Isolation in the Age of Agents

Anjali
University of Wisconsin-Madison
Madison, Wisconsin, USA
anjali@wisc.edu

Michael M. Swift
University of Wisconsin-Madison
Madison, Wisconsin, USA
swift@cs.wisc.edu

ABSTRACT

AI agents are changing the isolation landscape of modern cloud infrastructure. Unlike traditional workloads, agents execute heterogeneous and dynamic sequences of tools that read and write files, spawn processes, install packages, run tests, and interact with external services. These tools interact directly with OS resources, expanding the attack surface and exposure to interference, making isolation essential. Hence, the question is no longer simply whether to isolate agent execution, but how to choose an isolation boundary that balances isolation, performance, and efficiency for a dynamic and heterogeneous tool set.

In this paper, we propose *Expectation of Risk* (*EoR*), an isolation metric for quantifying interference risk induced by an agent's tool-mediated interactions with shared system resources. *EoR* models how much a workload exercises shared resources with the severity of interference through those resources, yielding a resource-specific isolation signal. We instantiate *EoR* for performance interference through shared host-kernel locks using host-level syscall profiles and a reusable syscall-to-lock profile, and evaluate it across Linux containers, gVisor, and Firecracker using replayed agent tool-call traces. Our results show that EoR ranks runtimes by their exposure to shared host-kernel state, and this ranking aligns with measured co-location slowdown, especially when the shared host-kernel state is the main source of interference.

1 INTRODUCTION

AI agents combine models for reasoning with tools for action, enabling them to translate human intent into autonomous execution [11]. They are hybrid systems that combine AI models with traditional software components such as orchestrators, vector databases, and execution runtimes [16], and are increasingly deployed in multi-tenant environments [7, 31]. From a systems perspective, the tool layer [16, 22, 24, 30, 31] is where an agent's decisions become resource-consuming execution. Tools read and write files, spawn processes, execute code, issue network requests, query services, and interact with external state [16, 22, 24, 30, 31].

Tool execution is hence a key source of interference in multi-tenant agent deployments. Whether tools run with the agent orchestrator [17] or through a remote multi-tenant tool server such as MCP [4, 8], they remain a major point of contact with shared OS resources and backend services. Although agents also introduce other shared components (*e.g.,* the model serving and orchestration layers) [8], this paper focuses on tool-mediated execution because it is where isolation mechanisms directly constrain agent actions.

Agentic workloads differ from prior multi-tenant workloads: they are (a) interactive but not necessarily latency-sensitive, (b) high-utilization because agents think time is low and parallelism can be high, and (c) heterogeneous, since an agent may invoke a wide range of tools, including tools not known when the environment was provisioned. These properties make the CPU and host execution environment central to agent isolation. Although accelerators may serve the model, the CPU coordinates tool invocation, runtime state, and OS interaction [23, 26, 29]. Thus, agent interference is not only a model-serving problem; it also arises where tool execution meets shared infrastructure.

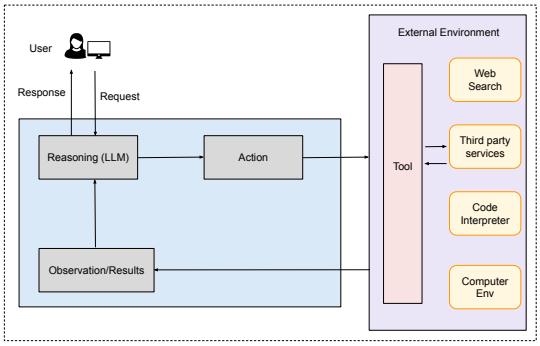
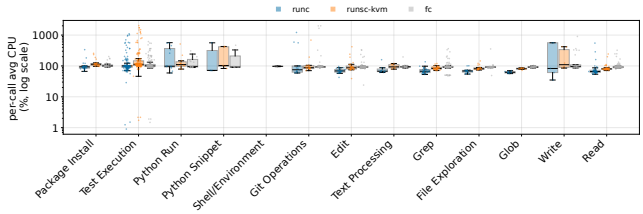


Figure 1: Workflow of an agentic system for a single agent.

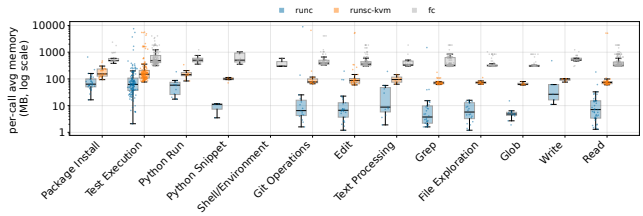
Modern deployments mitigate interference and security risks (*e.g.,* co-location slowdown and unintended shared-state access) using isolation mechanisms such as containers, secure containers, lightweight virtual machines, and other sandboxes [1, 2, 12, 14, 15, 19, 25]. However, agentic workloads make the isolation choice difficult. Isolating each tool in a separate environment would provide fine-grained boundaries, but it can also add startup latency, duplicate runtime state, and increase coordination overhead. In practice, an agent's tools are often executed within a common isolation environment, forcing systems to choose one boundary for a diverse and changing toolset. Moreover, recent agent sandboxes such as nono [20] and OpenShell [21] use Linux security modules for isolation and provide fine-grained, low-overhead access control, but they do not isolate shared kernel state.

