

Memory Reclamation Behavior Under cgroups v2 Versus Legacy cgroups: A Measurement Framework for Containerized Workloads

Nidhi Chahal¹, Tinu Anand Kumar^{2,*}, Reshika²

Abstract

Linux control groups (cgroups) are the kernel mechanism that lets container runtimes such as Docker, containerd, and CRI-O enforce per-container CPU, memory, and I/O limits, and the transition from the legacy cgroups v1 hierarchy to the unified cgroups v2 hierarchy changed several aspects of how the memory controller signals and responds to reclaim pressure most notably through the introduction of memory.low/memory.high soft-limit semantics and system-wide Pressure Stall Information (PSI). Kernel documentation and a small number of practitioner benchmarks describe these interface-level changes, and separate networking-focused studies have measured cgroups v2's lower CPU-instruction overhead relative to v1, but we are not aware of published work that measures, under controlled and reproducible conditions, how the two hierarchies actually behave when several memory-constrained containers compete for host memory on the same node—the scenario that determines achievable container density in practice. This paper makes two contributions. First, it synthesizes the documented and practitioner-reported differences between the v1 and v2 memory controllers into a specific set of hypotheses about reclaim behavior under contention (soft-limit responsiveness, page-cache versus anonymous-memory eviction balance, and OOM-kill fairness across co-located containers). Second, it specifies a controlled, reproducible benchmark protocol using a memory-pressure workload generator, a fixed host configuration, and a defined density-scaling procedure for testing these hypotheses directly, and it reports the results section as an execution plan with the intended tables rather than fabricated benchmark numbers, since the benchmark has not yet been run on physical or virtualized hardware at the time of writing. The protocol is designed so that any reader with access to a Linux host supporting both hierarchies can execute it and populate the results independently.

Keywords: cgroups, memory management, container runtime, Linux kernel, resource isolation, container density, reclaim pressure, PSI

*Author For Correspondence

Tinu Anand Kumar
E-mail: tinuanand336699@gmail.com

¹Assistant Professor, Electronics and Communication Engineering, Chandigarh Engineering College-CGC, Landran, Mohali, Chandigarh, India

²Student, Electronics and Communication Engineering, Chandigarh Engineering College-CGC, Landran, Mohali, Chandigarh, India.

Received Date: September 12, 2026
Accepted Date: September 17, 2026
Published Date: September 20, 2026

Citation: Nidhi Chahal, Tinu Anand Kumar, Reshika. Memory Reclamation Behavior Under cgroups v2 Versus Legacy cgroups: A Measurement Framework for Containerized Workloads. Recent Trends in Parallel Computing. 2026; 13(2): 27–34p.

INTRODUCTION

Container-based workloads share a single kernel and a single pool of physical memory across many isolated processes, and the mechanism that makes this sharing survivable—rather than a scramble in which one greedy container starves the rest—is the cgroups subsystem. Every major container runtime translates a user-specified memory limit into a cgroup memory controller configuration, and every container orchestrator's bin-packing and eviction decisions ultimately rest on assumptions about how that controller behaves when several containers are near their limits simultaneously.

The Linux kernel community redesigned this subsystem substantially in the move from cgroups

v1 to the unified cgroups v2 hierarchy, made non-experimental in kernel 4.5 and adopted as the default in most current distributions. Beyond the well-documented change to a single unified hierarchy (replacing v1's independent per-controller trees), the memory controller itself gained new interface files—`memory.low` and `memory.high` for soft protection and soft throttling, in addition to the existing hard `memory.max` limit—and cgroup-scoped Pressure Stall Information (PSI) counters that expose, per cgroup, how much time tasks spent stalled waiting for memory. Kernel documentation describes the semantics of these interfaces precisely at the level of a single cgroup, and a small number of independent practitioners and academic benchmarks have measured narrower performance properties of the v1-to-v2 transition. For instance, network-latency studies at the Technical University of Munich found cgroups v2 to be measurably more CPU-efficient than v1 in a low-latency networking context, and a large-scale OpenShift density study comparing the v1/runc and v2/crun combinations at up to 2,500 pods per node found broadly similar pod-scheduling latency between the two, with v2 modestly faster at the tail [1].

