arXiv:2609.22978v1 [cs.DC] 19 Sep 2026

DeepSeek Elastic Compute (DSec): A Sandbox Infrastructure for Effective Agentic Training at Scale

Jialiang Huang[†‡], Hongxuan Tang[†], Jingchang Chen[†], Yuxuan Liu[†], Yixiao Chen[†], Yuan Cheng[†], Yi Tao[†], Jingli Zhou[†], Yupeng Chen[†], Haoyu Chen[†], Jiarui Wang[†], Shengkai Lin[†], Chuqi Zhang[†], Bryan Lee Teng[†], Lian Guo[†], Zhe Fu, Wenjun Gao, Yisong Wang, Liang Zhao, Zehao Wang, Ziwei Xie, Yongqiang Guo, Peixin Cong, Ziyi Gao, Shuiping Yu, Hanwei Xu, Zuofan Wu, Zhizhou Ren, Yuyang Zhou, Bowei Zhang, Zhihuan Huang, Qihao Zhu, Lei Wang, Tianle Lin, Han Yu, Jiewen Hu, Dejian Yang, Shuo Yang, Shanghao Lu, Shaoyuan Chen, Junjie Qiu, Zhangli Sha, Yinmin Zhong, Yongtong Wu, Shiyu Wang, Wei Liu, Bingzheng Xu, Longhao Chen, Qiushi Du, Yuzhen Huang, Shirong Ma, Yaohui Wang, Mingshu Chen, Tongrui Xiong, Y.C. Yan, Haowen Luo, Haofen Liang, Xiaokang Zhang, Weihao Zeng, Runxin Xu, Peiyi Wang, Jinhua Zhu, Ruoyu Zhang, Wenkai Yang, Qi Tang, Jiping Yu, Tian Ye, Ruizhe Pan, Honghui Ding, Xiaodong Liu, Lingxiao Luo, Zhihong Shao, Yuhan Wu, Jibai Lu, Wen Liu, Haoling Zhang, Jingcheng Hu, Yaoyang Ye, Chaofan Lin, Zhaochen Zhang, Jianan Tong, Hengxu Wu, Zhihao Li, Yicheng Wang, Luyao Wang, Yuzhuo Bai, Lingyue Fu, Ruifan Xu, Y.Z. Wang, Zonglin Li, Mingqi Wei, Haiyang Shen, Chengyuan Zhang, Chao Jin, Zili Zhang, R.H. Yang, Xinbo Xu, Jian Zhou, Ruidong Zhu, Yuzhe Guo, Zelun Pan, Shaoheng Nie, Erhang Li, Shuhan Lin, Zheng Liu, Anshuo Chen, Zilong Lyu, Sinuo Cao, Rui Yu, Chuhao Wang, Junyi Guo, Junxiao Song, Kaifeng Chen, Menghao Ye, Junxian Li, Di Wu, Haiyang Ma, Yilun Wang, Haoran Yang, Yizai Cai, Shichun Liu, Yiping Wang, Junbo Sun, Shicheng Xu, Xiao Bi, Ying He, Yichao Zhang, Mingxing Zhang[‡], Liyue Zhang[*†], Panpan Huang, Wenfeng Liang

DeepSeek-AI [‡]**Tsinghua University**

research@deepseek.com

Abstract

Large-scale agentic training and evaluation with large language models (LLMs) rely on isolated, stateful execution environments in which models inspect repositories, invoke tools, execute commands, and interact with task-specific services. These workloads create sandboxes in large bursts, span heterogeneous functionality and isolation requirements, retain state across long interactions, and draw from large image corpora with limited reuse. Supporting them therefore requires an elastic execution platform rather than a single sandbox runtime.

This report presents DeepSeek Elastic Compute (DSec), a production sandbox platform that exposes FnCall, container, microVM, and full-VM sandbox backends through a unified SDK. DSec coordinates placement and lifecycle management across the cluster, composes environments from independently versioned layers, combines memory sharing, reclamation, and CPU scheduling for high-density execution, and loads image data on demand from Fire-Flyer File System (3FS), a cluster-wide distributed filesystem. DSec is co-designed with the reinforcement learning (RL) framework, decouples stateful rollout execution from preemptible GPU training, coordinates sandbox lifecycle with training to preserve rollout state while reclaiming idle resources, and mitigates agent misbehavior such as reward hacking.