This creates a fundamental dilemma: *what is the appropriate isolation platform for an agent with heterogeneous tool usage?* [6, 18, 27] Stronger isolation mechanisms (*e.g.,* microVMs) may reduce exposure to shared host state but introduce unnecessary overhead for benign or short-lived operations. Lighter mechanisms (*e.g.,* containers) may reduce overhead but expose more tool executions to shared host resources. Existing systems largely rely on one-size-fits-all isolation choices or static resource controls, lacking a principled way to reason about this trade-off between isolation, performance, and resource efficiency.

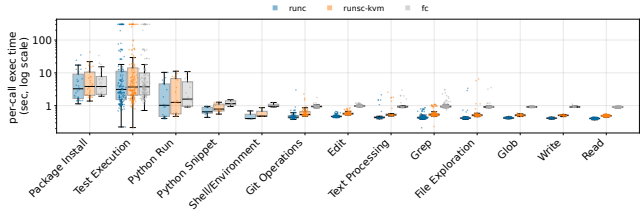
A key challenge in making such isolation decisions is that an agent's exact sequence of tool invocations is often not known in advance. Agent execution is inherently dynamic, with tool usage depending on intermediate results, external responses, and evolving plans. However, despite this uncertainty, agent behavior is typically



(a) Average CPU utilization per tool call. >100% = tool used multiple cores in parallel and <100% = time blocked/waiting off-CPU.



(b) Average memory usage per tool call.



(c) Execution-time distribution per tool call.

Figure 2: Resource utilization and execution-time distribution per tool category across isolation platforms.

guided by a bounded set of accessible tools and task-level intent [22, 30]. While individual execution trajectories may vary, these signals provide a basis for approximating how likely an agent is to interact with shared resources.

Building on this observation, we seek a principled way to quantify the interference risk induced by an agent's interactions with shared resources, primarily through its tool usage. The goal is to make an isolation selection tool-aware: to estimate how much shared-state exposure an execution environment creates for an agent task, and when stronger isolation may be needed. Instead of relying only on worst-case assumptions, we estimate exposure from observed or previously profiled resource behavior, combining how frequently a workload exercises a shared-resource interface with the potential impact of that interaction.

We introduce *Expectation of Risk* (*EoR*), an isolation metric that estimates the interference risk induced by an agent's tool-mediated interactions with shared infrastructure. *EoR* is a resource-specific isolation signal: for one interference dimension, it combines how much a workload exercises shared resources with the severity of interference through those resources. In this paper, we instantiate EoR for shared host-kernel exposure using host-level syscall profiles and shared kernel locks. This provides a first step toward *tool-aware runtime selection*, where isolation risk can be combined with performance and efficiency goals.

2 MOTIVATION: TOOL-AWARE ISOLATION

Agent tool calls are the system-facing actions through which an agent accesses OS resources. A single task may invoke many tools, and each tool can stress different parts of the system: file-search tools interact with filesystem metadata, test commands spawn processes and consume CPU and memory, and package managers access filesystems, networks, package caches, and build tools. However, tool behavior alone does not determine isolation: the same logical tool may expose different amounts of shared host-kernel state depending on whether it runs in a container, secure container, or microVM.

We replay Haiku tool-call traces from AgentCgroup [31], collected on SWE-rebench software-engineering tasks [5]. These tasks require agents to interact with real codebases by inspecting files, modifying code, running commands, and adapting based on execution feedback. Figure 2 shows the CPU utilization, memory usage, and execution-time distribution of all tool calls across all Haiku tasks on each runtime. We observe that tool behavior is highly heterogeneous: test execution and Python-based tools show higher CPU use, package installation tends to have larger memory and execution-time costs, and simple reads, edits, and text processing are comparatively lightweight. The same logical tool can also behave differently across platforms; for example, Firecracker (*fc*) often incurs a larger memory footprint due to the microVM boundary, while *runc* appears cheaper because it shares the host kernel directly.

This variation motivates *tool-aware* platform selection: different tools may need different isolation boundaries depending on their resource behavior and shared host-state exposure. Resource usage alone is insufficient, since a higher-overhead runtime may expose less shared host state, while a lightweight runtime may expose more. Thus, platform selection must consider both resource cost and shared system state exposed by each runtime.