What remains comparatively unstudied, at least in published and reproducible form, is memory reclaim behavior, specifically under multi-container contention: when several containers on the same node simultaneously approach their memory limits, does the v2 controller's addition of soft-limit semantics and PSI-based signaling change which container gets reclaimed from first, how much page cache versus anonymous memory is evicted, and which container is selected by the OOM killer when reclaim alone cannot free enough memory? These questions are directly relevant for container density planning: an operator deciding how tightly to pack memory-constrained containers on a host is, in effect, betting on the fairness and predictability of exactly this reclaim behavior, and the kernel documentation's description of individual interface semantics does not, by itself, predict the aggregate, multi-tenant outcome [2].

The remainder of this paper is organized as follows. Section 2 reviews the documented architecture of the v1 and v2 memory controllers and the existing (narrower) empirical literature comparing them. Section 3 outlines the specific gap this leaves in the literature. Section 4 lists the objectives of this study. Section 5 proposes a set of testable hypotheses derived from documented interface differences. Section 6 describes the benchmark architecture used for testing. Section 7 specifies the experimental protocol, workload generator, and host configuration required to run the benchmark. Section 8 presents the results as a labelled execution plan because the benchmark has not yet been run. Sections 9 and 10 discuss the limitations and future work, and Section 11 concludes [3].

BACKGROUND AND LITERATURE REVIEW

The cgroups v1 memory controller, documented in the kernel's legacy memory-controller reference, tracks a cgroup's resident and cached pages jointly and enforces a single hard limit (`memory.limit_in_bytes`) above which the kernel invokes direct reclaim within that cgroup, or the OOM killer if reclaim cannot free enough pages [2]. This design has two properties frequently cited as pitfalls in the container literature: a cgroup's memory limit is enforced without any memory being reserved for it in advance, and the accounting pool mixes anonymous (heap/stack) memory with page cache in a way that lets aggressive cache use by one container's reclaim behavior interact unpredictably with a neighboring container's working set, since the kernel can reclaim cache system-wide independent of any single cgroup's stated limit [3].

The cgroups v2 unified hierarchy retains a hard limit (`memory.max`) but adds two further controls, as documented in the current kernel administration guide. `Memory [1]. min` defines a hard protection floor below which a cgroup's memory is never reclaimed except to avoid an out-of-memory condition; `memory.low` defines a best-effort protection floor that is only crossed if no reclaimable memory is available elsewhere in the system. Above these floors, the documentation specifies that pages are reclaimed proportionally to how far a cgroup's usage sits above its effective boundary, rather than as an all-or-nothing cutoff. cgroups v2 also exposes `memory.pressure`, a per-cgroup Pressure Stall

Information file reporting the fraction of wall-clock time in which at least one task in the cgroup was stalled waiting on memory, which orchestration systems can poll to anticipate reclaim-driven latency spikes before they show up as application-level errors [4].

Direct empirical comparisons of these two hierarchies remain limited in scope. Wiedner, Daichendt, Andre, and Carle, and the closely related conference version by Daichendt, Wiedner, and Andre, measured network function virtualization latency under cgroups v1 versus v2 in isolation on commodity hardware and found that the two versions achieve comparable isolation, but that v2's implementation consumes roughly 2.4% fewer instructions than v1's, translating into lower tail latency; their evaluation, however, targeted CPU and networking paths rather than memory reclaim under contention. A separate industry report from Red Hat's OpenShift performance engineering team compared the cgroup v1/runc and cgroup v2/crun combinations at pod densities of up to 2,500 pods per node on bare-metal servers, finding similar pod-scheduling throughput between the two configurations with a modestly faster tail latency under v2; however, this evaluation focused on pod scheduling and startup rather than isolating memory-reclaim fairness as a separate variable. On the applied side, Wang et al. measured how allocating additional memory to one containerized, memory-intensive application on a shared host affected co-located workloads through cache and memory-bus contention, finding significant and workload-dependent performance degradation, which is suggestive of the general problem this study is concerned with, but was not designed to isolate the cgroups version as an experimental factor [5].

An older but frequently cited practitioner analysis, notably LinkedIn's internally reported "pressure testing" of production containers under cgroups v1, documented that a cgroup's memory limit counting both anonymous and cached pages in a single pool could let memory-hungry neighbors indirectly evict another container's page cache well before that container's own working set was under pressure, motivating some of the design changes eventually adopted in cgroups v2 [6]. To our knowledge, however, no published study has repeated a comparable multi-container contention experiment specifically to compare v1 and v2 side by side under matched conditions, with cgroups version as the sole controlled variable.