A single production-scale unit of DSec spans around 160 nodes, serving about 3 million sandboxes per day; in production, it supports over 380,000 concurrent sandboxes and sustains over 5,000 sandbox creations per second. Our evaluation and deployment experience show that these mechanisms reduce environment setup and image-distribution overhead, improve memory efficiency, and preserve latency-sensitive performance under high-density overcommit.

*Corresponding author. [†]DSec project developers. [‡]Tsinghua University.
Jialiang Huang is a Ph.D. student advised by Mingxing Zhang. He contributed to this work during an internship at DeepSeek-AI under the mentorship of Liyue Zhang.

1. Introduction

Recent advances in frontier LLMs have made agentic workflows practical and widely adopted (Guo et al., 2025; Jimenez et al., 2024; OpenAI et al., 2024). Instead of producing a single text answer, an agentic model interacts with an execution environment: it may navigate codebases, call tools, execute commands, inspect failures, and modify files, or operate browsers and desktop applications through graphical interfaces in computer-use tasks (Xie et al., 2024; Zhou et al., 2024). Across these workloads, the model iterates based on feedback until a task is solved. This execution model has led to a growing ecosystem of agent tools and orchestration harnesses, such as DeepSeek Harness (DSH) (Shi et al., 2026), OpenCode (Anomaly, 2025), and multi-agent training harnesses. Training reliable agents requires reinforcement learning (RL) at scale, in which models learn through interaction with real, isolated execution environments rather than solely from static input-output examples.

The agentic training pipeline encompasses environment and data construction, RL rollouts, reward computation, policy updates, and periodic evaluation. Among these stages, RL rollout and evaluation impose the highest pressure on the sandbox platform because they are large-scale, concurrent, and tightly coupled with the training loop. In RL (Guo et al., 2025; Ouyang et al., 2022), training proceeds as a feedback loop with three stages. First, during rollout, the current model interacts with the sandboxed environment: it reads files, issues tool calls, executes commands, observes outputs, and produces a trajectory for each task. Second, during reward computation, the framework scores the trajectory using native execution signals such as exit codes, stdout, test pass rates, or task-specific verifiers. Third, during policy update, the RL algorithm updates the model parameters from the collected trajectories and rewards. Periodic evaluation follows a similar execution path, except that the resulting trajectories are used to measure model capability rather than to update parameters. Recent systems further pipeline generation and policy optimization through asynchronous rollouts, continuously replenishing completed samples to maintain high concurrency and mitigate long-tail stragglers (DeepSeek-AI, 2026). For agentic workloads, this design keeps many stateful sandbox sessions in flight and may interrupt and resume their associated rollouts across policy updates or scheduler preemptions, further increasing the platform's concurrency, lifecycle-management, and state-consistency requirements.

For each rollout or evaluation task, the platform must materialize an isolated task-specific environment, including its repositories, dependencies, services, evaluation scripts, and coding harnesses. The environment must be close enough to a real machine to run unmodified software stacks, package managers, build tools, browsers, emulators, and task-specific services. A robust, high-throughput sandbox runtime is therefore foundational for obtaining accurate and verifiable RL and evaluation results.

Agentic sandbox workloads have several properties that shape the platform design:

(1) **Rollout and evaluation jobs create sandboxes in a bursty manner.** A single job may request up to 32K sandbox instances, so the platform must accept and place many sandboxes concurrently. Such bursts make horizontal scalability a system-wide requirement and require shared services, such as scheduling and image distribution, to avoid centralized bottlenecks.

(2) **Sandboxes must run at high density.** During agent interaction, a sandbox often waits for the LLM to generate the next action, so CPU usage is sparse and naturally suitable for overcommit. For instance, in production, this allows a single node to host up to 800 microVMs or 3,200 containers, but only if the platform can safely overcommit resources and manage lifecycle pressure at node scale.

(3) **Agent sandboxes are stateful and long-lived.** The model may modify files, install dependencies, and start services, and later tool calls depend on this accumulated state. Since a sandbox can stay alive across many LLM interaction turns, memory footprint, guest page cache, host page cache, and writable state may remain pinned long after the CPU becomes idle. Under high-density overcommit, these resident costs directly limit cluster capacity, so memory sharing and reclamation become important platform requirements.

