

ACM Reference Format:
Yongqian Sun, Qingliang Zhang, Xiao Xiong, Mengyao Li, Yimin Zuo, Shenglin Zhang, Xidao Wen, Wenwei Gu, Huandong Zhuang, Bowen Deng, Ruiyuan Wan, and Dan Pei. 2026. *PerfScout*: An Adaptive Workload Generator in Software Performance Testing. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEIP '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3786583.3786893

1 Introduction

With the widespread availability of high-speed internet, user expectations for software performance have increased considerably. Failures under high-load conditions can result in severe operational and financial consequences. For example, Costco's official website suffered a 16.5-hour outage during Thanksgiving due to a traffic surge, incurring an estimated $11 million in losses [34]. Similarly, HealthCare.gov experienced one of the most significant system crashes in history when user traffic reached 250,000—far exceeding the projected 50,000—shortly after launch [1]. These cases highlight the critical role of performance in maintaining user trust, operational continuity, and business viability. Ensuring system stability and efficiency under peak workloads (*i.e.*, the number of user requests per second) is therefore essential.

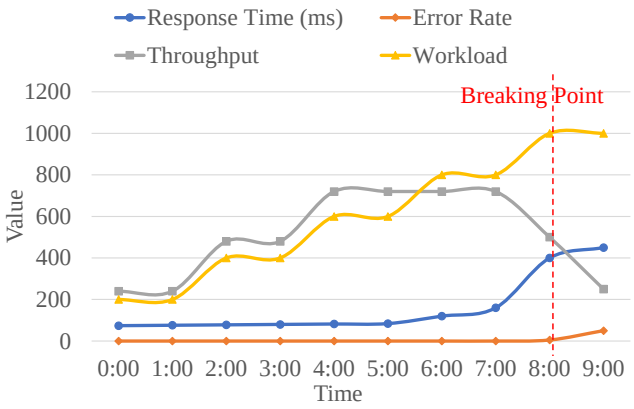

Figure 1: System behavior under progressive workload increase during performance testing. The breaking point indicates the onset of performance degradation, characterized by rising response time and error rate, and a drop in throughput.

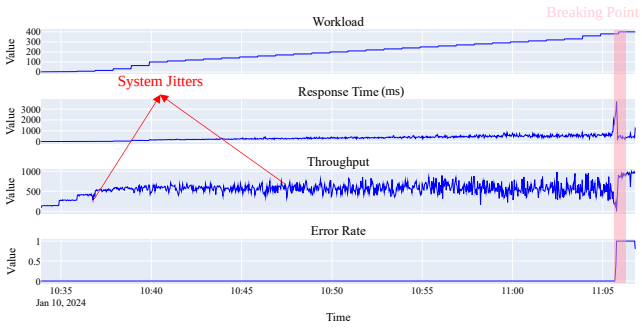

Figure 2: A real-world performance testing case illustrating system jitter and breaking point. Transient fluctuations in throughput and response time are observed prior to the onset of sustained performance degradation.

To ensure the stability of large-scale software systems, performance testing is typically conducted prior to deployment to evaluate system capacity and inform mitigation strategies such as traffic throttling and service degradation [9, 14, 17, 19]. As shown in Fig. 1, this process involves progressively increasing the workload on the system under test (SUT) while monitoring key performance indicators (KPIs) such as throughput, response time, and error rate. Initially, the system maintains stable performance, but beyond a certain threshold—the breaking point—response time and error rate rise sharply while throughput declines, indicating that the system has exceeded its capacity and entered a state of performance degradation. The workload at this point is defined as the system's maximum sustainable load under the given conditions.

In standard industry practice, this process is largely manual [13, 20]. Engineers typically script discrete workload levels and define static thresholds for KPIs to determine when performance has degraded. Testing proceeds iteratively: workloads are applied, KPIs are observed, and the load is increased until performance degradation is detected or threshold conditions are violated. While commonly used, this approach is labor-intensive, prone to human error, and poorly suited to the dynamic and heterogeneous nature of modern systems. These limitations highlight the need for automated, efficient solutions that can adapt to diverse system behaviors with minimal human oversight.

In recent years, numerous workload generation methods have been proposed to improve the efficiency and automation of performance testing [2, 4, 15, 21–23, 28, 30–32, 35, 37]. Broadly, these methods fall into three categories: white-box workload generation methods (WWGMs), static black-box workload generation methods (SBWGMs), and dynamic black-box workload generation methods (DBWGMs). WWGMs [35, 37], such as data flow analysis and symbolic execution, can pinpoint performance bottlenecks with high precision but require source code access, restricting their practicality in automated black-box testing. SBWGMs [28, 30–32] predefine workloads with tools like JMeter [10], yet rely heavily on expert input and manual configuration, which undermines testing efficiency and hinders scalability. DBWGMs [4, 15] improve adaptability by adjusting workloads at runtime based on KPI monitoring, but still depend on manually designed rules and thresholds. Consequently, despite their progress, existing approaches struggle to achieve fully automated and efficient stress testing, especially in dynamic and heterogeneous system environments.

Reinforcement learning (RL) offers a promising direction for automating workload generation, as agents can efficiently adapt to dynamic system behaviors without handcrafted rules or source code access [2, 21–23]. Nevertheless, existing RL-based approaches remain far from delivering fully automated and scalable performance testing. Specifically, they continue to face key challenges in real-world scenarios, including:

(1) Difficulty in accurately identifying performance limits. Performance testing typically relies on predefined stopping conditions, such as response time or throughput thresholds [28, 30, 31], to determine when a system reaches its limit. However, variations in business logic and resource configurations across SUTs can cause KPI behaviors to differ markedly under stress. Fixed thresholds may miss true degradation points, resulting in inaccurate capacity estimates and repeated test cycles.

(2) Interference of transient system jitter. As shown in Fig. 2, during performance testing, the KPIs of the SUT can experience temporary jitters due to factors such as network fluctuations and resource contention. If these KPIs, which include such jitters, are used as the basis for deciding the workload for the next concurrency phase, it will lead to inaccurate decision-making, reducing the reliability of measuring the system's capacity.

(3) Great diversity in the performance feedback of SUTs. The performance feedback of SUTs varies widely across interfaces, yielding heterogeneous KPI patterns that are difficult to model consistently [2, 4, 15, 21–23]. Such diversity complicates workload generation, as strategies effective in one setting may fail in another. Designing methods that can automatically adapt to diverse performance feedback without system-specific customization remains a key challenge.

We propose *PerfScout*, a non-intrusive RL-based framework that automates workload generation for efficient and adaptive performance testing.Specifically, *PerfScout* consists of three modules that correspond to the above three challenges:

(1) *Breaking Point Identification.* We use the SPOT [29] technique based on Extreme Value Theory (EVT) to dynamically generate KPI thresholds, and accurately identify performance limits by detecting KPI anomalies that deviate from expectations.