PROBLEM STATEMENT

The kernel documentation specifies precisely how the memory of a single cgroup is protected and reclaimed under both hierarchies. It does not, and by its nature as an interface reference cannot, specify the emergent, system-level behavior that results when several containers with different memory.low/memory.high configurations compete for a bounded pool of physical memory on the same host, which is exactly the situation that container density planning has to reason about. Existing empirical comparisons of v1 and v2 either target a different subsystem (CPU/network latency) or a different variable (density at fixed pod-scheduling throughput) rather than reclaim fairness and OOM-kill behavior under memory contention specifically. This leaves operators and orchestrator designers without a reproducible, cgroups-version-controlled answer to a concrete operational question: for a given level of memory overcommitment, does the v2 soft-limit mechanism actually produce more predictable, more fairly distributed reclaim pressure across co-located containers than v1's single hard-limit model, or does it mainly change which container is reclaimed from first without changing the aggregate outcome?

OBJECTIVES

This paper has four objectives:

1. First, to translate the documented interface-level differences between the v1 and v2 memory controllers into a small number of directly testable hypotheses about multi-container reclaim behavior under contention [7].
2. Second, to design a benchmark architecture that isolates cgroups version as the sole experimental variable while holding kernel version, container runtime, workload, and host hardware fixed.

3. Third, to specify the exact workload generator, container configuration matrix, and measurement instrumentation (drawing on memory.pressure/PSI counters and OOM-kill event logs) needed to execute the benchmark reproducibly on any host that supports both hierarchies.
4. Fourth, to report honestly what has and has not yet been measured, presenting the results section as a specific execution plan with placeholder tables rather than invented figures, since the benchmark described here has not yet been executed on physical hardware.

PROPOSED HYPOTHESES

Drawing directly on the documented interface semantics reviewed in Section 2, we propose three hypotheses for empirical testing rather than presenting them as established findings [8].

H1—Soft-limit Responsiveness

Because memory.low/memory.high allows the v2 controller to apply reclaim pressure proportionally to a cgroup's overage before a hard limit is hit, we hypothesize that, under a gradually increasing multi-container memory load, cgroups v2 will show a measurably longer interval between the onset of elevated memory pressure stall time and the first OOM-kill event, compared with cgroups v1 reaching an equivalent aggregate host memory-utilization level, because v1 has no equivalent proportional-throttling stage before its single hard limit is reached.

H2—Cache-versus-anonymous Eviction Balance

Because v1 accounts for anonymous and cached pages together at the point of enforcement while v2 documents cgroup writeback as coordinated between the memory and I/O controllers, we hypothesized that, for a mixed workload combining a cache-heavy container and an anonymous-memory-heavy container on the same host, cgroups v2 would show a smaller cross-container swing in cache-hit ratio degradation for the cache-heavy container when the anonymous-memory-heavy container is placed under load, compared with the equivalent v1 configuration.

H3—OOM-kill Fairness

Because memory.min provides an explicit, unconditional protection floor unavailable in v1, we hypothesize that when several containers are configured with a mix of protected (memory.min set) and unprotected containers and the host is driven into an out-of-memory condition, cgroups v2 will show a lower rate of the OOM killer selecting a process from a protected container than the equivalent v1 configuration relying only on relative memory.limit_in_bytes sizing, because v1 has no mechanism to exempt a cgroup from consideration once system-wide memory is exhausted.

BENCHMARK ARCHITECTURE

The proposed benchmark architecture consists of a control host running a fixed kernel version compiled or selected to support both the v1 and v2 hierarchies (recent kernels support v2 natively, with v1 available via the systemd.unified_cgroup_hierarchy=0 boot parameter, which allows both configurations to be exercised on identical hardware and kernel binaries), a container runtime (containerd, configured to target each hierarchy in turn), a synthetic memory-pressure workload generator, such as the widely used stress-ng tool configured to produce controllable anonymous-memory and page-cache-heavy load profiles, and an instrumentation layer that polls each container's memory.pressure (v2) or an equivalent vmstat/memory.stat-derived proxy (v1, which lacks native per-cgroup PSI) at fixed intervals, alongside the kernel's OOM-kill event log parsed from dmesg or the kernel's oom_kill tracepoint (Figure 1 and 2).