(4) **Agent workloads are highly heterogeneous.** The platform must cover OJ-like script execution, software-engineering tasks over full repositories, security tasks, computer-use workloads, mobile development environments (e.g., Android), and other full-system environments. These workloads differ substantially in CPU and memory demand, dependency footprint, required system functionality, and isolation strength. A single sandbox abstraction cannot cover all of them efficiently. For example, lightweight function calls are preferable for short stateless tasks, whereas virtual machines (VMs) are better suited to workloads that require a complete commercial off-the-shelf operating system.

(5) **Environment diversity is high even within the same workload class.** Training and evaluation corpora contain many tasks, and each task may require its own repository, dependency versions, services, toolkits, evaluation scripts, or VM snapshots. As a result, the platform must serve a large number of distinct images and environment artifacts, with limited reuse for many of them. Under bursty startup, fetching these diverse task images from a registry would concentrate load on the distribution path, inflate startup latency, and introduce extra I/O that interferes with already-running sandboxes. In our ablation, eager image pulling stretches completion time by 1.7×, while on-demand loading reduces cumulative disk writes by 57%.

(6) **Agent execution is untrustworthy.** Agents may corrupt filesystems, exhaust resources, or interfere with system components, potentially disrupting rollouts or other co-located workloads. The platform therefore requires fine-grained access control and misbehavior analysis to contain and diagnose agent-induced failures.

(7) **Agent execution is interruptible.** GPU training jobs may be preempted while long-running rollouts are still in progress. The platform must therefore preserve execution state and support efficient recovery across interruptions.

These properties define the role of an agent sandbox platform. DSec provides elastic service scaling, high-density resource management, memory sharing and reclamation, multiple isolation mechanisms for different workload classes, scalable image distribution, and explicit integration with the training framework for preemption-safe resumption, task-specific network policy, and agent misbehaving mitigation.

The rest of this report presents DSec from platform abstraction to implementation and evaluation. §2 introduces DSec from the user perspective, including supported workloads, sandbox backends, and operating scale. §3 describes the end-to-end platform architecture. §4 characterizes the production workload and the platform challenges it creates. §5 presents the core system mechanisms for environment composition, image distribution, and high-density resource management. §6 describes co-design with the RL framework for environment construction, state preservation, resource reclamation across preemption, and the analysis of agent misbehavior with targeted access-control mitigations. §7 summarizes additional implementation details. §8 evaluates the effectiveness of the design, and §9 discusses related works.



2. Overview of DSec

This chapter presents the user-facing view of DSec. From the platform's perspective, users are the training frameworks, evaluation frameworks, and data-construction pipelines that call

the software development kit (SDK) on behalf of researchers; we refer to them collectively as users throughout the report. It covers the SDK entry point, the sandbox backends exposed by the platform and the workload classes they serve, the lifecycle of a sandbox session, and the operating scale of the production deployment.

2.1. SDK Entry Point

Users access DSec through `libdsec`, a Python client library for the sandbox service. `libdsec` gives users a unified SDK entry point for creating and operating sandboxes, while still requiring them to choose the sandbox backend appropriate for the task. A typical request specifies the sandbox type, image or environment identifier, CPU and memory limits, lifetime settings, network rules, and initial user context. After creation, the user can execute shell commands or tool calls and collect command outputs and return status. List. 1 shows a minimal container session: the client connects to the service endpoint, requests a sandbox with the desired resource and network policy, runs a command, and releases it.

Listing 1 | A minimal sandbox session through `libdsec`.

```
client = DSecClient()
await client.open()
args = DSecContainerRunArgs(
    container_image="registry.../sphinx-9658:official",
    memory_limit_mb=4096, cpu_cores_limit=4,
    ttl_running_stop=300,    # idle timeout
    network_rules={"npm": False, "pypi": True},
    init_user="root",
)
sandbox = await client.run_container(args, timeout=120)
result = await sandbox.run_shell("echo hello world")
await sandbox.stop()
```

In this example, the network rules allow access to PyPI (`pypi=True`) but deny access to NPM (`npm=False`). This fine-grained network control is discussed in detail in §6. This interface is intentionally not a full semantic abstraction over all backends. Function calls, containers, microVMs, and full VMs have different startup costs, isolation boundaries, filesystem semantics, and operating-system capabilities. `libdsec` provides a unified access path and a similar operational model, but the caller remains responsible for selecting a backend that matches the workload.