(2) *Local Stationarity Identification.* This module uses stationarity testing techniques to determine whether the SUT is in a local stationary state. Only when the system is stationary can subsequent workload adjustment decisions be made, thereby mitigating the impact of transient performance jitter.

(3) *Adaptive Workload Decision.* We incorporate the dynamic thresholds provided by SPOT into the RL reward design, removing the need for system-specific manual configuration and improving generalization across diverse environments.

The contributions of our work are summarized as follows:

- We propose *PerfScout*, a general-purpose performance testing framework based on RL that achieves automated workload generation and efficient capacity assessment across diverse systems.
- To eliminate the interference of system jitter on the performance testing results, *PerfScout* uses stationarity test techniques to determine whether the system is in a local stationary state. If it is in the stationary state phase, the subsequent workload adjustment mechanism is triggered to ensure the reliability of the test results. Moreover, *PerfScout* integrates the dynamic thresholds provided by SPOT into the design of the RL reward elements, enhancing the agent's adaptability to performance feedback from different systems.
- *PerfScout* has been deployed in *Huawei Cloud* for nine months. To evaluate its performance, we conducted experiments on two datasets from *Huawei Cloud*'s testing and production environments. Results show that *PerfScout* improves testing efficiency by up to 90%, achieves a breaking point identification accuracy above 82%, and yields a harmonic mean exceeding 86%, outscoring baseline strategies with fixed step sizes or static heuristics. Due to confidentiality agreements, the data cannot be released while our training and evaluation code is made available[1].

2 Motivation

This section addresses two key questions: (1) How do SUTs behave near performance limits? and (2) What common patterns emerge at the breaking point? The variability of KPIs across systems and the presence of shared degradation signatures highlight the need for a generalizable, automated approach that adapts dynamically to diverse environments.

2.1 How Do Different SUTs Behave at the Breaking Point?

Existing methods [2, 21–23] leverage RL for automated workload generation but often rely on manually set KPI thresholds. These static thresholds fail to account for varying performance across different SUTs. To examine this limitation, we conducted an empirical analysis on two real-world datasets, D1 and D2 (details in Section 5.1), extracting workload, response time, and error rate at the identified breaking point for each case.

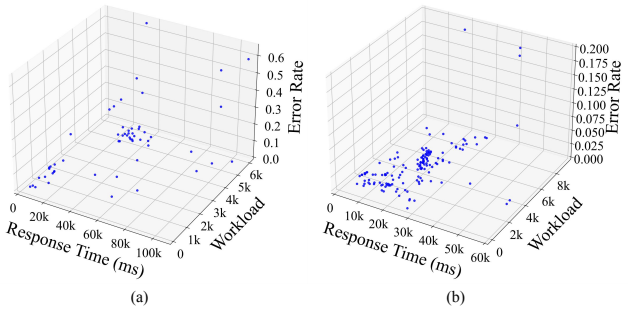

Figure 3: Distributions of SUT performance at the breaking point across three dimensions: workload, response time, and error rate. (a) Data from dataset D1; (b) Data from dataset D2. Each point represents a unique test case.

Fig. 3 presents 3D scatter plots of these metrics. The distributions vary widely: workloads span 1,000–8,000, response times 3,000–30,000 ms, and error rates 0.01–0.2. Such heterogeneity indicates that fixed thresholds (e.g., 5,000 ms response time) are either overly conservative or insufficient, leading to premature or delayed detection. This motivates the need for adaptive mechanisms that adjust to system-specific behaviors.

Beyond inter-system diversity, we also observe substantial intra-system instability. KPIs such as throughput frequently fluctuate due to transient contention or network jitter—even before degradation occurs. As shown in Fig. 2, throughput oscillates sharply prior to sustained slowdown. If workload decisions react to such noise, the agent may take erratic actions, slowing convergence and reducing accuracy. These findings underscore the need to assess local stationarity before adjusting workloads.

Finding 1: SUTs exhibit substantial variability in KPIs at the breaking point, rendering manually configured thresholds ineffective across heterogeneous systems. Moreover, intra-system KPI fluctuations prior to stabilization can mislead workload decisions, compromising both the efficiency and accuracy of performance testing.

2.2 What Common Performance Patterns Emerge Across SUTs at the Breaking Point?

To explore eliminating manual thresholds, we analyzed over 200 cases from D1 and D2 to identify consistent patterns. Despite absolute value differences, many systems exhibit similar degradation trajectories. Fig. 2 presents a representative case. Initially, the system maintains stable behavior with gradual response time increases and near-zero error rates. However, once the workload exceeds a

[1] https://anonymous.4open.science/r/PerfScout-457E

Table 1: Proportions of response time spikes and error rate increases observed at the breaking point across different datasets.

Dataset	Response Time Spike (%)	Error Rate Increase (%)
D1	60.7	75.0
D2	74.7	82.9

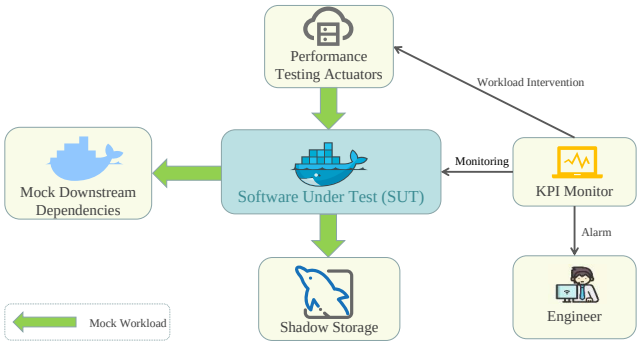

Figure 4: System architecture of software performance testing. Actuators generate mock workloads while the monitor collects real-time KPIs to detect anomalies.

threshold (e.g., 380), both metrics escalate sharply—response time jumps from 800 ms to over 3,000 ms, and error rates surge. This sudden shift signals the breaking point.

This pattern is widespread. As shown in Table 1, 60.7%–74.7% of cases exhibit response time spikes, and 75.0%–82.9% show error rate increases at the breaking point. Nearly all cases display at least one of these phenomena. These observations suggest that such KPI shifts serve as robust, system-agnostic indicators for test termination, eliminating the need for manual tuning and providing a foundation for adaptive workload control.

Finding 2: SUTs consistently exhibit a spike in response time, an increase in error rate, or both when approaching the performance limit. These common patterns can be leveraged as generalizable signals for automatically terminating performance tests, thereby eliminating the reliance on manually configured thresholds.

3 Preliminaries

3.1 Performance Testing and KPIs

As illustrated in Fig. 4, the testing framework monitors the SUT under simulated workloads generated by actuators. Key Performance Indicators (KPIs), including throughput, response time, and error rate, are collected to assess capacity. We define a KPI sequence as time-series data $X = \{x_1, x_2, \ldots, x_T\}$, where $x_t \in \mathbb{R}$ represents the metric value at timestamp t.