Each experimental run places a fixed set of containers on the host, each configured with a memory limit and, where applicable, memory.low/memory.min protection values, and drives them through a scripted, gradually increasing combined memory demand using the workload generator until either a stable steady state is reached or an OOM-kill event is triggered. The full time series of per-container memory usage, pressure-stall time, and reclaim events is logged for the run, and each configuration (varying which containers are protected and varying overall host overcommitment ratio) is repeated across both hierarchies under otherwise identical conditions [9].

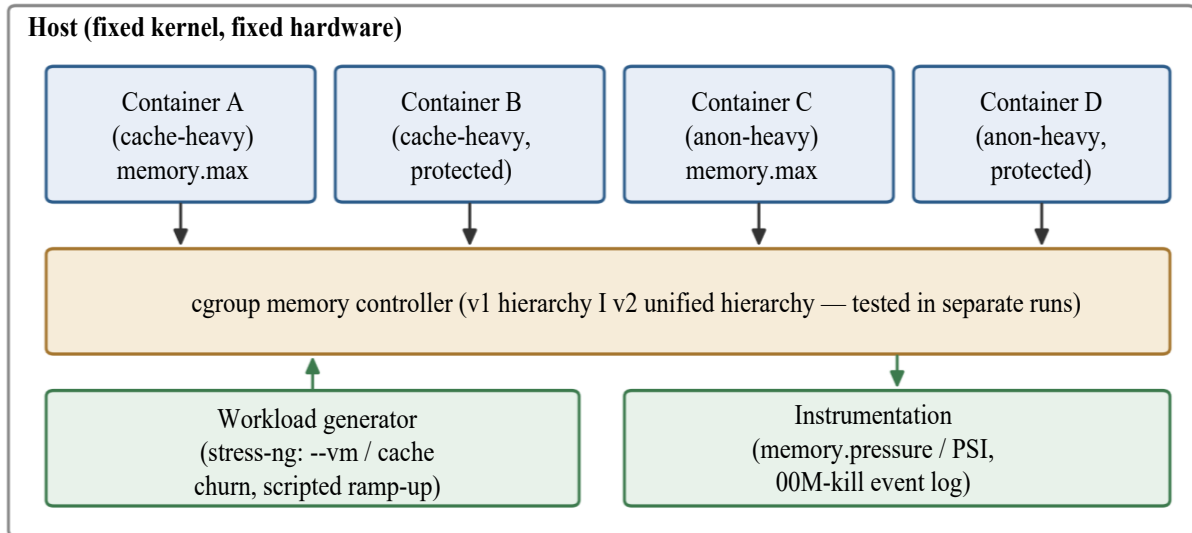


Figure 1. Benchmark architecture: four co-located containers under a shared cgroup hierarchy, driven by a scripted workload generator and monitored via pressure-stall and OOM-kill instrumentation.

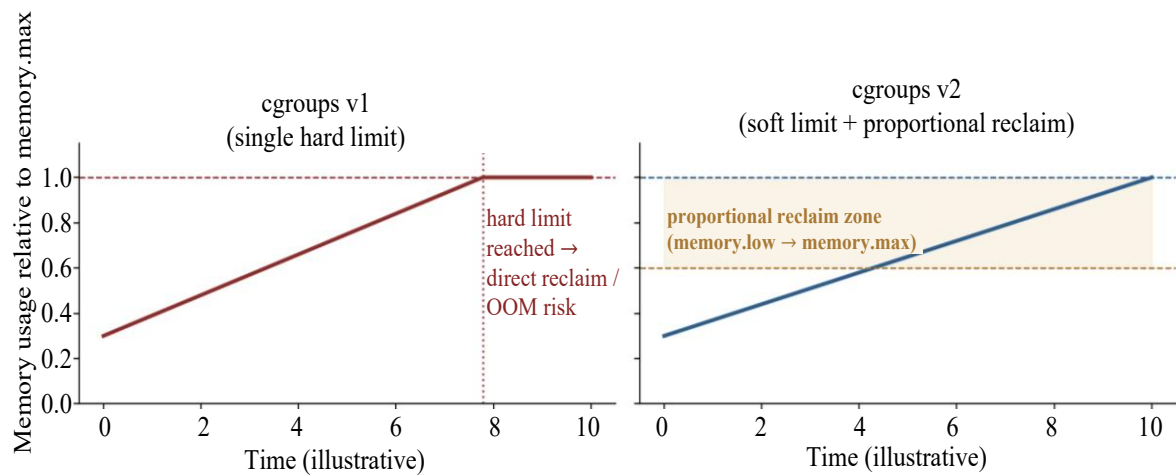


Figure 2. Schematic contrast between v1 hard-limit enforcement and v2 proportional soft-limit reclaim, as described in kernel documentation. Illustrative only—not measured data.