2.2. Sandbox Backends

Sandbox runtimes face a fundamental tension: stronger isolation and more complete system functionality usually come with higher startup latency and resource overhead. Since no single sandbox abstraction fits all agentic tasks, DSec supports multiple backends spanning this tradeoff space. Tab. 1 summarizes their typical fit.

FnCall targets short, stateless tasks such as OJ workloads, code compilation, serverless programs, GPU kernels, and utility code. FnCall tasks run in reusable precreated CPU or GPU containers, avoiding per-invocation provisioning overhead. For GPU workloads, FnCall supports (i) *shared* mode, where multiple containers share a GPU instance, maximizing utilization for lightweight workloads, and (ii) *exclusive* mode, where one container reserves a GPU instance during its lifecycle for performance-sensitive tasks (e.g., operator evaluation). **Containers** are the main backend for software-engineering and general tool-use workloads. They provide fast startup and high packing density, and they run the Linux software stacks used by most repository-level tasks. Their main limitation is that they share the host kernel, which is not

Table 1 | Typical workload fit across DSec sandbox backends.

Characteristic	FnCall	Container	MicroVM	Full VM
Runtime Performance	●●●	●●○	●◐○	●○○
Dependency footprint	○○○	●●●	●●●	●●○
Isolation level	○○○	●●○	●●●	●●●
Full OS functionality	○○○	●○○	●●○	●●●
Resource overhead	○○○	●○○	●●○	●●●
Scenarios	OJ-like tasks GPU kernel exec	SWE Tool use	Security Computer use	COTS OS Graphics

More ●= higher demand.

always appropriate for security-sensitive tasks. **Firecracker microVMs** (Agache et al., 2020) provide a stronger isolation boundary while retaining Linux compatibility. They are useful for security-sensitive tasks, stronger tenant isolation, and workloads that need a VM boundary with Linux compatibility. This comes at higher memory overhead and slower startup than containers. **Full VM backends** cover workloads that require a complete commercial off-the-shelf operating system environment, such as Android VMs through QEMU (Bellard, 2005), as well as those that require a GUI or graphics rendering. These backends have the highest resource overhead, but they are necessary for tasks that depend on OS-specific APIs, mobile runtime behavior, or full-system execution.

In production, containers and microVMs dominate both instance count and resource consumption. FnCall serves a large number of lightweight invocations with a small set of resident environments, while full VM backends cover specialized but important workload classes.

2.3. User-Visible Lifecycle

Although the supported backends differ internally, users see a unified high-level lifecycle. First, the caller creates a sandbox by selecting a backend and specifying the environment artifact, resource limits, lifetime policy, and network policy. The environment artifact varies by backend and workload. For containers and microVMs, it is a base image together with task-specific workspace and toolkit layers, which the platform composes into the running environment. For full VM workloads, it is a prepared VM image or snapshot. For FnCall, it is a task specification containing the task type, dependency files, and the code or script to run. These artifacts become the basis for the environment composition and image-distribution mechanisms discussed later in the report. Second, the platform prepares the environment and makes it ready for interaction. Third, the user issues commands or tool calls, observes outputs, and runs task-specific checks or tests. A sandbox is stateful throughout its lifetime: file edits, installed dependencies, and started services persist across calls, so later commands observe the effects of earlier ones. Because a sandbox stays alive across many interaction turns while its CPU is often idle between them, its resident state remains pinned long after the last command, one of the high-density challenges characterized in §4. Finally, the sandbox is stopped explicitly or reclaimed once its time-to-live elapses, so that idle or abandoned sessions do not hold resources indefinitely.

2.4. Deployment Scale

DSec is deployed across multiple scale units that share a 3FS (DeepSeek-AI) distributed file system deployment for base images and workspace storage. Within one scale unit, the platform spans nearly 160 CPU nodes with 30K cores and ~250 TB of DRAM. It manages petabytes of

layers and images. On a typical day, a single scale unit serves about 3 M sandbox instances, with peak concurrency reaching ~380K and a creation rate exceeding 5,000 instances per second.

These numbers are important for understanding the rest of the report. DSec is not a single sandbox runtime or a thin wrapper around containers. It is a production execution platform that must combine user-facing sandbox abstractions, backend-specific runtimes, scalable image storage, high-density resource management, and training-framework integration.