This motivates our research question: *Can we use the tool behavior within an agent task to estimate which platform provides sufficient isolation?* Different isolation platforms change which system calls reach the host kernel. For example, *fc* handles most syscalls internally. Thus, the same tool can produce different host-level syscall profiles across platforms. We use host-level syscall profiles as a lightweight signal of how the task reaches the host kernel, map those profiles to shared kernel lock activity to compute task-level *EoR*, and use *EoR* to compare the isolation provided by different execution environments. *EoR* can later be combined with performance and efficiency goals to guide placement decisions.

3 ISOLATION AS A QUANTIFIABLE METRIC

We model an isolated workload s as an application workload w executed under a runtime that enforces an isolation boundary. For agentic systems, w is a tool invocation or a sequence of tool invocations issued by an agent. Let $\mathcal{R}_s$ denote the resources accessed by s, and let T_s denote the set of co-located workloads. The interference resources are those accessed by both s and at least one neighbor:

$$\mathcal{I}_s = \mathcal{R}_s \cap \left(\bigcup_{t \in T_s} \mathcal{R}_t \right). \tag{1}$$

If $\mathcal{I}_s = \emptyset$, then s is fully isolated with respect to the resource dimension being measured. Otherwise, the question is how much potential interference exists through the overlapping resources.

Expectation of Risk. We quantify this potential interference using *Expectation of Risk* (EoR):

$$\mathrm{EoR}(s) = \sum_{i \in \mathcal{I}_s} \mathrm{mag}(i) \cdot r(i). \tag{2}$$

Here, $\mathrm{mag}(i)$ is the interference magnitude, capturing how much the workload exercises resource i, and $r(i)$ is the associated risk, capturing the severity of interference through that resource. The neighbor dependence enters through $\mathcal{I}_s$: only resources accessed by both s and at least one co-located workload contribute to *EoR*. These overlapping resources can be identified from previously collected solo profiles of each workload–platform pair, without requiring the workloads to be profiled while co-running. *EoR* is comparative: lower *EoR* indicates lower exposure for the measured resource dimension.

The metric is resource-specific. Different isolation questions may instantiate $\mathcal{I}_s$, mag, and r differently. In this paper, we target performance interference from host-shared kernel state. We use host-level syscalls as the observable signal of how an agent task reaches the host kernel, and shared host-kernel locks as the interference resources. This produces a task-level estimate of host-lock exposure under each isolation runtime.

4 EMPIRICAL EVALUATION METHODOLOGY

We evaluate whether agent tool-call behavior can be used to estimate the isolation provided by different isolation platforms/runtimes. Our evaluation uses tool-call traces from Haiku agent executions collected by AgentCgroup [10, 31]. Each trace consists of the ordered sequence of tool calls issued during an agent task, including tool labels, command information where available, and tool call boundaries. We replay these traces under different isolation runtimes and collect execution time, resource usage, and host-level syscall activity for each tool call window. These measurements allow us to compare how the same logical agent task interacts with the underlying system across runtimes.

We measure across three isolation platforms: (1) native Linux containers (*runc*), (2) gVisor (*runsc-kvm*) release-20260520.0 using the KVM platform [13], and (3) Firecracker (*fc*) v1.15.1. These platforms represent increasingly strong isolation boundaries: *runc* shares the host kernel directly, *runsc-kvm* mediates system calls via a user-space kernel, and *fc* executes the workload inside a microVM. All experiments run on CloudLab [9] c220g5 nodes with 2.20GHz Intel Xeon Silver 4114 CPUs, 192GB memory, an Intel DC S3500 480GB SSD, a 1TB 7200RPM HDD, and a 10Gbps NIC. We run Ubuntu 22.04 with Linux kernel v6.1.

4.1 Tool-Call Replay Across Platforms

Our replay framework executes the extracted Haiku tool-call sequence inside each isolation platform. We measure only the execution time of the replayed tool-call sequence after the isolation environment has been initialized. Startup costs for creating the container, gVisor sandbox, or microVM are excluded. We run two sets of replay experiments.

First, we run each Haiku task in isolation under each platform and collect CPU and memory utilization metrics. These measurements motivate the runtime-selection problem by showing that the same logical agent task exhibits different resource behavior across isolation platforms, as shown in Figure 2.

Second, we run profiling experiments across two modes under each platform: (a) solo and (b) co-run. In the solo mode, the target task runs alone, and in the co-run mode, the same target task runs concurrently with N co-located homogeneous neighboring workloads on the same platform. We use the ratio (corun/solo) between solo and corun execution time to compute slowdown at the task and tool call granularity. This slowdown is the observed interference risk signal used to validate whether higher *EoR* estimates correspond to higher performance degradation. For each replay, we also collect the host-level syscall profile for each tool call window.