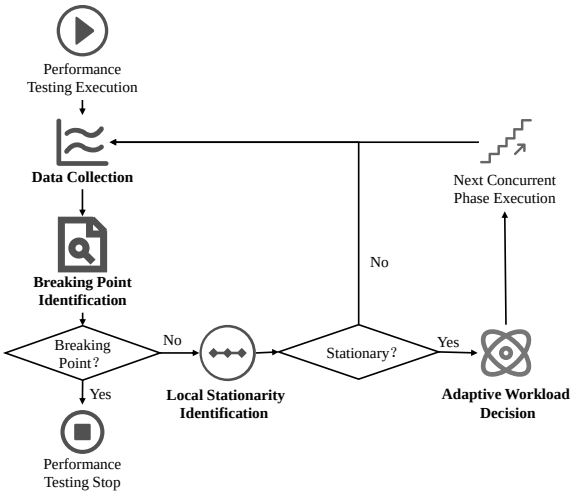

Figure 5: The overall framework of *PerfScout*, illustrating its four core modules: data collection, breaking point identification, local stationarity identification, and adaptive workload decision.

3.2 Reinforcement Learning

Reinforcement Learning (RL) optimizes a policy π_θ to maximize expected cumulative rewards. We employ Proximal Policy Optimization (PPO) [8] as our core algorithm. PPO outperforms classical methods like REINFORCE [36] and TRPO [8] by utilizing a clipped surrogate objective, which ensures stable policy updates and high sample efficiency without the computational overhead of second-order optimization.

3.3 Problem Statement

Our objective is to accurately identify the SUT's breaking point—the maximum sustainable workload—while minimizing testing duration. Traditional static thresholds fail to adapt to heterogeneous system behaviors and resource configurations. We formulate this as an RL problem where the agent dynamically adjusts workloads based on real-time KPI feedback to efficiently converge on the true performance limit.

4 Approach

In this section, we introduce the overall design of *PerfScout* and describe its four main modules that work together to support automated and adaptive performance testing.

4.1 Overview

As illustrated in Fig. 5, *PerfScout* is composed of four modules, which work in a closed feedback loop to progressively explore the system's performance under increasing workload.

(1) Data Collection. KPIs, such as throughput, response time, and error rate, are periodically collected from the system under test. These metrics serve as the input for subsequent decision-making steps.

(2) Breaking Point Identification. To detect the onset of performance degradation, *PerfScout* utilizes the SPOT algorithm [29], which leverages Extreme Value Theory (EVT) to establish dynamic thresholds. Unlike static thresholding methods, this approach adapts to varying system behaviors. A custom alarm rule further filters transient fluctuations, ensuring robust detection of the actual breaking point, thereby addressing the challenge of manual configuration.

(3) Local Stationarity Identification. To prevent misjudgment caused by short-term metric jitter, *PerfScout* applies two complementary statistical tests—ADF [7] and KPSS [16]—to assess whether the system has reached a steady state at the current load level. Only upon confirmation of local stationarity does the system proceed to the next workload level, effectively mitigating noise interference.

(4) Adaptive Workload Decision. The workload adjustment process is modeled as a RL problem. Using the PPO algorithm, an agent is trained to select the optimal concurrency level based on real-time KPI feedback. The reward function incorporates SPOT-derived thresholds to balance between efficiency and reliability, allowing *PerfScout* to generalize across heterogeneous systems without prior tuning.

4.2 Data Collection

When the performance testing begins, actuators send requests to the SUT according to the specified number of concurrent users (*i.e.*, workload). The SUT generates KPI data every second as it processes these requests, including indicators such as throughput (TPS), response time (RT), and error rate (ER). To minimize network overhead, the data collection system aggregates and reports these performance indicators at six-second intervals. Based on this periodically collected data, *PerfScout* makes decisions about workload adjustments to further explore the SUT's performance characteristics.

4.3 Breaking Point Identification

As shown in Fig. 6, we illustrate the primary process for identifying the breaking point, where *PerfScout* determines whether the SUT has reached its performance limit by analyzing ER and RT indicators. The process begins with *PerfScout* applying data smoothing, specifically a simple moving average with backward filling, to the most recent KPI data received from the indicator collection system. This smoothing step mitigates noise in KPI reporting from the SUT, reducing spurious fluctuations and enhancing the robustness of breaking point detection. When the smoothed error rate exceeds the predefined threshold of 0.01, *PerfScout* flags that time point as a potential breaking point. Simultaneously, *PerfScout* calculates the first-order difference of RT and employs SPOT to detect abrupt changes, which yields another set of suspected breaking points. In the final step, *PerfScout* applies a customized alarm rule to filter these candidates and identify the definitive breaking point where SUT performance significantly degrades.

SPOT. SPOT is an adaptive thresholding algorithm based on EVT [29], used to detect rare but significant changes such as performance breaking points. SPOT first sets a peak threshold t (e.g., 98th percentile) and fits a Generalized Pareto Distribution (GPD) to excesses above t to compute the anomaly threshold z_q. During

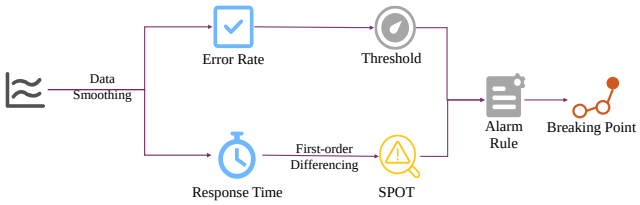

Figure 6: Workflow of breaking point identification. After smoothing the incoming KPI data, *PerfScout* checks whether the error rate exceeds a fixed threshold and applies first-order differencing and SPOT to detect abrupt changes in response time. An alarm rule then integrates both signals to confirm the breaking point.

detection, data above z_q are flagged as anomalies, while those between t and z_q are used to incrementally update the GPD, enabling SPOT to adapt to dynamic data streams with evolving distributions.

Alarm Rule. To reduce noise interference, *PerfScout* employs a sliding window–based alarm rule to stabilize detection results. A queue records the most recent w observations, and a breaking point is declared when the fraction of anomalies detected by SPOT within this window exceeds a predefined threshold k (e.g., $w = 12$, $k = 0.5$). Once activated, this rule signals test termination, indicating that the system has reached its performance limit.

4.4 Local Stationarity Identification

After determining that no breaking point has been identified, *PerfScout* conducts local stationarity identification based on the TPS indicator from the most recently reported KPIs to decide whether to proceed to the next concurrent phase in the performance testing. As shown in Fig. 7, *PerfScout* first extracts TPS from the most recent interval of length l (set to 12) and simultaneously performs both ADF [7] and KPSS [16] stationarity tests on this data segment. If both tests indicate non-stationarity in the TPS data, it suggests that the system has not yet reached a stationary state at the current concurrency phase, requiring continued testing at this level. Conversely, if the tests indicate stationarity, the system is considered to have stabilized at the current concurrency phase, and *PerfScout* will adjust the workload to proceed to the next phase of performance testing.