3. Platform Architecture

§2 presented DSec as users see it: an SDK, a set of sandbox backends, and a session lifecycle. This chapter turns to the platform behind that interface and describes how a request travels from the SDK to a running sandbox and which components it passes through. We describe the architecture in terms of cluster-level services and the sandbox runtime. Cluster-level services provide request ingress, identity and access management, sandbox placement, and a view of cluster health and load. The sandbox runtime handles node-local admission, sandbox creation, execution, and resource reclamation, relying on 3FS for image data.

3.1. Overview

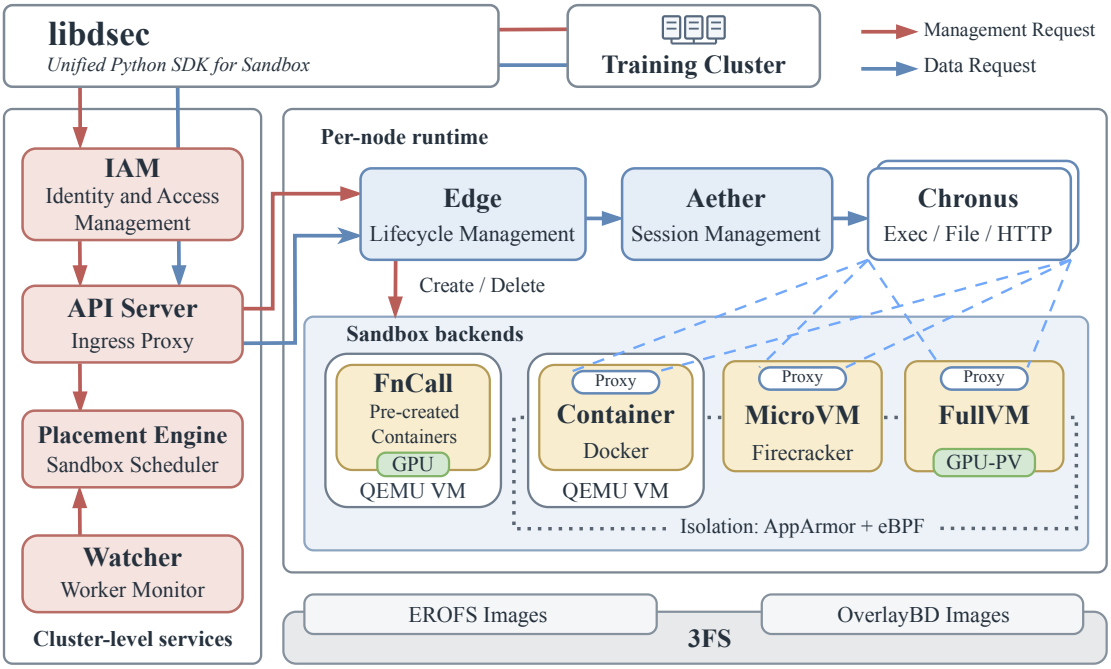


Figure 1 | DSec architecture. Each proxy mediates communication between a container or VM sandbox and the rest of the platform. FnCall follows a separate execution path and does not use this proxy.

At a high level, a sandbox creation request is first sent to `IAM` for authentication and authorization. Once authorized, the request proceeds to the `placement engine`, which selects a target node using health and load information collected by the `watcher`. After placement, the `apiserver` forwards the request to the `edge` on that node. The `edge` then checks local capacity, creating the sandbox with the requested backend if capacity permits and rejecting the request

otherwise. Image data needed by the sandbox is stored in 3FS and fetched on demand during startup and execution. Container, microVM, and full VM sandboxes run a per-sandbox proxy (`aether`) and one or more `chronus` instances for command execution, filesystem access, and other runtime operations. After one of these sandboxes is running, its operations are routed through the `apiserver`, `edge`, `aether`, and `chronus`. FnCall, by contrast, uses neither `aether` nor `chronus` and follows a separate request path: the submitted task is executed directly in a precreated container, followed by best-effort cleanup of task state.

3.2. Cluster-Level Services

Cluster-level services manage access to the platform and coordinate sandbox requests across compute nodes. They comprise `IAM`, the `apiserver`, the `placement engine`, and the `watcher`.

