Investigating the Impact of Isolation on Synchronized Benchmarks

Nils Japke
TU Berlin
Scalable Software Systems Research Group
Berlin, Germany
nj@3s.tu-berlin.de

Furat Hamdan
TU Berlin
Scalable Software Systems Research Group
Berlin, Germany
fuha@3s.tu-berlin.de

Diana Baumann
TU Berlin
Scalable Software Systems Research Group
Berlin, Germany
diba@3s.tu-berlin.de

David Bermbach
TU Berlin
Scalable Software Systems Research Group
Berlin, Germany
db@3s.tu-berlin.de

Abstract

Benchmarking in cloud environments suffers from performance variability from multi-tenant resource contention. Duet benchmarking mitigates this by running two workload versions concurrently on the same VM, exposing them to identical external interference. However, intra-VM contention between synchronized workloads necessitates additional isolation mechanisms.

This work evaluates three such strategies: *cgroups* and *CPU pinning*, *Docker containers*, and *Firecracker MicroVMs*. We compare all strategies with an unisolated baseline experiment, by running benchmarks with a duet setup alongside a noise generator. This noise generator "steals" compute resources to degrade performance measurements.

All experiments showed different latency distributions while under the effects of noise generation, but results show that process isolation generally lowered false positives, except for our experiments with Docker containers. Even though Docker containers rely internally on *cgroups* and *CPU pinning*, they were more susceptible to performance degradation due to noise influence. Therefore, we recommend to use process isolation for synchronized workloads, with the exception of Docker containers.

CCS Concepts

• **General and reference** → **Measurement**; **Performance**; • **Software and its engineering** → **Software performance**; **Software testing and debugging**.

Keywords

software performance, cloud benchmarking, duet benchmarking, process isolation

ACM Reference Format:
Nils Japke, Furat Hamdan, Diana Baumann, and David Bermbach. 2025. Investigating the Impact of Isolation on Synchronized Benchmarks. In *2025 IEEE/ACM 18th International Conference on Utility and Cloud Computing (UCC '25), December 1–4, 2025, Nantes, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3773274.3774703

1 Introduction

Many applications and services nowadays rely on the cloud, which makes the performance of cloud applications particularly interesting to software developers. However, cloud computing platforms suffer from unpredictable performance [21]. Many techniques have been developed to overcome this problem, such as *Randomized Multiple Interleaved Trials* (RMIT) [1, 2] and *duet benchmarking* [5, 6].

RMIT overcomes the unstable cloud performance by increasing the number of iterations across levels (such as replicating the benchmark using different instances), and shuffling them. This, however, strongly increases the overall experiment time and cost. Duet benchmarking has become popular in research as it aims to overcome this problem in regression testing scenarios. For a regression test, two versions of the *system under test* (SUT) are compared against each other on the same *virtual machine* (VM) in the cloud. However, because the two SUT versions might influence each other depending on, e.g., CPU scheduling, strong isolation of the processes is necessary.

This leads to the following questions:

- How strongly are measurements affected by other processes running on the same VM?
- Which different process isolation techniques can help to ensure high measurement quality?

The necessity of running the SUT in a controlled environment during a benchmark is generally understood and recommended in order to remove any outside influence on the measurements that we can directly account for. Prior studies have already focused on quantifying the performance loss to "noisy neighbors", which often happens in cloud environments [29]. For a synchronized benchmarking experiment such as duet benchmarking, however, the exact influence on measurement results by outside influence has not been established yet.

In order to answer the research questions posed above, we provide the following contributions:

- We develop a noise generating application that attempts to influence performance measurements of a benchmark by using a high amount of CPU time.

- We run benchmarking experiments using duet benchmarking with a testbed application first presented by Japke et al. [19] to compare the results with different process isolation techniques at different noise levels.
- In particular, we compare the following isolation setups:
  (1) No isolation
  (2) Resource assignment using *cgroups* and *CPU pinning* of the main SUT process
  (3) Isolating the SUT in a Docker container (including *CPU pinning* of the container on the host machine)
  (4) Isolating the SUT in a Firecracker MicroVM (including *CPU pinning* of the MicroVM on the host machine)

We find that noise levels are strongly apparent when using either no isolation or Docker containers, but are not as clearly visible when using *cgroups* and *CPU pinning* or MicroVMs. Additionally, we find that when calculating the relative differences of median performance between the SUT versions and using state-of-the-art regression detection techniques, all setups mostly correctly report "no change", as expected in an A/A test, except when using Docker containers, which had a higher amount of false positives, even without any noise present. Therefore, we conclude that the duet benchmarking technique can work even at higher noise levels to provide largely correct results. However, stronger isolation still helps to reduce the amount of false positives.