ADF test operates by constructing an autoregressive model to perform regression analysis on time series data, specifically testing for the presence of unit roots. The presence of a unit root indicates non-stationarity in the series, while its absence confirms stationarity. In contrast, KPSS test establishes a linear trend model and analyzes time series data through regression to examine deviations from this linear model. If these deviations are sufficiently small, the time series is proven stationary. Otherwise, it is non-stationary. As demonstrated by extensive experimental results in Section 5.3, *PerfScout*'s approach of combining these two stationarity tests not only enhances performance testing efficiency but also ensures the accuracy of breaking point identification, providing complementary statistical validation that mitigates the limitations of either test used in isolation.

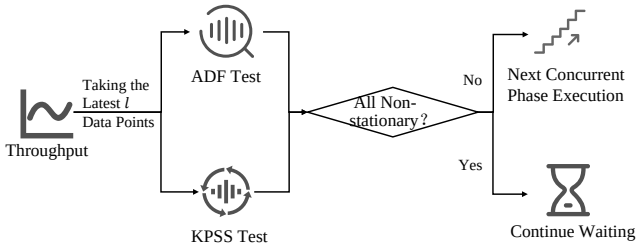

Figure 7: Workflow of local stationarity identification. The latest l throughput observations are subjected to both ADF and KPSS tests. If neither test indicates non-stationarity, the system is deemed stable and proceeds to the next concurrency phase; otherwise, it continues waiting.

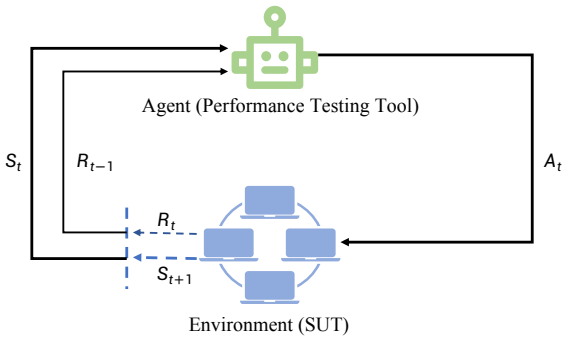

Figure 8: RL loop for adaptive workload decision. The agent (performance testing tool) observes the current system state S_t, selects an action A_t to adjust the workload, and receives a reward R_t based on system feedback. The environment (SUT) responds with a new state S_{t+1} and performance indicators such as RT_t and ERR_t.

4.5 Adaptive Workload Decision

Reinforcement Learning (RL) enables an agent to learn optimal decision strategies through interaction with the environment. An RL framework consists of states S, actions A, and rewards R. As shown in Fig. 8, at each step the agent observes the current state S_t, selects an action A_t, receives a reward R_t, and transitions to a new state S_{t+1}. The reward reflects the quality of the action, and the agent's objective is to maximize the expected cumulative return through continual exploration and policy improvement. This trial-and-error optimization makes RL well suited for sequential decision-making problems.

During the performance testing process, *PerfScout* treats the performance testing tool as an agent and the SUT as the environment. The KPIs of the SUT are considered as the state S_t, the workload adjustment in each iteration is treated as the action A_t, and the reward R_t is calculated by comprehensively considering both the SUT's KPIs and the number of stress adjustments. This framework establishes a RL model that enables adaptive decision-making for workload adjustments throughout the performance testing process. The reward function is specifically represented as follows:

$$R_t = \left(\frac{RT_t}{RT_{spot}}\right)^2 + \left(\frac{ERR_t}{ERR_{\text{threshold}}}\right)^2 - c * \alpha \tag{1}$$

The design philosophy behind the reward R_t is to ensure that the response time (RT_t) and error rate (ERR_t) reach their thresholds as quickly as possible while keeping the SUT operating safely and minimizing the number of stress adjustments c (with a penalty coefficient α set to 0.01).

It is worth noting that due to different business logic and resource configurations across various SUTs, the normal range of the RT indicator typically varies and cannot have a standard range of $[0, 1]$ like the ERR indicator. This presents certain challenges for model training convergence. To address this issue, *PerfScout* incorporates the dynamic threshold RT_{spot} obtained from the SPOT to standardize the RT indicators of different SUTs, thereby accelerating the convergence speed of the training process. After completing the design of the three essential elements of RL, *PerfScout* employs the PPO [8] algorithm to train the agent.

5 Evaluation

In this section, we empirically evaluate *PerfScout* by addressing the following research questions:

- **RQ1**: How effectively does *PerfScout* perform in improving testing efficiency?
- **RQ2**: What is the contribution of each core module to the overall performance of *PerfScout*?
- **RQ3**: How sensitive is *PerfScout* to its hyperparameters?

5.1 Experimental Setup

Datasets. To evaluate *PerfScout*, we conduct experiments on two real-world datasets, D1 and D2, derived from internal records of *Huawei Cloud*'s cloud-based testing platform, *CodeArts PerfTest*[2], which supports performance assessment across development and operations. D1 contains 56 test records covering 10 APIs from data and monitoring services, while D2 includes 146 production-level records across 16 APIs spanning data, monitoring, and authentication services. The datasets differ in environment, API coverage, and service diversity (full API list in Table 2). Direct RL training in real environments is impractical due to the high cost and thousands of iterations needed for convergence. To address this, we construct simulation environments E1 and E2 by fitting Gaussian distributions [11] to D1 and D2, enabling efficient and reproducible generation of synthetic KPI sequences for agent training.

Evaluation Metrics. To assess the performance of *PerfScout*, we adopt three evaluation metrics: accuracy in identifying the breaking point (Accuracy), improvement in performance testing efficiency (Efficiency), and their harmonic mean (HM). The formula for Accuracy is as follows:

$$\text{Accuracy} = \frac{1}{|A|}\sum_{a \in A} \mathbb{I}(f(a) \in Y_a) \tag{2}$$

Where A is the set of test samples, Y_a denotes the interval range label of the breaking point for test sample a, $f(a)$ is the breaking point identified by *PerfScout* and $\mathbb{I}$ represents the indicator function, which takes the value 1 if the condition is true and 0 otherwise.

[2]https://www.huaweicloud.com/product/cpts.html

Table 2: APIs and their service types in D1 and D2.

Service Type	APIs in D1	APIs in D2
Data Services	exportMonitorDataV3, getPerformanceDataV3, getReportUrlByTaskIdV3, getTaskReportDetails, getTaskReportOutline, getTestcaseDataV3, retrieveTaskDetailV3	exportCsv, exportMonitorDataV3, getFullChartByCaseUri, getPerformanceDataV3, getProjectUsingGET, getTaskCaseList, getTaskReportDetails, getTaskReportOneCaseDetail, getTaskReportOutline, getTestcaseDataV3
Monitoring Services	progressOfAomData, queryAomAppList, queryProjectId	health monitor, progressOfAomData, queryAomAppList, queryCaseDetailsUsingGET, queryProjectId
Authorization Services	—	cptsServiceAuthorization

Accuracy measures how closely PerfScout's detected breaking point aligns with the true saturation interval. A higher value indicates more precise and reliable breaking-point identification.