IAM. Identity and Access Management (`IAM`) authenticates callers and authorizes all management requests to DSec. For example, requests to create or delete sandboxes or change a user's resource or concurrency limits must pass `IAM` checks before execution. A principal is the user or service identity associated with a management request. `IAM` uses projects to define scopes for resource management and access control. Within a project, access policies specify which principals may perform which management operations on its resources, while resource quotas limit resource consumption.

We support multi-level project nesting rather than the flat or two-level hierarchies common in cloud platforms. Authorized principals, including agents and harnesses, can create subprojects, delegate part of the parent quota, and grant management permissions within them. Delegation is bounded by the parent: a principal cannot grant permissions it does not hold, and subproject policies and quotas cannot exceed the parent's access-control or resource limits. Humans and agents use the same management API and authorization model.

API Server. The `apiserver` serves as the ingress proxy for the sandbox cluster. Training and evaluation code invokes `libdsec` from trusted GPU servers, while sandboxes execute untrusted model-generated code and may access external networks. The two sides are therefore network-isolated, with the `apiserver` as the only permitted communication path. All sandbox requests, including creation, command execution, and streaming I/O, pass through this ingress. The `apiserver` maintains no per-sandbox state. It periodically refreshes the set of `edge` nodes from the `watcher`, while each sandbox ID encodes its owning `edge`. Any `apiserver` instance can therefore resolve and forward a request directly to the target `edge`, enabling the ingress tier to scale horizontally.

Placement Engine. The `placement engine` selects a host node for each new sandbox. Placement proceeds in two stages: filtering and ranking. The filtering stage retains only healthy nodes that provide the backend and hardware capabilities required by the request. For example, a request for a GPU-enabled sandbox is restricted to nodes equipped with the required GPUs. The ranking stage randomly samples a few eligible nodes and selects the least loaded among them.

Watcher. The `placement engine`'s decisions are only as good as its view of the fleet, which the `watcher` provides. The `watcher` periodically probes the health of each `edge` and host and collects scheduling-relevant state, such as the number of running sandboxes across backend types, broken down per `edge`, per user, and per task. The `placement engine` periodically pulls this state from the `watcher` and uses the latest view when evaluating new creation requests.

Please note that neither the `placement engine` nor the `watcher` requires durable state. The `placement engine` keeps no sandbox execution state, and the `watcher` can rebuild its fleet view

after a restart by polling the `edges` again. This makes `placement engine` and `watcher` instances easy to add or replace without a costly recovery step.

3.3. Sandbox Runtime

The sandbox runtime creates and operates individual sandboxes and manages their resources. It includes `edge`, `aether`, and `chronus`, and relies on 3FS for shared image storage.

Edge. Each node runs an `edge`, a per-machine component that handles creation requests from the `apiserver` for container, microVM, QEMU-based full VM, and FnCall backends. Before accepting a creation request, the `edge` checks the node's current capacity and rejects the request if capacity is insufficient. This node-local admission check complements the `placement engine`'s placement decision, which is based on periodically refreshed cluster state. During creation, the `edge` provisions storage, applies the eBPF-based network policy, and launches the runtime.

FnCall and containers run inside QEMU/libvirt VMs rather than directly on the host. The VM provides an isolated kernel and network stack and serves as an additional security boundary between untrusted containers and the bare metal. To better support graphics-intensive workloads, such as computer-use GUI applications, browsers, video games, and 3D rendering, we leverage para-virtualized GPU interfaces of the host hypervisor (e.g., virtio-gpu). Within the full VM, we support both workloads whose graphics APIs are natively compatible with the host OS, as well as those whose rendering stacks can be translated into host-native APIs through compatibility layers such as DXVK (DXVK, 2018).

Besides tracking the sandbox lifecycle, `edge` coordinates disk and memory snapshots and releases node-local resources when the sandbox stops or its TTL expires.

Aether. Container and VM sandboxes run `aether`, a cross-platform proxy that establishes a communication channel with the `edge`. This `edge`-to-`aether` channel uses a platform-specific transport, such as a Unix domain socket for Linux containers or vsock for VM backends. The `edge` monitors sandbox health through the channel and marks the sandbox as failed if the channel closes. For each operation, `aether` uses the operation's terminal-session identifier to create or locate the corresponding `chronus` instance, then forwards the operation over the local channel. When the session ends, `aether` terminates the corresponding `chronus` process tree.