2 Background

In this section, we introduce the necessary scientific background that this paper relies on.

Regression Testing. Regression testing refers to comparing two different versions of a SUT. This is typically done by running the same benchmark workload separately against both versions while collecting performance measurements (e.g., latency). Then, both results are compared using statistical methods, such as comparing confidence intervals or using hypothesis tests.

Cloud Performance Variability. The performance of VMs on cloud platforms typically varies, both over time and even across different VMs of the same type [21]. This creates challenges when trying to run benchmarks of applications in cloud environments as different performance characteristics of the platform can bias measurements.

Several techniques have been proposed to deal with these challenges. Here, we focus on the two following techniques.

Duet Benchmarking aims to deal with cloud performance variability without needing additional infrastructure [5, 6]. The central idea of duet benchmarking is that for benchmarking experiments with comparative results, such as when performing a regression test between two versions of an SUT, we can perform both benchmarking experiments simultaneously on the same VM. This ensures that any performance variability affects both SUTs in the exact same way, so that we can still measure the relative performance difference between both SUTs. Using duet benchmarking, we cannot obtain any *absolute* performance metrics that are usable outside of such a comparison, however, as the single VM might bias the results due to cloud performance variability. Duet benchmarking also contains an additional implementation concern, i.e., isolating both SUT processes from one another to ensure that both are treated fairly and have access to the same amount of resources during the entire experiment.

Duet benchmarking has been used in scientific studies on application benchmarks and microbenchmarks [5, 6, 13, 19]. The main advantage over RMIT is the significantly lower infrastructure cost and experiment runtime, but as explained above duet benchmarking is limited to comparative benchmarks such as regression tests.

Process Isolation. We look at the following strategies for process isolation under a Linux host system:

- *CPU pinning and cgroups:* With the `taskset` utility, we can set a process to only execute on one CPU core. We combine this with *cgroups*, which are a mechanism of the Linux kernel to assign different resource restrictions to processes, including CPU, memory, and I/O.
- *Docker Containerization:* Containers are a lightweight method for isolating processes and libraries in their own separate environment. They build on *cgroups* and other mechanisms of the Linux kernel to achieve this, and still run using the host kernel.
- *Firecracker MicroVMs:* Firecracker is a Virtual Machine Manager for MicroVMs, which are a lightweight variant of VMs. MicroVMs use their own *guest kernel*, and are, therefore, a complete Linux subsystem running on the host machine.

3 Study Design

In this section, we describe our study design.

3.1 Benchmarking Testbed Application

The application used as the SUT in this study is a lightweight service called `flight-booking-service`. Originally developed by Japke et al. [19] for investigating the capabilities of application-level and microbenchmarks in detecting performance regressions, this service has been adapted to demonstrate the effect of isolation strategies in a duet benchmarking environment.

The application is written in Go and replicates core functionalities of a typical flight booking system. It includes endpoints for searching flights, retrieving available seats, and booking flights. It exposes the following HTTP API: **GET** for `/destinations`, `/flights`, and `/flights/flight_id/seats`, as well as **POST** for `/bookings`.

As part of the testbed application, Japke et al. also provide an application benchmark, and a framework to run this benchmark for a regression test using *duet benchmarking* [19]. The benchmark includes two different workload scenarios:

S1 – Flight Search: This scenario mimics a user searching for available flights. It begins with a request for location airports (`GET /destinations`). A random location is selected, followed by a query for departing flights from there (`GET /flights?from=$airport`).

S2 – Flight Search and Booking: This extended scenario adds a booking step. After selecting a flight, the user retrieves seat availability (`GET /flights/$id/seats`) and proceeds to reserve seats by submitting a booking request (`POST /bookings`).

To simulate realistic usage patterns, the two scenarios are executed with different weights. The k6 configuration defines 50 virtual users

(VUs) executing 2,000 iterations each for S1 and 10 VUs executing 380 iterations each for S2. This results in approximately 103,800 search requests and 3,800 booking requests per benchmark run.

Each benchmark runs for roughly 30 minutes. Since the focus lies on performance characteristics rather than correctness, occasional timeouts or HTTP errors are tolerated as long as they do not systematically affect measurement outcomes.

3.2 Noise Generator

To simulate external interference and evaluate noise resilience, we use a custom noise generator written in Java. It supports the generation of CPU noise by running multiple threads that execute compute-bound loops. Users can define the number of threads, maximum CPU usage, and runtime duration. If a CPU usage cap is set, threads periodically sleep to throttle consumption.