The formula for Efficiency is given by:

$$\text{Efficiency} = 1 - \frac{\tau}{\tau'} \tag{3}$$

Here, τ denotes the time taken by *PerfScout* to complete the performance testing, while τ' represents the time required by a traditional fixed strategy—using a fixed concurrency duration of 60 seconds and a fixed workload adjustment of 50—to reach the same workload. The Efficiency metric quantifies the temporal advantage over traditional methods in completing performance testing. A value approaching 1 signifies maximal time savings and superior testing efficiency.

The HM is calculated as:

$$\text{HM} = 2 \times \frac{\text{Accuracy} \times \text{Efficiency}}{\text{Accuracy} + \text{Efficiency}} \tag{4}$$

HM integrates Efficiency and Accuracy to ensure holistic performance. By mitigating the risks of rapid yet erroneous detection, or accurate yet inefficient execution, it effectively reflects a method's operational viability. Higher HM value implies superior overall performance. Therefore, when evaluating the results of the performance tests, we place greater emphasis on the HM value.

Although newly defined for our evaluation, these metrics draw on non-novel core assessment concepts of accuracy and efficiency from prior work. For example, time saved to reach failure states and performance issues detected within fixed time windows were adopted in prior work [4, 23]. However, these metrics are tied to each method's unique workflow and stopping criteria, rendering them unsuitable for assessing breaking-point identification accuracy and time costs. Similarly, the metrics used in prior work (e.g., generated users, adjustment steps, bottleneck-triggering inputs) [2, 21, 23] only quantify the cost of meeting preset thresholds—not breaking-point localization precision or testing process duration. Thus, while conceptually aligned with prior work, our redefined metrics better evaluate a method's accuracy, efficiency, and overall effectiveness.

Baseline Approaches. To rigorously benchmark the effectiveness of *PerfScout*, we compare it against four representative baseline methods, each adopting distinct stopping criteria and workload adjustment strategies for performance testing. Their configurations are as follows:

- **Random Decision (RD)**: RD stops performance testing when the error rate exceeds a predefined threshold. It uses a fixed concurrency duration for each phase, with workload adjustments randomly sampled from a uniform distribution. Specifically, the error rate threshold is set to 0.01, the duration to 60 seconds, and the adjustment range to $[0, 100]$.
- **Linear Decision (LD)**: LD uses the same stopping criterion and concurrency duration for each phase as RD. However, unlike RD, the size of each workload adjustment is proportional to the current accumulated adjustment count. In the implementation, the proportionality coefficient is set to 2.
- **DYNAMO [4]**: DYNAMO monitors the KPIs generated during the performance testing in real-time. It assesses the system state based on the RT change rate and adjusts the workload dynamically. The RT change threshold is set to 0.05 in the implementation.
- **RELOAD [21]**: RELOAD integrates reinforcement learning into performance testing by designing a reward function based on response time and error rate, enabling the agent to automatically learn workload strategies. However, RELOAD relies on static thresholds, which limits its ability to adapt to varying system states. In the implementation, the response time threshold is fixed at 5,000 ms and the error rate threshold at 0.01.

Table 3: Comparison of overall performance testing results across all methods on datasets D1 and D2. Metrics include accuracy of breaking point identification, testing efficiency, and their harmonic mean (HM).

Method	D1			D2		
	Accuracy	Efficiency	HM	Accuracy	Efficiency	HM
PerfScout	**0.875**	**0.900**	**0.887**	**0.822**	**0.901**	**0.860**
RD	0.750	0.076	0.138	0.719	0.075	0.136
LD	0.732	0.149	0.248	0.719	0.156	0.256
DYNAMO	0.732	0.402	0.519	0.671	0.376	0.481
RELOAD	0.732	0.878	0.798	0.753	0.877	0.810

Implementation. *PerfScout* is implemented using Python 3.9.13, PyTorch 2.3.1, scikit-learn 1.5.1, and statsmodels 0.14.1. All experiments were conducted on a Linux server with 2 Intel(R) Xeon(R) CPU E5-2650 v4 processors (@ 2.20GHz) and 128GB of RAM (without GPUs).

The testing environment reports performance KPIs every 6 seconds. Accordingly, the queue size for local stationarity identification is set to 12, enabling comparison of the two most recent system states; smaller sizes are not recommended, as they compromise the reliability of the stationarity tests. For breaking point identification, the queue capacity and anomaly threshold are configured at 12 and 0.5, respectively [38]. The SPOT initial quantile (97%) and PPO clipping parameter ϵ (0.2) adhere to standard configurations [27, 29]. The following three parameters—SPOT risk probability (10^{-5}), KPSS and ADF significance level (0.05), and load adjustment penalty coefficient (0.01)—are validated through sensitivity experiments. The agent in *PerfScout* is first trained on E1 to learn optimal workload adjustment strategies, and then applied to both E1 and E2 to evaluate improvements in testing efficiency across 56 and 146 simulated tests, respectively.

5.2 RQ1: Overall Performance

TABLE 3 summarizes the overall performance of all baselines. *PerfScout* consistently outperforms others on D1 and D2, achieving the highest Accuracy, Efficiency, and harmonic mean (HM). Notably, *PerfScout* attains an HM of 0.887 and 0.860 on D1 and D2, respectively, surpassing the best-performing baseline, RELOAD, by at least 0.05, and exceeding the HM scores of RD, LD, and DYNAMO by at least 0.368.

This superiority stems from *PerfScout*'s adaptive mechanisms. Unlike RD, LD, and DYNAMO, which use fixed concurrency durations and fail to accommodate system variability, *PerfScout* dynamically detects local stationarity to guide workload adjustments, reducing test time while maintaining accuracy. Although RELOAD also applies RL, its reliance on fixed thresholds limits adaptability. In contrast, *PerfScout* employs SPOT for dynamic thresholding, enabling precise capture of diverse system behaviors and improving detection and decision efficiency. These results validate *PerfScout*'s effectiveness and generalizability in accurate, efficient performance testing across different systems.

5.3 RQ2: Ablation Study

To understand the contribution of each core component in *PerfScout*, we conduct an ablation study focusing on: (1) breaking point identification, (2) local stationarity identification, and (3) adaptive workload decision-making. For each component, we construct several controlled variants of *PerfScout* by substituting the corresponding technique, and evaluate them on both D1 and D2 datasets.

Breaking Point Identification. We assess the role of SPOT in breaking point detection by replacing it with alternative thresholding variants:

- **C1**: Replaces SPOT with boxplot-based detection [18].
- **C2–C4**: Replace SPOT with K-sigma detection using $k \in \{3, 5, 10\}$ [6].

Table 4: Ablation study on the breaking point identification module. Performance comparison between *PerfScout* and its variants (C1–C4).