4.2 Syscall and Execution-Time Profiling

We profile each tool call using a common interface across platforms. The replay driver marks the start and end of every tool invocation and records its wall-clock execution time. In parallel, an eBPF profiler attaches to the kernel's `raw_syscalls:sys_enter` and `raw_syscalls:sys_exit` tracepoints to collect host-level syscall events. For each syscall, the profiler records the syscall identifier, process id, return status, and latency. At the end of the run, syscall events are assigned to the tool call window in which they occurred. This produces per-tool-call syscall counts and latency statistics.

4.3 Estimating Shared-State Exposure

We use host-level syscalls as a lightweight signal for estimating shared kernel exposure. We construct the syscall-to-lock profile by executing syscalls while tracing their lock activity, and retaining only locks that can be shared across workloads based on the kernel objects they protect. The idea is that the runtime determines which host-level syscalls are issued during a tool call, while the syscall-to-lock profile estimates the shared host-kernel state exposure via the lock activity associated with those host-level syscalls. This is an approximation: the exact lock activity of a syscall can depend on arguments, object types, and execution context. However, it lets us estimate shared-state exposure without enabling expensive lock tracing [3] for every full agent replay. A more robust profile could average lock behavior over multiple representative invocations of each syscall to capture this variability.

4.4 EoR Calculation

Our evaluation uses homogeneous neighbors that replay the target workload; thus, the target's reachable shared resources are the interference set, $\mathcal{I}_s = \mathcal{R}_s$ (Eq. 1). For heterogeneous neighbors, the same solo profiles apply, but $\mathcal{I}_s$ is restricted to resources reached by both the target and its neighbors.

We calculate *EoR* using host-level system calls as observable signals of access to shared host-kernel state. In this instantiation, the interference resources are shared host-kernel locks. Each syscall s is mapped to the shared locks it exercises using the syscall-to-lock profile (§4.3).

For a task t running on platform p, let $S_{t,p}$ be the set of host-level system calls observed during the task's solo execution. We evaluate

task-level isolation by aggregating system calls across all tool-call windows within the task. We compute:

$$\mathrm{EoR}(t,p) = \sum_{s \in S(t,p)} \lambda(t,p,s) \cdot R(s) \tag{3}$$

where $\lambda(t,p,s)$ is the interference magnitude of system call s, and $R(s)$ is the associated risk of one call of s. For tool-level *EoR*, we apply the same calculation after aggregating the solo syscall profiles for each tool category across all tasks, yielding a single *EoR* value per tool category and platform (§5.3.3).

We exclude blocking-by-design syscalls (*e.g.,* `epoll_pwait`, `poll`, `nanosleep`, `wait4`) from $S_{t,p}$, since their host-lock time is dominated by waiting rather than contention.

Interference Resource is the shared host-kernel state that can be contended across co-located tasks; we use shared kernel locks as its proxy.

Unit of Interference is the host-level system-call invocation. We denote the syscall identifier by s (*e.g.,* `read`, `write` or `mmap`). For each t and p, we count how many invocations of each syscall s are observed during solo execution.

Interference Magnitude is the solo issue rate of syscall s, computed by dividing the number of invocations of s by the execution time.

$$\lambda(t,p,s) = \frac{\texttt{count\_solo}(t,p,s)}{T_{\mathrm{solo}}(t,p)} \tag{4}$$

Risk is the shared-lock exposure per invocation of a host-level syscall s. We treat $R(s)$ as platform-independent: once a runtime issues syscall s to the host kernel, the syscall is serviced by the same host kernel and can reach the same shared kernel locks. Thus, different platforms affect *EoR* through the set and rate of host-level syscalls they generate, $\lambda(t,p,s)$, while the per-syscall risk $R(s)$ is obtained from a common syscall-to-lock profile. For each syscall s, we compute:

$$R(s) = \sum_{L} \frac{\texttt{hold\_ns}(s,L)}{\texttt{calls}(s,L) \cdot 10^9} \tag{5}$$

where L ranges over shared kernel locks reached by s, and $\texttt{calls}(s,L)$ is the number of invocations of s recorded in the same profiling run measured by $\texttt{hold\_ns}(s,L)$. Each ratio is the average hold time on L per invocation of s; the 10^9 factor converts nanoseconds to seconds, so $R(s)$ has units of seconds per invocation.

Thus, $R(s)$ is not the risk of the syscall as a resource; it is the aggregate risk of the shared lock resources reached by one invocation of syscall s.

Multiplying the system-call issue rate in calls per second by the per-call risk in seconds per call gives a dimensionless *EoR* contribution. Thus, *EoR* estimates the fraction of execution exposed to shared host-kernel lock contention, using only solo syscall behavior and the syscall-to-lock calculation.