3.3 Performance Change Detection

For all experiments, we run duet benchmarking with different process isolation strategies in a regression test using the same version of the SUT twice (i.e., an A/A test). This follows the established *duet benchmarking* methodology [5, 6, 13, 19, 24]. In order to analyze the results of each experiment regarding measured performance regressions, we use two different statistical methods.

For the first method, *confidence interval overlap*, we first calculate the *relative change* by dividing the median latency of version 2 by the median latency of version 1. Then, we construct a confidence interval (CI) around the measured relative change with bootstrapping, a non-parametric method to calculate confidence intervals using resampling (confidence level 99%, 10,000 bootstrap iterations). Finally, we compare whether the CI overlaps 1, which indicates no change. If it does not overlap 1, we report a significant performance difference.

For the second method, we compare the latency values of both versions using the Wilcoxon rank-sum test. This is a non-parametric test, which compares the median of two different samples. We use a confidence level of 99%, meaning if the p value is lower than 0.01, we report a statistically significant performance difference.

In order to quantify the influence of noise, we apply both techniques to result data from the noise phase and non-noise phase separately, as well as all data overall from one experiment. The separation helps us identify the impact of noise influence, while practitioners would only observe the overall data.

4 Evaluation

In this section, we present the experimental setup and results of evaluating isolation strategies in duet benchmarking within cloud environments. The goal is to assess how effectively these strategies reduce performance interference.

4.1 Experiment Setup

We host all experiments using Google Cloud. Each VM uses the `n2-custom-6-4096` configuration, with 6 virtual CPUs and 4 GB of memory. Figure 1 shows the general experiment setup. One VM hosts the SUT instances, while a second VM runs the benchmarking clients. This separation prevents competition for resources between

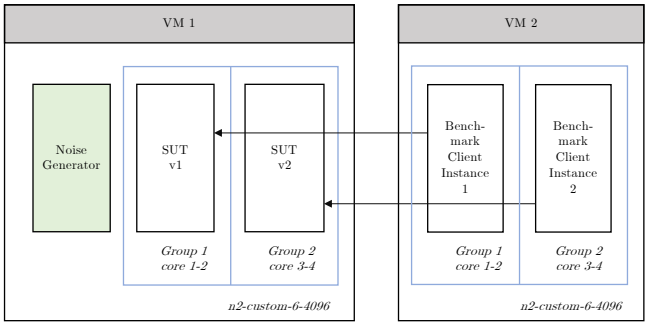

Figure 1: An illustration of the general experiment setup. The different *groups* on both VMs represent the different isolation strategies, i.e., using *cgroups* and *CPU pinning*, using *Docker* with similar functionality, and using *Firecracker MicroVMs* with similar functionality. In the baseline experiment, we use no isolation strategy, which means that this blue box does not exist for that particular setup.

the SUT instances and the workload generator. Both the SUT versions and the two instances of the benchmarking client are isolated using the same strategy that is used for the particular experiment (potentially unisolated in the baseline experiment). The noise generator is deliberately left unisolated, as its purpose is to introduce system-level interference that ideally affects both SUT instances equally. We run four experiments: (1) baseline – no isolation, (2) *cgroups* and *CPU pinning isolation*, (3) *Docker container* isolation, and (4) *Firecracker MicroVM* isolation. Each experiment includes six configurations: *0, 3, 6, 20, 40, and 60 noise CPU threads*. We run each experiment for approx. 30 minutes. The noise generator always starts deterministically *200 seconds* after experiment start, and shuts down *500 seconds* after experiment start. We also remove the first and last 60 seconds of measurements as warmup and cooldown periods.

We use A/A testing, i.e., we compare the same version of the SUT to itself in a regression test. This helps us to identify the influence of the noise generator, as we know the ground truth of the regression test is 0% performance deviation. This means that any observed performance differences are caused by environmental variability or insufficient isolation.

We provide all code and data artifacts, as well as an online appendix in our replication package.[1] Due to the large amount of result data, we only show highly aggregated plots in this section. For full result tables and supplemental plots, see our online appendix.