Method	D1			D2		
	Accuracy	Efficiency	HM	Accuracy	Efficiency	HM
PerfScout	**0.875**	0.900	**0.887**	**0.822**	0.901	**0.860**
C1	0.429	0.886	0.578	0.582	0.896	0.706
C2	0.786	0.900	0.839	0.726	0.900	0.804
C3	0.786	0.901	0.840	0.774	**0.902**	0.833
C4	0.750	**0.902**	0.819	0.760	0.901	0.825

Table 5: Ablation study on the local stationarity identification module. Comparison of *PerfScout* with variants (C5–C10) using different stationarity detection techniques.

Method	D1			D2		
	Accuracy	Efficiency	HM	Accuracy	Efficiency	HM
PerfScout	**0.875**	**0.900**	**0.887**	**0.822**	**0.901**	**0.860**
C5	0.732	0.826	0.776	0.774	0.826	0.799
C6	0.786	0.897	0.838	0.781	0.897	0.835
C7	0.821	0.731	0.773	0.767	0.729	0.748
C8	0.750	0.167	0.273	0.767	0.170	0.278
C9	0.804	0.371	0.507	0.733	0.368	0.490
C10	0.714	0.479	0.573	0.740	0.490	0.590

As shown in Table 4, *PerfScout* consistently outperforms all variants on D1 and D2. Limitations in the baselines hamper their performance: the boxplot method (C1) struggles with heavy-tailed distributions due to rigid interquartile range, while K-sigma methods (C2-C4) lack generalizability by assuming data normality. In contrast, grounded in Extreme Value Theory , SPOT enables distribution-agnostic adaptation to diverse KPI patterns. This ensures robust breaking point identification across varying scenarios.

Local Stationarity Identification. To evaluate the effectiveness of combining ADF and KPSS tests for detecting steady state, we compare against six variants:

- **C5**: Uses only the ADF test [7].
- **C6**: Uses only the KPSS test [16].
- **C7**: Uses the PP test [25].
- **C8–C10**: Use the slope method [12] with slope thresholds set to 0.6, 0.8, and 1, respectively.

As shown in Table 5, *PerfScout* achieves superior performance on D1 and D2, validating the effectiveness of combining ADF and KPSS. Single-test baselines exhibit distinct limitations: ADF (C5) and KPSS (C6) suffer from noise sensitivity and specific assumptions, while the PP test (C7), despite correcting for autocorrelation and heteroscedasticity, proves unstable on short time series. Furthermore, slope-based methods (C8–C10) are highly noise-sensitive, where static thresholds lead to a trade-off between premature adjustments (*e.g.*, C8) and missed instabilities (*e.g.*, C10).

Adaptive Workload Decision. To examine the effectiveness of PPO for workload control, we evaluate the following replacements:

- **C11**: Uses Q-learning [33].
- **C12**: Uses DQN [24].

Table 6: Ablation study on the adaptive workload decision module. Comparison of *PerfScout* with RL-based variants (C11–C13) across datasets D1 and D2.

Method	D1			D2		
	Accuracy	Efficiency	HM	Accuracy	Efficiency	HM
PerfScout	0.875	**0.900**	**0.887**	**0.822**	**0.901**	**0.860**
C11	0.804	0.896	0.848	0.740	0.897	0.811
C12	**0.893**	0.842	0.867	0.781	0.840	0.809
C13	0.875	0.873	0.874	0.767	0.873	0.817

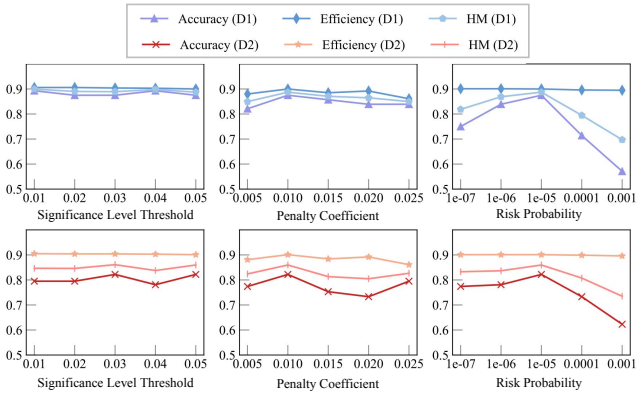

Figure 9: Sensitivity analysis of *PerfScout* with respect to three key hyperparameters: significance level threshold, penalty coefficient, and risk probability. Results are reported on datasets D1 and D2 using Accuracy, Efficiency, and HM.

- **C13**: Uses REINFORCE [36].

Table 6 demonstrates that *PerfScout* achieves the best performance with PPO. Baselines exhibit distinct shortcomings: Q-learning (C11) and DQN (C12) are constrained by discrete action spaces and Q-value estimation bias, while REINFORCE (C13) suffers from high gradient variance and low sample efficiency. In contrast, PPO leverages a clipped surrogate objective to maintain training stability and support continuous action outputs. This design ensures superior convergence speed and policy robustness, validating its effectiveness for adaptive workload decision-making.

5.4 RQ3: Hyperparameter Sensitivity

To evaluate the robustness of *PerfScout*, we analyze its sensitivity to three key hyperparameters: the significance level threshold for stationarity tests, the penalty coefficient in the reward function, and the risk probability used in SPOT. As illustrated in Fig. 9, *PerfScout* maintains stable performance across a wide range of settings, demonstrating its robustness.

Significance Level Threshold. The significance level threshold governs hypothesis testing in the ADF and KPSS procedures. We vary this threshold from 0.01 to 0.05, a standard range in statistical testing. Across both D1 and D2, the model exhibits minimal performance fluctuations, confirming the resilience of *PerfScout* to this parameter.

Penalty Coefficient. This coefficient penalizes frequent workload adjustments. Higher values favor faster convergence by promoting larger steps, while lower values enable finer control. However, overly large penalties reduce accuracy by suppressing necessary adjustments near the breaking point. A value of 0.01 offers the best trade-off between efficiency and precision across both datasets.

Risk Probability. This parameter regulates SPOT's sensitivity to detecting anomalies. A smaller value reduces false positives but may overlook critical performance shifts, while a larger value increases false alarms. Fig. 9 reveals a peak in overall performance when the risk probability is set to 10^{-5}, which we adopt as the default setting.

6 Industrial Deployment and Discussion

6.1 Workflow of *PerfScout*'s Deployment

PerfScout has been deployed within *Huawei Cloud* for nine months, supporting multiple departments with efficient and accurate performance testing. Its industrial workflow (Fig. 10) operates as follows: a test user initiates a stress test through the front-end interface; the request is routed via the HAProxy load balancer [26] to the scheduler, which allocates resources and instructs the data center to stream KPI data to the algorithm service (i.e., *PerfScout*). The algorithm service analyzes real-time KPIs, identifies performance inflection points, and returns the results to the front end. In production, *PerfScout* achieves high time efficiency: each training episode averages 7.198 s on D1, and each load-adjustment inference takes only 0.171 s. A practical issue is that high concurrency can delay KPI reporting, leading to stale or unstable inputs. *PerfScout* mitigates this by comparing successive reports and using only updated metrics, improving robustness under fluctuating latencies. Because *PerfScout* adapts workloads directly from real-time KPIs without relying on system-specific models, it transfers well across heterogeneous architectures. Although currently deployed within *Huawei Cloud*, similar service-oriented designs and monitoring pipelines in cloud and enterprise systems suggest broad applicability.