5 RESULTS

In this section, we evaluate whether tool-call behavior can be used to estimate the isolation provided by different isolation platforms for agent tasks. We first measure the performance impact of tool-calls co-location by comparing solo and co-running executions at both task (§5.1) and tool call (§5.2) granularity, establishing the observed performance interference that an isolation metric should explain. We then evaluate whether task-level EoR (§5.3), computed from host-level syscall behavior and syscall-to-lock profiles, captures the shared host-kernel exposure associated with that interference.

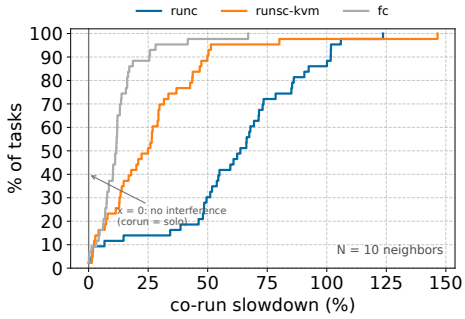


Figure 3: Task-level slowdown under co-location with N=10 neighboring workloads.

5.1 Slowdown under co-location

Figure 3 shows task-level slowdown for the replayed SWE-rebench tasks, each co-located with N=10 copies of the same task. *runc* has the largest slowdown, *runsc-kvm* is intermediate, and *fc* has the smallest slowdown. The magnitude of the gap between runtimes is substantial: the median slowdown under *runc* is approximately 64%, compared with 25% for *runsc-kvm* and below 20% for *fc*. Thus, under the same co-location pressure, *runc* tasks slow down more than three times as much as *fc* tasks at the median.

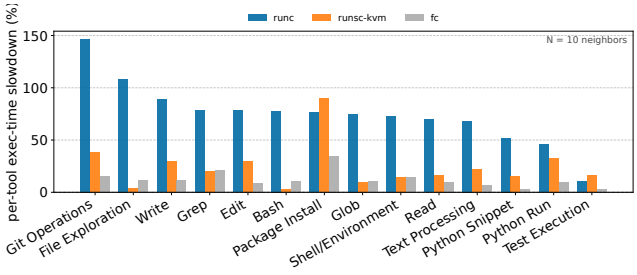
Importantly, some tasks experience little slowdown while others experience substantially larger degradation, particularly under *runc*. This task-level heterogeneity means that a fixed platform-level choice can over-provision low-risk tasks or under-protect high-risk ones. This implies the need for a per-task isolation signal that estimates the shared host state exposed by a task. Such a signal captures the isolation side of the placement trade-off; combined with performance targets and resource costs, it can guide selection of the *right* platform that keeps interference within an acceptable bound.

5.2 Tool-level Slowdown

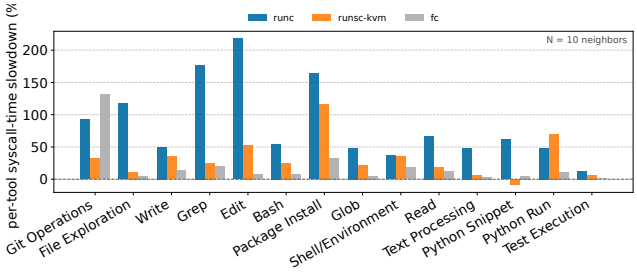
We now break down the slowdown by tool category (Figure 4a), aggregating solo and co-run execution time across all calls of each category, to understand which parts of the tool sequence are most affected. Under *runc*, several filesystem-facing categories experience large slowdowns, including Git operations, file exploration, writes, grep, edit, and Bash commands. Thus, the task-level slowdown is not spread uniformly across the tool sequence but concentrated in specific tool categories.

runsc-kvm generally shows lower slowdown than *runc*, but not uniformly. Package installation, for example, slows down more under *runsc-kvm*, likely because it combines many filesystem, process, and package-cache operations that are mediated by the user-space kernel, amplifying runtime-specific overhead. *fc* shows the lowest slowdown for most tool categories, indicating lower end-to-end impact from co-located tool execution.

Next, we ask whether these end-to-end slowdowns are reflected in host-kernel syscall time. Figure 4b shows syscall-time



(a) Per-tool execution time slowdown.



(b) Per-tool syscall time slowdown.

Figure 4: Tool-level slowdown under co-location with N=10 neighboring workloads.

slowdown for the same tool categories, computed by summing `count` × `avg_lat_ns` across all syscalls and tool calls in each category and comparing co-run against solo.

For many tool categories, host syscall time increases under co-location, especially under *runc*. This shows that co-location affects a broad set of host syscall paths, not only the tools with the largest end-to-end slowdowns. However, syscall-time slowdown is tool-dependent and does not always match end-to-end slowdown. We minimize hardware-level interference by pinning CPUs, disabling hyperthreading, provisioning memory, and partitioning the LLC, but some shared residual channels may remain. We therefore treat host syscall behavior as one observable isolation dimension rather than a complete account of end-to-end slowdown.