4.2 Results & Findings

For each experiment, we provide results in the form of boxplots of the resulting distributions of relative changes in request latency that were measured during the experiments (see Figures 2 to 5). We show the distributions for each of the four endpoints of the `flight-booking-service` for all six configurations of CPU noise threads. Each distribution is further split into the following categories: (1) "no noise", which only includes data outside of the phase

[1] https://github.com/njapke/isolation-replication-package

with the noise generator active, (2) "only noise", which only includes data inside the phase with the noise generator active, and (3) "all data", which includes all data from the particular experiment. For *0 threads*, there is no noise generation even during the noise phase. A potential benchmarking practitioner would only observe "all data", which might include strong impact of noise. Since all experiments were A/A tests, the relative change should always be 0%. The distributions of relative changes can therefore give us insight into whether that observation is disturbed by the noise generation.

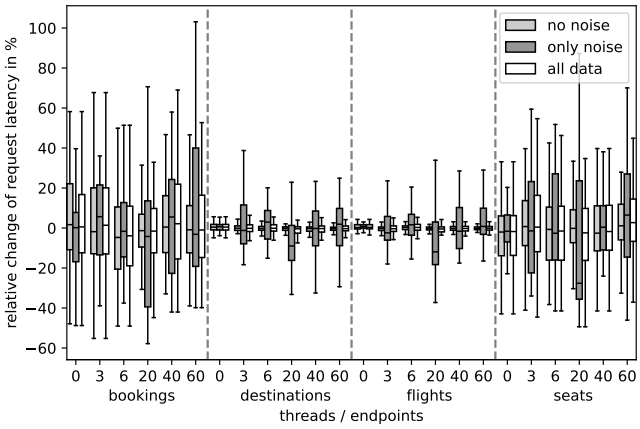

Figure 2: Boxplots for all results of *Experiment 1 – Baseline*. For each endpoint, we show the distribution of relative changes in request latency throughout the experiment for all configurations. We also show the distribution outside the noise generation (*no noise*), during noise generation (*only noise*), and all result data together (*all data*).

Experiment 1 – Baseline. Figure 2 shows the results of the baseline experiment. We observe that for all four endpoints, the boxplots mostly correctly show a median close to 0%, but the distributions for "only noise" are much wider (especially notable for *destinations* and *flights*). Further comparing the medians of all three boxplots per experiment, the "no noise" and "all data" medians tend to lie closer together, while the median of "only noise" is sometimes far off from 0% (e.g., *seats* at 20 threads). This indicates that the noise generation strongly degrades the capability to correctly assess performance changes, but that these effects even out with the data that is unaffected by noise.

Experiment 2 – cgroups and CPU pinning. Figure 3 shows the results of the *cgroups* and *CPU pinning* experiment. In contrast to the baseline experiment, we can see that the distributions during the noise phase are generally closer to the distributions outside the noise phase. There is, however, a strong outlier at 3 threads of noise for the *destinations*, *flights*, and *seats* endpoints, where the noise effect is much stronger than during all other configurations, which have higher noise generation. This could be caused by CPU scheduling, which might behave differently for a higher number of threads within the noise process.

Experiment 3 – Docker containers. Figure 4 shows the results of the Docker container experiment. Here, we can observe the same

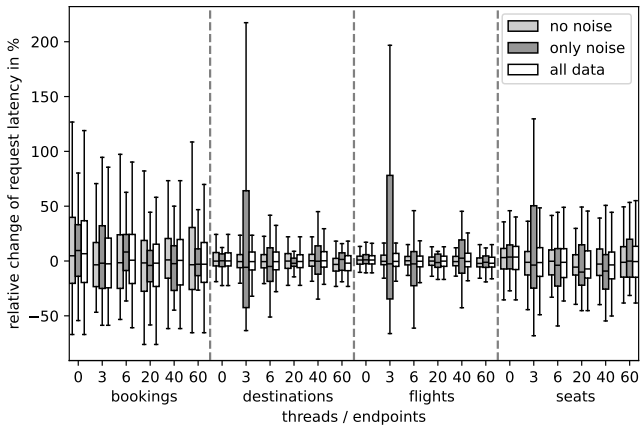

Figure 3: Boxplots for all results of *Experiment 2 – cgroups and CPU pinning*. For each endpoint, we show the distribution of relative changes in request latency throughout the experiment for all configurations. We also show the distribution outside the noise generation (*no noise*), during noise generation (*only noise*), and all result data together (*all data*).