IMPLEMENTATION AND EXPERIMENTAL SETUP

We propose the following protocol. The host would run a single Linux kernel version (at minimum 5.15, to ensure stable PSI support) on fixed hardware with a known total memory capacity, partitioned so that the container memory limits sum to a chosen overcommitment ratio relative to the host capacity (e.g., 110%, 150%, and 200% of the physical memory, tested as separate conditions). For each overcommitment ratio, four containers would be launched: two configured as cache-heavy (repeatedly reading a working set larger than their memory limit to generate page-cache churn) and two configured as anonymous-memory-heavy (allocating and touching heap memory up to their limit using stress-ng's --vm mode), with one cache-heavy and one anonymous-memory-heavy container additionally configured with memory.min protection under the v2 condition, and an equivalent proportionally larger memory.limit_in_bytes under the v1 condition as the closest available analog, since v1 has no direct protection-floor equivalent—this asymmetry is itself part of what the experiment is designed to surface, not a flaw to be engineered away [10].

Each of the resulting configurations (three overcommitment ratios $\times$ two hierarchies) would be run for a fixed duration (proposed: fifteen minutes of steady-state load following a two-minute ramp-up)

and repeated five times to characterize run-to-run variance, which matters because reclaim and OOM-kill timing are sensitive to scheduling non-determinism. The primary measurements are as follows: time from load-ramp start to first sustained memory; pressure elevation above a fixed threshold (testing H1); the ratio of cache-hit-rate degradation between the two cache-heavy containers when only one anonymous-memory-heavy neighbor is under load (testing H2); and in runs that reach an actual OOM condition, which container's process is selected by the OOM killer, cross-tabulated against whether that container was configured as protected (testing H3).

As with the companion paper on bus-factor trajectories submitted alongside this one, we are explicit that this protocol has been designed but not yet executed: at the time of writing, we have not run this benchmark on physical or virtualized hardware, and Section 8 below presents the results section as the exact tables this protocol is designed to populate, with every numeric cell marked as pending, rather than as completed measurements.

RESULTS AND DISCUSSION (PROPOSED EXECUTION PLAN)

Table 1 shows the intended reporting format for the three hypotheses under each of the three overcommitment ratios; all numeric entries are placeholders pending execution of the protocol in Section 7.

Once populated, the intended discussion would first assess H1 by comparing time-to-first-OOM across hierarchies at each overcommitment ratio, asking whether the v2 soft-limit mechanism measurably delays the point at which containers begin failing outright or merely shifts stall time earlier without preventing an eventual OOM at the same overcommitment level. Second, it would assess H2 by comparing the cache-hit degradation ratio, asking whether the two container types genuinely interfere less under v2, or whether the improvement (if any) is confined to specific workload shapes not represented in this protocol. Third, it would assess H3 directly from the protected-container OOM-kill rate column, which is the most straightforward of the three comparisons to interpret: a rate near zero under v2 and reliably above zero under v1 would be strong, mechanism-consistent evidence that memory.min changes real outcomes and not merely the documented intent.

We again avoid speculating about the direction in which these results will point before the protocol is executed. The value of stating hypotheses explicitly in Section 5 is that they are falsifiable in either direction by the specific measurements in Table 1, and a completed version of this paper would report whichever outcome the data show, including a null result on any of the three hypotheses if that is what occurs.

LIMITATIONS

The proposed protocol uses a synthetic workload generator rather than production application traces, which is a deliberate simplification for controllability but means that the results, once obtained, would characterize reclaim mechanics under a specific, somewhat artificial load shape rather than under the full diversity of real containerized workloads; a natural next step, noted in Section 10, is to repeat the protocol with a small set of representative real applications (for example, a cache-heavy key-value store alongside a compute-heavy batch job). The proposed v1 analog for memory.min protection (a proportionally larger hard limit) is not a true equivalent, since it changes the protected container's effective capacity rather than exempting it from OOM consideration outright, and this asymmetry needs to be reported transparently alongside any H3 results rather than presented as a like-for-like comparison. Finally, results obtained on one kernel version, one container runtime, and one hardware configuration may not generalize to other combinations, particularly given that memory-reclaim behavior has continued to evolve across kernel releases even within the v2 hierarchy.