Together, these results show that co-location slowdown is both tool-dependent and runtime-dependent. Execution-time slowdown shows where interference appears in the tool sequence, while syscall-time slowdown captures the shared host-kernel state dimension exposed through syscall paths. We next evaluate whether *EoR* can estimate this shared-state exposure from solo syscall behavior.

5.3 EoR

We evaluate the task-level *EoR* defined in §4 (Eq. 3). Since *EoR* is computed entirely from solo runs, the measured co-run slowdown serves as a separate validation signal.

5.3.1 EoR–Performance Correlation. *EoR* is intended as a comparative isolation signal rather than a calibrated predictor of absolute slowdown, hence, we use Spearman rank correlation [28] to test whether higher task-level *EoR* corresponds to higher measured co-run slowdown. Table 1 shows the correlation pooled across all task–platform points and within each platform.

We observe that *EoR* is strongly correlated with slowdown ($\rho = 0.67$, $n = 129$) when pooled across runtimes, showing that it

Scope	n (data points)	Spearman ρ
Pooled (all platforms)	129	0.67
runc	43	0.46
runsc-kvm	43	−0.11 (n.s.)
fc	43	0.34

Table 1: Spearman rank correlation between *EoR* and measured co-run slowdown (N=10), pooled across platforms and within each platform. *n.s.*: not significant ($p > 0.05$).

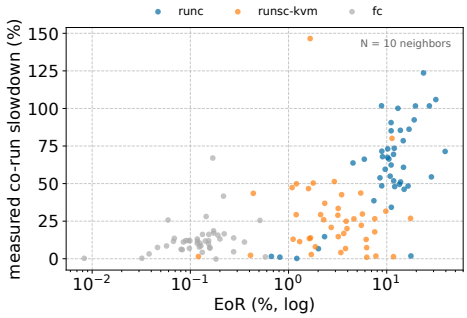


Figure 5: Task-level *EoR* versus measured co-run slowdown.

Platform	Median *EoR* (%)	Median slowdown (%)
runc	11.1	64
runsc-kvm	3.2	25
fc	0.13	12

Table 2: Median task-level *EoR* and measured co-run slowdown by platform (N=10).

captures the large cross-platform differences in performance interference. This pattern of correlations is what the mechanism predicts. Within *runc*—the platform that shares the host kernel directly—*EoR* predicts per-task slowdown ($\rho = 0.46$), confirming that where host-lock sharing is the dominant channel, *EoR* tracks it. For *runsc-kvm* and *fc*, which limit syscalls to the host kernel, host-lock exposure is low by design; accordingly, *EoR* shows a weaker (*fc*, $\rho = 0.34$) or statistically insignificant (*runsc-kvm*, $\rho = -0.11$) within-platform correlation. Thus, *EoR* predicts the *shared host-lock component* of interference, which dominates under *runc* and is small under the stronger isolation boundaries; the insignificant *runsc-kvm* result indicates no host-lock signal to detect, rather than an inverse relationship.

5.3.2 Cross-Platform EoR Ordering. Beyond per-task prediction, *EoR* reproduces the *cross-platform ordering* of interference from solo data alone. Table 2 reports median *EoR* and median measured slowdown per platform. *EoR* ranks the platforms in the same order as measured slowdown: *fc* < *runsc-kvm* < *runc*. This comparison is important as slowdown is the observed co-location outcome, while *EoR* is computed without running the workload under co-location.

The *EoR* gap is substantial: median *EoR* under *runc* is about 85× higher than under *fc* and about 3.5× higher than under *runsc-kvm*. This range is much wider than the corresponding slowdown range (64%, 25%, and 12%), indicating that *EoR* is not a linear predictor of absolute slowdown. Instead, it measures the host-shared-lock exposure created by each runtime. Thus, the same replayed agent tasks reach very different amounts of shared host-kernel state depending on the runtime.

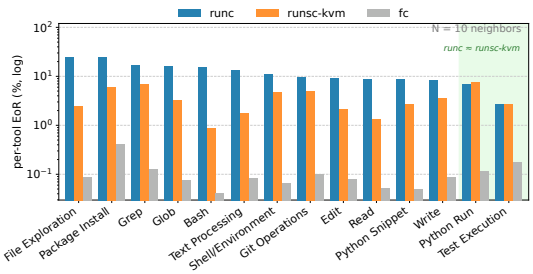


Figure 6: Tool-level *EoR*.

Scope	n (total tools)	Spearman ρ (exec)	Spearman ρ (syscall)
Pooled (all platforms)	42	0.82 (p<0.001)	0.74 (p<0.001)
runc	14	0.52 (p=0.059)	0.41 (p=0.144)
runsc-kvm	14	0.53 (p=0.049)	0.52 (p=0.059)
fc	14	0.36 (p=0.210)	0.23 (p=0.422)

Table 3: Spearman rank correlation between per-tool *EoR* and measured per-tool corun slowdown (N=10), using execution-time (Figure 4a) and syscall-time (Figure 4b) slowdown. Each within-platform correlation uses 14 tool categories.