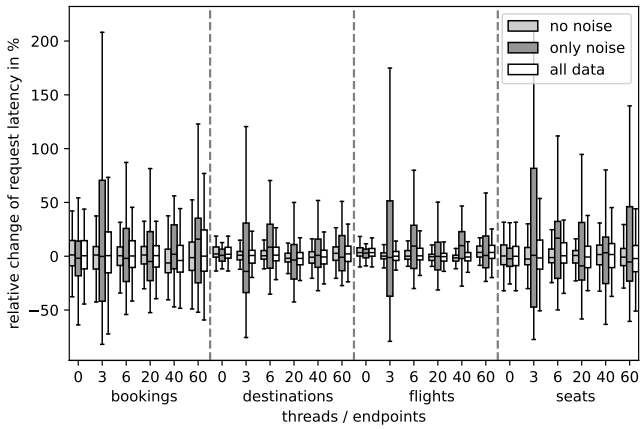

Figure 4: Boxplots for all results of *Experiment 3 – Docker containers*. For each endpoint, we show the distribution of relative changes in request latency throughout the experiment for all configurations. We also show the distribution outside the noise generation (*no noise*), during noise generation (*only noise*), and all result data together (*all data*).

effect for 3 threads of noise as in experiment 2, however, the distributions during the noise phase remain wider in all configurations. This suggests that, unlike for regular *cgroups* and *CPU pinning*, the resource allocation for Docker containers remains more contested, despite Docker using the same mechanisms for resource assignment.

Experiment 4 – Firecracker MicroVMs. Figure 5 shows the results of the Firecracker MicroVM experiment. Here, we can observe the same effect at 6 threads of noise that we encountered in experiment 2 and 3 at 3 threads of noise. Overall, Firecracker MicroVMs show lower effects due to noise (note the different y-axis scaling). This

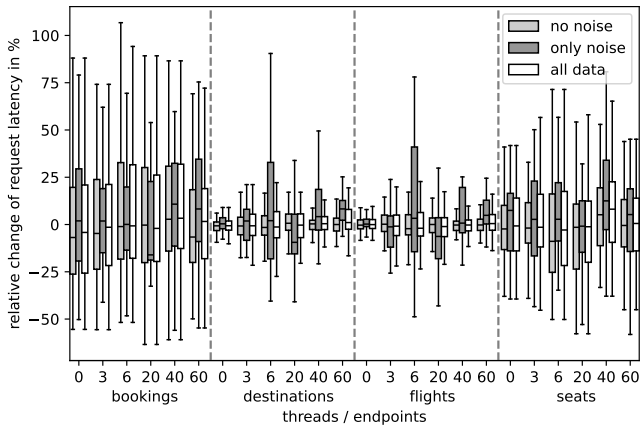

Figure 5: Boxplots for all results of *Experiment 4 – Firecracker MicroVMs*. For each endpoint, we show the distribution of relative changes in request latency throughout the experiment for all configurations. We also show the distribution outside the noise generation (*no noise*), during noise generation (*only noise*), and all result data together (*all data*).

suggests that the increased process isolation mitigates more of the effects due to noise.

False Positives in Performance Change Detection. Now, we evaluate how state-of-the-art performance change detection techniques are influenced by the noise generation. We calculate significant performance changes according to CI overlap and the Wilcoxon rank-sum test, as explained in §3.3, and show all false positives (FP) measured in all experiments in Table 1. More detailed result tables for all experiments are available in the online appendix of our replication package.

When comparing experiments 2 and 4 to the baseline experiment, we can see that the employed isolation techniques strictly decrease FPs. This is different, however, for experiment 3, i.e., Docker container isolation, where even outside of the noise phase both performance change detection methods show several statistically significant performance changes. This is in contrast to experiment 2, since Docker also builds on *cgroups* and *CPU pinning* for isolating processes. This leaves experiment 3 as the only experiment, where performance changes for "all data" increased compared to the baseline. As such, we do not recommend to use Docker for isolating processes in benchmarks, and to instead use either of the other isolation strategies.

5 Discussion

In this section, we discuss the limitations of this study.

Narrow Scope of Simulated Noise: The experiments in this study focused solely on CPU-based contention using thread saturation. While this form of interference is highly relevant in multi-tenant cloud environments, it does not represent the full spectrum of real-world resource contention. Cloud-native applications often face performance degradation due to memory pressure, I/O bottlenecks, or network latency.

Generalizability Across Cloud Platforms: The evaluation was conducted exclusively on Google Cloud using `n2-custom-6-4096` instances. However, isolation strategies may behave differently across cloud providers due to variations in hypervisor technologies (e.g., KVM on Google Cloud versus Nitro on AWS), CPU architectures (Intel vs. AMD), and resource allocation policies. For example, Firecracker MicroVMs are optimized for AWS Nitro hardware, and their performance characteristics might vary when deployed on KVM-based hosts like Google Cloud. The fixed VM type also limits generalizability, as it remains unclear how isolation strategies scale with different instance sizes or resource configurations.