6.2 Case Study

To demonstrate the workflow and effectiveness of *PerfScout*, we present a representative case from D1, comparing it with a traditional fixed strategy that increases 50 virtual users every 60 seconds, as shown in Fig. 11. The fixed strategy applies workload increases at fixed intervals without regard for system state, lacking local stationarity detection and adaptive control. Consequently, it takes 49 minutes to complete the test, overshoots the system's breaking point, and fails to accurately identify the capacity limit, risking system instability. In contrast, *PerfScout* initially ramps up the workload linearly to gather data for local stationarity identification and breaking point identification. Once steady-state behavior is identified, it leverages a RL agent to adaptively adjust both the frequency and magnitude of workload changes based on real-time KPI feedback. When SPOT detects a surge in response time—signaling the breaking point—*PerfScout* halts further increases. It accurately identifies the SUT's capacity (green dashed line) and completes the test in 6 minutes and 18 seconds, reducing testing time by nearly 87% compared to the fixed strategy.

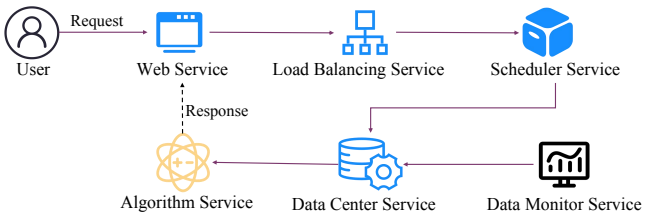

Figure 10: The workflow of *PerfScout* in *Huawei Cloud*' industrial environment.

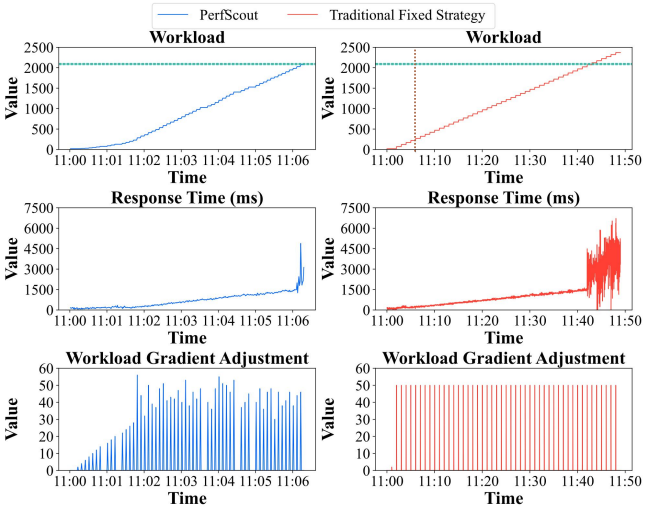

Figure 11: A representative case comparing *PerfScout* and the traditional fixed strategy. Green dashed lines indicate the SUT's maximum workload range from 2074 to 2106 users, and the brown dashed line marks when *PerfScout* completes the test.

6.3 Lessons Learned and Threats to Validity

(1) Lessons Learned. Modular design significantly enhances system flexibility and adaptability. By decomposing *PerfScout* into independent modules, each can be optimized or upgraded without affecting the overall structure, enabling seamless integration of new research or tools and dynamic module selection for diverse performance testing scenarios.

(2) Threats to Validity. A primary threat to our study is the effectiveness of the RL reward function, which critically influences strategy learning in *PerfScout*. Since performance testing scenarios often prioritize different KPIs, a fixed reward design may limit generalization. To address this, *PerfScout* supports flexible reward configuration, allowing users to tailor KPIs to specific testing contexts. This design improves adaptability and enhances the framework's applicability across diverse performance testing scenarios.

7 Related Work

7.1 White-box Workload Generator

White-box workload generators construct test workloads by analyzing source code. SLG [37] uses data flow analysis and symbolic execution to explore execution paths and identify high-load cases for capacity evaluation. SLT [35] computes load sensitivity indices from control flow graphs [3] to locate critical code blocks and generate targeted test cases. However, these approaches depend on source code access, limiting their use in black-box scenarios.

7.2 Static Black-box Workload Generator

Static black-box workload generators use predefined strategies without source code access. Traditional tools like Apache JMeter require manual workload design to probe system capacity. USENIX-ATC'08 [28] proposed a modular automated framework but still relied on manual range specification. BaaSP [30] introduced statistical methods to reduce manual intervention, while Markov4JMeter [31] modeled user behavior with Markov chains [5], and WESSBAS [32] automatically extracted workloads from session logs to generate executable test plans and performance models.

7.3 Dynamic Black-box Workload Generator

Dynamic black-box workload generators adapt test inputs at runtime based on performance feedback. Early work relies on heuristics or load sensitivity analysis to reduce manual configuration and enable online adjustment, such as ICPEC'17 [15] and DYNAMO [4]. More recent approaches adopt reinforcement learning, including RELOAD [21], Poster [22], PerfXRL [2], and SaFReL [23], which formulate workload generation as a sequential decision-making problem using variants of Q-learning or deep RL. Despite their effectiveness, most of these methods depend on manually defined performance indicators, limiting their generalizability across heterogeneous systems.

PerfScout is an RL-based dynamic black-box workload generator that operates without source-code access and adapts workload online. Unlike heuristic-based methods [4, 15], *PerfScout* learns an adaptive workload policy via PPO. Compared with prior RL-based approaches [2, 21–23], *PerfScout* integrates SPOT-based dynamic thresholding with ADF/KPSS stationarity tests to adapt adjustment intervals and more accurately detect performance breaking points. Furthermore, incorporating SPOT-estimated thresholds into the reward enables PPO to better capture subtle behavioral shifts.

8 Conclusion

This paper presents *PerfScout*, an RL-based framework for adaptive workload generation in performance testing. By combining dynamic thresholding (SPOT) with stationarity analysis (ADF, KPSS), *PerfScout* accurately identifies system breaking points and stable states. Unlike rule-based or fixed-threshold approaches, it autonomously optimizes workload adjustments via data-driven learning, reducing manual effort and expert intervention. Experiments on real-world datasets demonstrate substantial gains in testing accuracy and efficiency. *PerfScout* has been deployed and validated within *Huawei Cloud*, and will be integrated into *CodeArts PerfTest* to provide intelligent full-link performance testing to external users.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (62272249, 62302244), and the Fundamental Research Funds for the Central Universities (XXX-63253249).