With only three well-understood platforms, this ordering may seem unsurprising. Its value is that *EoR* derives the ranking from each platform's measured host-lock exposure rather than assuming it. The same procedure can therefore be applied when the ordering is not known a priori, such as with new runtimes, unfamiliar sandboxes, or different configurations of the same runtime. Here, the known *ordering of fc < runsc-kvm < runc* serves as a sanity check that *EoR* captures the expected shared-host lock exposure differences between these platforms.

Figure 5 shows that this separation holds at the task level, not only at the median: the platforms form ordered *EoR* bands that align with their slowdown bands, placing *runsc-kvm* between *runc* and *fc*. Because *EoR* is computed entirely from solo execution, it can provide the isolation side of a placement decision by ranking platforms according to shared-host state exposure, without requiring direct co-run measurements for every task.

5.3.3 Tool-Level EoR. Task-level *EoR* aggregates over an entire agent trajectory. We now compute *EoR* at tool-call granularity, aggregating solo syscall profiles within each of the 14 tool categories, and ask whether *EoR* ranks *tools* by interference exposure the way it ranks tasks and platforms.

Figure 6 shows per-tool *EoR* across platforms. We make two key observations. First, the isolation benefit is tool-dependent. For file and I/O-heavy tools such as File Exploration and Read, *runsc-kvm* has substantially lower *EoR* than *runc*, by about 5-10×. In comparison, for Test Execution and Python Run, *EoR* under *runsc-kvm* is comparable to *runc*, indicating that *runsc-kvm* does not substantially reduce host-lock exposure for these tool categories.

Second, the relative *EoR* of tools changes across platforms. File Exploration has one of the highest *EoR*s under *runc*, but drops substantially under *runsc-kvm*. In contrast, Python Run has relatively low *EoR* under *runc*, but is among the highest under *runsc-kvm*. This result shows that the isolation benefit of a platform depends on the tools an agent executes. Hence, the same isolation platform may provide different benefits depending on a task's toolset.

Table 3 reports the rank correlation between per-tool *EoR* and the tool-level slowdowns of §5.2. Pooled across platforms, the correlation is strong (ρ=0.82 against execution-time slowdown, ρ=0.74 against syscall-time slowdown). As with the task-level result, the pooled correlation partly reflects differences across platforms. A harder test is whether *EoR* can rank tools within each platform.

Unlike the task-level result, where *runsc-kvm* showed no within-platform signal (ρ=−0.11), the tool-level correlations are positive on all three platforms. The execution-time correlations are similar for *runc* (ρ=0.52) and *runsc-kvm* (ρ=0.53). This suggests that aggregation by tool category reduces some per-task variation and reveals a rank signal that is less visible at task granularity. However, with only n=14 tool categories per platform, statistical power is limited, so we interpret these within-platform results as directional rather than conclusive.

The correlation is weakest under *fc* (ρ=0.36 for execution time). As Figure 6 shows, *fc* keeps host-lock exposure low across all tool categories. Within this low-exposure regime, differences in end-to-end slowdown may increasingly reflect interference sources outside the shared host-lock state captured by the current *EoR* instantiation.

6 DISCUSSION AND FUTURE WORK

EoR complements resource partitioning. CPU pinning, cgroups, and cache or bandwidth partitioning control resource allocation, but they do not eliminate interference through shared kernel state. Co-located workloads may still contend on allocator structures, or filesystem-journaling paths even when CPU, memory, and cache resources are controlled. EoR captures this residual shared kernel-state interference and can serve as the isolation side of a future placement policy, while allocation controls manage partitionable resources within the chosen isolation boundary.

From prediction to placement. EoR provides the isolation side of the placement trade-off, but runtime selection must balance isolation, performance, and efficiency. A future policy would combine EoR with slowdown targets and runtime overheads to choose a runtime that provides sufficient isolation without over-isolating low-risk tasks or under-isolating high-risk ones. This requires calibrating EoR against performance targets and extending it beyond the current shared-kernel-state signal to other interference dimensions.