Application-Specific Limitations: All experiments were conducted using a single application: the `flight-booking-service` testbed. While its architecture represents common microservice patterns, and it includes both stateful (write) and stateless (read) operations, it is nonetheless a simplified service with minimal external dependencies and in-memory storage. This limits its resemblance to complex production systems, which often involve distributed data stores, API integrations, and asynchronous processing. Isolation strategies may behave differently in applications with more intensive disk I/O, network activity, or concurrency challenges. Additionally, the nature of duet benchmarking itself may not be suitable for all system types. Applications with multiple necessary components (e.g., distributed applications) may not work well with isolated execution. Duet benchmarking assumes independent execution of multiple SUT instances, which is not always feasible in real-world architectures. The reliance on Go as the implementation language may also introduce language-specific artifacts due to Go's concurrency model and garbage collection behavior. Broader validation across different languages and system architectures is needed to verify the generalizability of these findings.

Limitations of Application Workload: The benchmark workload involved 50 VU executing scripted interactions. While this setup generated measurable performance differences, it does not capture the scale or variability of real-world traffic. The absence of traffic spikes, irregular user patterns, or bursty requests likely underrepresents tail latency and peak stress scenarios. As a result, the impact of isolation on outlier behavior (e.g., high-percentile latency) may be underestimated.

Statistical and Methodological Constraints: The performance comparisons relied on median response ratios and bootstrap confidence intervals. While statistically sound, this method involved aggregating data to one median value per second, which may obscure high-frequency fluctuations or fine-grained instability. Additionally, the 99% confidence level, while conservative, may have overlooked subtle but relevant changes.

6 Related Work

Detecting and Quantifying Performance Changes. Benchmarking requires repeatability and consistent results under identical conditions. However, this is difficult in cloud environments due to dynamic resource allocation, hardware heterogeneity, and interference from noisy neighbors [4, 5, 27, 28]. A common approach is to use RMIT to repeat and shuffle iterations of a particular workload [1, 2]. This increases the repeatability of results in the presence

Table 1: Overview of all measured false positives (FP) across all experiments. During each experiment, we compared 6 configurations × 4 endpoints = 24 observations. We show the number of observations, split by "no noise", "only noise", and "all data", which were reported as having a *performance change* (i.e., v2 is either slower or faster than v1) by the CI overlap and Wilcoxon rank-sum test. All 24 − *n* unlisted observations were correctly reported as *no performance change*.

Experiment	FP by CI Overlap			FP by Wilcoxon Rank-Sum Test		
	no noise	only noise	all data	no noise	only noise	all data
1 – Baseline	0	3	1	1	4	1
2 – cgroups + CPU Pinning	0	0	0	1	0	1
3 – Docker Containers	4	1	3	6	3	3
4 – Firecracker MicroVMs	0	2	0	0	4	0

of noise, but also increases the experiment time and, therefore, cost. On the platform side, cloud providers can use host-level strategies like interference-aware scheduling and VM migration. For instance, Delimitrou and Kozyrakis propose an interference-aware scheduler that optimizes server consolidation [9], while Roytman et al. use cache access behavior to guide VM placement [25]. When using public clouds, however, practitioners need to use special analysis methods to obtain good results on performance regressions. Amannejad et al. propose *Collaborative Response Time Estimation*, a machine learning approach using collaborative filtering to predict expected transaction times [3]. Deviations from these estimates signal performance interference and enable dynamic mitigation strategies like load redistribution. Daly et al. apply the E-Divisive means algorithm to identify changepoints in continuous integration systems to enhance performance regression detection [8]. Fleming et al. build on this by replacing the permutation-based testing in E-Divisive with a Student's *t*-test in their tool *Hunter*, which improves accuracy and runtime [10]. Ordozgoiti et al. employ deep CNNs to identify noisy neighbors from time-series performance metrics, achieving high classification accuracy in real cloud environments [22]. There also exist approaches to determine the optimal execution time for a benchmarking experiment in the presence of noise [14, 15, 17, 18, 20]