Table 1. Planned reporting format across overcommitment ratios and hierarchies. All values are placeholders pending benchmark execution.

Overcommitment	Hierarchy	Time to first OOM (s)	Cache-hit degradation ratio	Protected-container OOM-kill rate
110%	cgroups v1	[PENDING]	[PENDING]	[PENDING]
110%	cgroups v2	[PENDING]	[PENDING]	[PENDING]
150%	cgroups v1	[PENDING]	[PENDING]	[PENDING]
150%	cgroups v2	[PENDING]	[PENDING]	[PENDING]
200%	cgroups v1	[PENDING]	[PENDING]	[PENDING]
200%	cgroups v2	[PENDING]	[PENDING]	[PENDING]

FUTURE SCOPE

Beyond repeating the protocol with production-representative workloads, a natural extension is to test whether the same hierarchy-dependent reclaim behavior changes as the number of co-located containers scales well beyond the four used here, which is directly relevant to the higher-density regimes explored in industry benchmarks such as the OpenShift 2,500-pods-per-node study. A second extension would incorporate cgroup-aware CPU scheduling interactions alongside memory reclaim, since CPU throttling and memory reclaim can interact when a container's reclaim-driven page faults compete with CPU quota enforcement. A third extension, more tool-oriented, would package the instrumentation layer described in Section 6 as a standalone, runtime-agnostic diagnostic that cluster operators could run against their production node pools to characterize reclaim fairness under their actual overcommitment ratios, rather than only under the controlled synthetic conditions proposed here.

CONCLUSION

The move from cgroups v1 to cgroups v2 changed the memory controller's interface substantially, adding proportional soft-limit protection and per-cgroup pressure signaling that the kernel documentation describes clearly at the level of a single cgroup. Whether these interface changes translate into more predictable, fairly distributed reclaim behavior when several containers compete for memory on the same node is an empirical question that existing literature—focused on CPU/network overhead or on pod-scheduling throughput rather than reclaim fairness specifically—does not yet answer directly. This study proposed three falsifiable hypotheses grounded in the documented interface differences, together with a controlled, reproducible benchmark protocol for testing them. Consistent with the aim of academic transparency, we have reported the current state of the work accurately: the hypotheses and protocol are complete and ready to execute, while the benchmark itself, and therefore any comparative claim about which hierarchy actually behaves better under contention, remains to be run.

REFERENCES

1. Billimoria KN. Linux kernel programming: a comprehensive and practical guide to kernel internals, writing modules, and kernel synchronization. Birmingham (UK): Packt Publishing; 2024 Feb 29.
2. Jain SM. Cgroups. In: Linux containers and virtualization: utilizing Rust for Linux containers. Berkeley (CA): Apress; 2023. p. 47–81.
3. Wiedner F, Daichendt A, Andre J, Carle G. Control groups added latency in NFVs: an update needed? In: 2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN); 2023 Nov 7. p. 40–45.
4. Anderson J, Hu H, Agarwal U, Lowery C, Li H, Apon A. Performance considerations of network functions virtualization using containers. In: 2016 International Conference on Computing, Networking and Communications (ICNC); 2016 Feb 15. p. 1–7.
5. Wang Q, Kar S, Mishra P, Linduff C, Izard R, Anjam K, et al. Container resource allocation versus performance of data-intensive applications on different cloud servers [Preprint]. 2023. arXiv:2311.07818.

-
6. Brigoli R, Masood F. MLOps with Red Hat OpenShift: a cloud-native approach to machine learning operations. Birmingham (UK): Packt Publishing; 2024 Jan 31.
 7. Takkalapally D. Engineering for scale: performance testing and capacity planning in OpenShift and Pivotal Cloud Foundry. *Int J Comput Sci Eng Innov.* 2026 Feb 24:1–6.
 8. Gedia D, Perigo L. Performance evaluation of SDN-VNF in virtual machine and container. In: 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN); 2018 Nov 27. p. 1–7.
 9. Dziurzanski P, Indrusiak LS. Value-based allocation of Docker containers. In: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 2018 Mar 21. p. 358–362.
 10. Micallef Z, Rajenthiram K, Gerostathopoulos I. A systematic review of MLOps tools: tool adoption, lifecycle coverage, and critical insights [Preprint]. 2026. arXiv:2604.16371.