References

[1] ABC123. 2016. The Failed Launch of www.HealthCare.gov. https://d3.harvard.edu/platform-rctom/submission/the-failed-launch-of-www-healthcare-gov

[2] Tanwir Ahmad, Adnan Ashraf, Dragos Truscan, and Ivan Porres. 2019. Exploratory performance testing using reinforcement learning. In *2019 45th euromicro conference on software engineering and advanced applications (seaa)*. IEEE, 156–163.

[3] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.

[4] Vanessa Ayala-Rivera, Maciej Kaczmarski, John Murphy, Amarendra Darisa, and A Omar Portillo-Dominguez. 2018. One size does not fit all: In-test workload adaptation for performance testing of enterprise applications. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. 211–222.

[5] Richard Bellman. 1957. A Markovian decision process. *Journal of mathematics and mechanics* (1957), 679–684.

[6] Uzay Çetin and Mursel Tasgin. 2020. Anomaly detection with multivariate k-sigma score using monte carlo. In *2020 5th International Conference on Computer Science and Engineering (UBMK)*. IEEE, 94–98.

[7] David A Dickey and Wayne A Fuller. 1981. Likelihood ratio statistics for autoregressive time series with a unit root. *Econometrica: journal of the Econometric Society* (1981), 1057–1072.

[8] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. 2019. Implementation matters in deep rl: A case study on ppo and trpo. In *International conference on learning representations*.

[9] Gerald D Everett and Raymond McLeod Jr. 2007. *Software testing: testing across the entire software development life cycle*. John Wiley & Sons.

[10] The Apache Software Foundation. [n. d.]. Apache JMeter. https://jmeter.apache.org

[11] Ramesh A Gopinath. 1998. Maximum likelihood modeling with Gaussian distributions for classification. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, Vol. 2. IEEE, 661–664.

[12] Samuel Hawley, M Sanni Ali, Klara Berencsi, Andrew Judge, and Daniel Prieto-Alhambra. 2019. Sample size and power considerations for ordinary least squares interrupted time series analysis: a simulation study. *Clinical epidemiology* (2019), 197–205.

[13] IBM. 2021. What is Workload Simulator. https://www.ibm.com/docs/en/wsfz-and-o/1.1.0?topic=wsim-what-is-workload-simulator

[14] Paul C Jorgensen. 2013. *Software testing: a craftsman's approach*. Auerbach Publications.

[15] Maciej Kaczmarski, Philip Perry, John Murphy, and A Omar Portillo-Dominguez. 2017. In-test adaptation of workload in enterprise application performance testing. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. 69–72.

[16] Denis Kwiatkowski, Peter CB Phillips, Peter Schmidt, and Yongcheol Shin. 1992. Testing the null hypothesis of stationarity against the alternative of a unit root: How sure are we that economic time series have a unit root? *Journal of econometrics* 54, 1-3 (1992), 159–178.

[17] Qing Lei, Weidong Liao, Yingtao Jiang, Mei Yang, and Haifeng Li. 2019. Performance and scalability testing strategy based on kubemark. In *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. IEEE, 511–516.

[18] Arefeh Mazarei, Ricardo Sousa, João Mendes-Moreira, Slavo Molchanov, and Hugo Miguel Ferreira. 2025. Online boxplot derived outlier detection. *International journal of data science and analytics* 19, 1 (2025), 83–97.

[19] J Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea. 2007. *Performance testing guidance for web applications: patterns & practices*. Microsoft press.

[20] Microsoft. 2024. Performance Testing Recommendation for Power Platform Workloads. https://learn.microsoft.com/en-us/power-platform/well-architected/performance-efficiency/performance-test

[21] Mahshid Helali Moghadam, Golrokh Hamidi, Markus Borg, Mehrdad Saadatmand, Markus Bohlin, Björn Lisper, and Pasqualina Potena. 2021. Performance testing using a smart reinforcement learning-driven test agent. In *2021 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2385–2394.

[22] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. 2020. Poster: Performance testing driven by reinforcement learning. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 402–405.

[23] Mahshid Helali Moghadam, Mehrdad Saadatmand, Markus Borg, Markus Bohlin, and Björn Lisper. 2021. An autonomous performance testing framework using self-adaptive fuzzy reinforcement learning. *Software quality journal* (2021), 1–33.

[24] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. 2016. Deep exploration via bootstrapped DQN. *Advances in neural information processing systems* 29 (2016).

[25] Peter CB Phillips and Pierre Perron. 1988. Testing for a unit root in time series regression. *biometrika* 75, 2 (1988), 335–346.

[26] Luthfan Hadi Pramono, Robby Cokro Buwono, and Yanuar Galih Waskito. 2018. Round-robin algorithm in HAProxy and Nginx load balancing performance evaluation: a review. In *2018 international seminar on research of information technology and intelligent systems (ISRITI)*. IEEE, 367–372.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[28] Piyush Shivam, Varun Marupadi, Jeff Chase, Thileepan Subramaniam, and Shivnath Babu. 2008. Cutting corners: Workbench automation for server benchmarking. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*.

[29] Alban Siffer, Pierre-Alain Fouque, Alexandre Termier, and Christine Largouet. 2017. Anomaly detection in streams with extreme value theory. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1067–1075.

[30] Alain Tchana, Bruno Dillenseger, Noel De Palma, Xavier Etchevers, Jean-Marc Vincent, Nabila Salmi, and Ahmed Harbaoui. 2013. Self-scalable benchmarking as a service with automatic saturation detection. In *Middleware 2013: ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings 14*. Springer, 389–404.

[31] André Van Hoorn, Matthias Rohr, and Wilhelm Hasselbring. 2008. Generating probabilistic and intensity-varying workload for web-based software systems. In *SPEC International Performance Evaluation Workshop*. Springer, 124–143.

[32] Christian Vögele, André van Hoorn, Eike Schulz, Wilhelm Hasselbring, and Helmut Krcmar. 2018. WESSBAS: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems. *Software & Systems Modeling* 17, 2 (2018), 443–477.

[33] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8 (1992), 279–292.

[34] Liquid Web. 2023. 21 Websites that Crashed after Going Viral. https://www.liquidweb.com/blog/biggest-website-crash

[35] Cheer-Sun D Yang and Lori L Pollock. 1996. Towards a structural load testing tool. *ACM SIGSOFT Software Engineering Notes* 21, 3 (1996), 201–208.

[36] Junzi Zhang, Jongho Kim, Brendan O'Donoghue, and Stephen Boyd. 2021. Sample efficient reinforcement learning with REINFORCE. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 10887–10895.

[37] Pingyu Zhang, Sebastian Elbaum, and Matthew B Dwyer. 2011. Automatic generation of load tests. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 43–52.

[38] Shenglin Zhang, Xiao Xiong, Mengyao Li, Yongqian Sun, Yongxin Zhao, Xia Chen, Bowen Deng, and Dan Pei. 2024. Auto-PIP: Real-time Identification of Critical Performance Inflection Points in Software Stress Testing. In *2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 25–30.