Performance Comparisons of Virtualization and Isolation Methods. Ghatrehsamani et al. compare CPU pinning effects across containers, VMs, and bare-metal platforms [11]. Their results show that CPU pinning reduces overhead significantly, especially in I/O-intensive applications. Containers with more pinned cores exhibited better performance and lower overhead, and pinning helped mitigate resource tracking overhead within containerized environments. Podzimek et al. study CPU pinning in colocated workloads and find that unconventional pinning configurations improve energy efficiency, particularly under partial background loads [23]. Their findings support dynamically adapting pinning strategies based on workload characteristics and system load. Docker's performance has also been widely studied, e.g., comparing Docker against bare-metal systems [7, 26]. Grambow et al. focus on Docker's impact on database benchmarking [12]. Their findings suggest Docker can introduce unpredictable latency and throughput degradation, particularly for write-heavy workloads. In the context of MicroVMs, Wang compares Firecracker with Docker and QEMU-based virtual machines across multiple metrics [30]. Firecracker offered a middle ground: better isolation and security than containers but without the full overhead of traditional VMs. However, Firecracker did not consistently outperform Docker, especially in startup time and I/O throughput. Similarly, Hebbar et al. use Sysbench and RAMSpeed to benchmark Firecracker, Docker, and bare-metal systems [16]. Their results indicate that Firecracker matches Docker in CPU and memory performance, but lags in file I/O. They conclude that while Firecracker fulfills its design goals in terms of isolation, further improvements are needed to bridge remaining performance gaps.

7 Conclusion

In this work, we developed a noise generating application, and ran benchmarking experiments comparing three different process isolation techniques. We found that noise strongly affects performance measurements, but when analyzing the relative change of both SUT versions, duet benchmarking mostly provides correct results. Furthermore, we found that process isolation still decreased the amount of false positives, except when using Docker containers.

References

[1] Ali Abedi and Tim Brecht. 2017. Conducting repeatable experiments in highly variable cloud computing environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 287–292.

[2] Ali Abedi, Andrew Heard, and Tim Brecht. 2015. Conducting Repeatable Experiments and Fair Comparisons using 802.11n MIMO Networks. *ACM SIGOPS Operating Systems Review* 49, 1 (Jan. 2015), 41–50. doi:10.1145/2723872.2723879

[3] Yasaman Amannejad, Diwakar Krishnamurthy, and Behrouz Far. 2015. Managing Performance Interference in Cloud-Based Web Services. *IEEE Trans. on Netw. and Serv. Manag.* 12, 3 (Sept. 2015), 320–333. doi:10.1109/TNSM.2015.2456172

[4] David Bermbach, Erik Wittern, and Stefan Tai. 2017. *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective*. Springer, Cham, Switzerland.

[5] Lubomír Bulej, Vojtěch Horký, Petr Tuma, François Farquet, and Aleksandar Prokopec. 2020. Duet Benchmarking: Improving Measurement Accuracy in the Cloud. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ACM. doi:10.1145/3358960.3379132

[6] Lubomír Bulej, Vojtech Horky, and Petr Tůma. 2019. Initial Experiments with Duet Benchmarking: Performance Testing Interference in the Cloud. 249–255. doi:10.1109/MASCOTS.2019.00035

[7] Emiliano Casalicchio and Vanessa Perciballi. 2017. Measuring docker performance: What a mess!!!. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. 11–16.

[8] David Daly, William Brown, Henrik Ingo, Jim O'Leary, and David Bradford. 2020. The use of change point detection to identify software performance regressions in a continuous integration system. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 67–75.

[9] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices* 48, 4 (2013), 77–88.

[10] Matt Fleming, Piotr Kolaczkowski, Ishita Kumar, Shaunak Das, Sean McCarthy, Pushkala Pattabhiraman, and Henrik Ingo. 2023. Hunter: Using Change Point Detection to Hunt for Performance Regressions. In *Proceedings of the 2023 ACM/SPEC*

International Conference on Performance Engineering. 199–206.

[11] Davood Ghatrehsamani, Chavit Denninnart, Josef Bacik, and Mohsen Amini Salehi. 2020. The art of cpu-pinning: Evaluating and improving the performance of virtualization and containerization platforms. In *Proceedings of the 49th International conference on parallel processing*. 1–11.