Scaling and tool non-determinism. Applying EoR at scale raises two practical questions. First, EoR requires profiling tool behavior, which appears costly for a large tool ecosystem. However, the expensive component—the syscall-to-lock profile $R(s)$ is a property of the host kernel, measured once and reused across all tools and workloads; per-task EoR then needs only cheap solo syscall counts (λ), with no co-run or lock tracing. Because an agent's toolset is bounded and tools recur across tasks, these per-tool profiles amortize. Second, a bounded tool set does not bound tool *behavior*: a single tool such as a shell or test runner can issue very different syscalls depending on its arguments and inputs, which are not known in advance. EoR is defined as an *expectation* precisely to accommodate this: rather than a point prediction, a tool can be characterized by a distribution of syscall behavior collected over many invocations, and placement can reason about expected interference.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their time and valuable feedback on this paper. This work was supported in part by PRISM, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] Ai agent sandboxing, 2026. https://edera.dev/use-case/ai-agent-sandboxing.
[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, February 2020.
[3] Anjali. *Isolation as a Quantifiable Resource in Modern Lightweight Virtualization*. PhD thesis, University of Wisconsin–Madison, 2026. https://www.proquest.com/dissertations-theses/isolation-as-quantifiable-resource-modern/docview/3373860803/se-2.
[4] Anthropic. Introducing the model context protocol, 2024. https://www.anthropic.com/news/model-context-protocol.
[5] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. https://arxiv.org/abs/2505.20411, 2025.
[6] Emir Beganović. Your container is not a sandbox. https://emirb.github.io/blog/microvm-2026/, 2026.
[7] Teofil Bodea, Masanori Misono, Julian Pritzi, Patrick Sabanic, Thore Sommer, Harshavardhan Unnibhavi, David Schall, Nuno Santos, Dimitrios Stavrakakis, and Pramod Bhatotia. Trusted ai agents in the cloud, 2025.
[8] Google Cloud. Choose your agentic ai architecture components. https://docs.cloud.google.com/architecture/choose-agentic-ai-architecture-components, 2026.
[9] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, July 2019.
[10] eunomia lab. Agentcgroup: Understanding and controlling os resources of ai agents, 2026. https://github.com/eunomia-bpf/agentcgroup.
[11] Dan Fernández. What is an ai agent? from chatbots to autonomous systems, 2025. https://edera.dev/stories/what-is-an-ai-agent-from-chatbots-to-autonomous-systems.
[12] gvisor, 2026. https://gvisor.dev/.
[13] Platform guide, 2026. https://gvisor.dev/docs/architecture_guide/platforms/.
[14] Hyper-v containers, 2026. https://learn.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container.
[15] The speed of containers, the security of vms, 2026. https://katacontainers.io/.
[16] Juhee Kim, Xiaoyuan Liu, Zhun Wang, Shi Qiu, Bo Li, Wenbo Guo, and Dawn Song. The attack and defense landscape of agentic ai: A comprehensive survey. https://arxiv.org/abs/2603.11088, 2026.
[17] Chad Kittel, Clayton Siemens, Hemavathy Alaganandam, James Lee, Ritesh Modi, Mahdi Setayesh, Mark Taylor, and Yaniv Vaknin. Ai agent orchestration patterns. https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/ai-agent-design-patterns, 2026.
[18] Janakiram MSV. Aws, microsoft, and google agree the session is the new unit of compute. they disagree on how to isolate it. https://thenewstack.io/agent-session-aware-runtime/, 2026.
[19] Nabla containers: a new approach to container isolation, 2026. https://nabla-containers.github.io/.
[20] NoLabs AI. nono: kernel-enforced sandboxing for AI agents. https://github.com/nolabs-ai/nono, 2026.
[21] NVIDIA. OpenShell: A safe, private runtime for autonomous AI agents. https://github.com/NVIDIA/OpenShell, 2026.
[22] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: large language model connected with massive apis. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.
[23] Ritik Raj, Souvik Kundu, Ishita Vohra, Hong Wang, and Tushar Krishna. Towards understanding, analyzing, and optimizing agentic ai execution: A cpu-centric perspective. https://arxiv.org/abs/2511.00739, 2026.
[24] Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.
[25] Multikernel Technologies. Sandlock, 2026. https://github.com/multikernel/sandlock.
[26] UncoverAlpha. The forgotten chip: Cpus the new bottleneck of the agentic ai era. https://www.uncoveralpha.com/p/the-forgotten-chip-cpus-the-new-bottleneck, 2026.
[27] David Wiesen. Building a safe, effective sandbox to enable codex on windows. https://openai.com/index/building-codex-windows-sandbox/, 2026.
[28] Wikipedia. Spearman's rank correlation coefficient, 2026. https://en.wikipedia.org/wiki/Spearman
[29] Ben Wodecki. Arm follows nvidia into a crowded enterprise cpu race. https://www.sdxcentral.com/news/arm-follows-nvidia-into-a-crowded-enterprise-cpu-race/, 2026.
[30] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. https://arxiv.org/abs/2210.03629, 2023.
[31] Yusheng Zheng, Jiakun Fan, Quanzhi Fu, Yiwei Yang, Wei Zhang, and Andi Quinn. Agentcgroup: Understanding and controlling os resources of ai agents. https://arxiv.org/abs/2602.09345, 2026.