[12] Martin Grambow, Jonathan Hasenburg, Tobias Pfandzelter, and David Bermbach. 2019. Is it safe to dockerize my database benchmark?. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol, Cyprus) *(SAC '19)*. Association for Computing Machinery, New York, NY, USA, 341–344. doi:10.1145/3297280.3297545

[13] Martin Grambow, Denis Kovalev, Christoph Laaber, Philipp Leitner, and David Bermbach. 2023. Using Microbenchmark Suites to Detect Application Performance Changes. *IEEE Transactions on Cloud Computing* 11, 3 (2023), 2575–2590. doi:10.1109/TCC.2022.3217947

[14] Martin Grambow, Christoph Laaber, Philipp Leitner, and David Bermbach. 2021. Using application benchmark call graphs to quantify and improve the practical relevance of microbenchmark suites. *PeerJ Computer Science* 7:e548 (2021). doi:10.7717/peerj-cs.548

[15] Sen He, Glenna Manns, John Saunders, Wei Wang, Lori Pollock, and Mary Lou Soffa. 2019. A Statistics-Based Performance Testing Methodology for Cloud Applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 188–199. doi:10.1145/3338906.3338912

[16] Anushka Hebbar, Shree Thavarekere, Anusha Kabber, KV Subramaniam, et al. 2022. Performance Comparison of Virtualization Methods. In *2022 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. IEEE, 43–47.

[17] Nils Japke, Martin Grambow, Christoph Laaber, and David Bermbach. 2025. μOpTime: Statically Reducing the Execution Time of Microbenchmark Suites Using Stability Metrics. *ACM Transactions on Software Engineering and Methodology* 34, 8 (2025), 26 pages. doi:10.1145/3715322

[18] Nils Japke, Sebastian Koch, Helmut Lukasczyk, and David Bermbach. 2025. Towards an Optimized Benchmarking Platform for CI/CD Pipelines. In *Proceedings of the 13th IEEE International Conference on Cloud Engineering* (Rennes, France) *(IC2E '25)*. IEEE, New York, NY, USA, 36–41. doi:10.1109/IC2E65552.2025.00010

[19] Nils Japke, Christoph Witzko, Martin Grambow, and David Bermbach. 2023. The Early Microbenchmark Catches the Bug – Studying Performance Issues Using Micro- and Application Benchmarks. In *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing (UCC '23)*. ACM, 1–10. doi:10.1145/3603166.3632128

[20] Christoph Laaber, Stefan Würsten, Harald C. Gall, and Philipp Leitner. 2020. Dynamically reconfiguring software microbenchmarks: reducing execution time without sacrificing result quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 989–1001. doi:10.1145/3368089.3409683

[21] Philipp Leitner and Jürgen Cito. 2016. Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Technol.* 16, 3, Article 15 (April 2016), 23 pages. doi:10.1145/2885497

[22] Bruno Ordozgoiti, Alberto Mozo, Sandra Gómez Canaval, Udi Margolin, Elisha Rosensweig, and Itai Segall. 2017. Deep convolutional neural networks for detecting noisy neighbours in cloud infrastructure. *COSTAC 2017* 59 (2017).

[23] Andrej Podzimek, Lubomír Bulej, Lydia Y Chen, Walter Binder, and Petr Tuma. 2015. Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 1–10.

[24] Tim C. Rese, Nils Japke, Sebastian Koch, Tobias Pfandzelter, and David Bermbach. 2024. Increasing Efficiency and Result Reliability of Continuous Benchmarking for FaaS Applications. In *Proceedings of the 12th IEEE International Conference on Cloud Engineering* (Paphos, Cyprus) *(IC2E '24)*. IEEE, New York, NY, USA, 93–100. doi:10.1109/IC2E61754.2024.00017

[25] Alan Roytman, Aman Kansal, Sriram Govindan, Jie Liu, and Suman Nath. 2013. Algorithm design for performance aware VM consolidation. *Microsoft Res., Redmond, WA, USA, Tech. Rep.: MSR-TR-2013-28* (2013).

[26] Pankaj Saha, Angel Beltre, Piotr Uminski, and Madhusudhan Govindaraju. 2018. Evaluation of docker containers for scientific workloads in the cloud. In *Proceedings of the Practice and Experience on Advanced Research Computing: Seamless Creativity*. 1–8.

[27] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 460–471.

[28] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. 2020. Is big data performance reproducible in modern cloud networks?. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 513–527.

[29] Simon Volpert, Benjamin Erb, Georg Eisenhart, Daniel Seybold, Stefan Wesner, and Jörg Domaschka. 2023. A Methodology and Framework to Determine the Isolation Capabilities of Virtualisation Technologies. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering* (Coimbra, Portugal) *(ICPE '23)*. Association for Computing Machinery, New York, NY, USA, 149–160. doi:10.1145/3578244.3583728

[30] Zicheng Wang. 2021. Can "micro VM" become the next generation computing platform?: Performance comparison between light weight Virtual Machine, container, and traditional Virtual Machine. In *2021 IEEE International Conference on Computer Science, Artificial Intelligence and Electronic Engineering (CSAIEE)*. IEEE, 29–34.