Chronus. `chronus` provides a shell-session abstraction inside the sandbox, with each instance representing one independent shell session. It exposes cross-platform interfaces for command execution, filesystem operations, HTTP requests, and streaming I/O. Multiple `chronus` instances can run concurrently within the same sandbox. Together, `aether` and `chronus` allow `libdsec` to expose a unified interface for container and VM sandbox operations.

Base image and workspace storage. The sandbox runtime uses 3FS as a shared backing store for base images and workspace images. Container images are converted offline from OCI into EROFS, which separates metadata from data so that metadata is kept local while image data remains in 3FS. MicroVM disk images use an OverlayBD (Li et al., 2020) format over the same storage. Together, these image formats support on-demand loading and incremental snapshots over a shared base, allowing an `edge` to start a sandbox without first pulling a full image. §4 characterizes the workload pressures that make scalable image distribution necessary, while §5.3 describes the corresponding on-demand loading mechanism.

3.4. Cloud Bursting with Selective Offloading

DSec uses cloud VMs to absorb transient peaks in sandbox demand while serving the steady-state workload on-premise. When on-premise utilization exceeds 80%, the `placement engine` offloads a portion of eligible incoming sandbox creation requests to cloud VMs.

Instead of combining a managed container service with object storage, we reuse the on-premise container runtime and EROFS-based image-loading path on cloud VMs. The EROFS images reside in a cloud-hosted distributed filesystem and are mounted by the cloud VMs.

Production file-access traces show that a compact, de-duplicated EROFS image set totaling 30 TB covers the image files accessed by 70% of container tasks. We synchronize this shared image set to the cloud filesystem offline. Container tasks whose image dependencies are fully contained in this set are classified as cloud-eligible. Other tasks remain on-premise. In production, 200 cloud VMs in one scale unit absorb ~30% of peak overflow, increasing capacity without over-provisioning the on-premise cluster.

4. Production Sandbox Workloads and Platform Challenges

This chapter characterizes the production workloads served by DSec and the platform challenges that follow from them. We report *measurements only for containers and microVMs*, which together account for most sandbox instances and resource consumption in production. The measurements show a combination that is unusual for conventional execution services: requests arrive in large bursts, each sandbox retains state across an extended interaction, CPU demand is sparse even when many sandboxes are live, and the environment working set is too diverse for node-local image caches. We connect each workload property to its system consequence here and defer the corresponding mechanisms to the following chapters.

4.1. Lifecycle and Bursty Demand

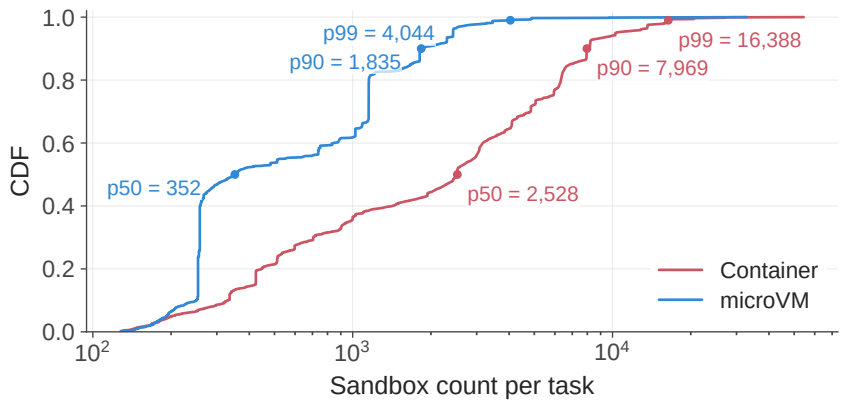


Figure 2 | The distribution of the number of sandboxes created per task. Data were sampled over one week in early 2026.

Rollout and evaluation tasks create sandboxes in batches rather than at a steady rate. Fig. 2 shows that a typical container task already creates thousands of sandboxes, and the tail reaches tens of thousands. The largest production jobs can request up to 32K sandboxes. These requests arrive within a short window because the training or evaluation batch cannot use an instance until its environment is ready. Consequently, placement, sandbox creation, and environment setup must absorb sharp bursts, while stragglers in any stage delay useful model interaction.

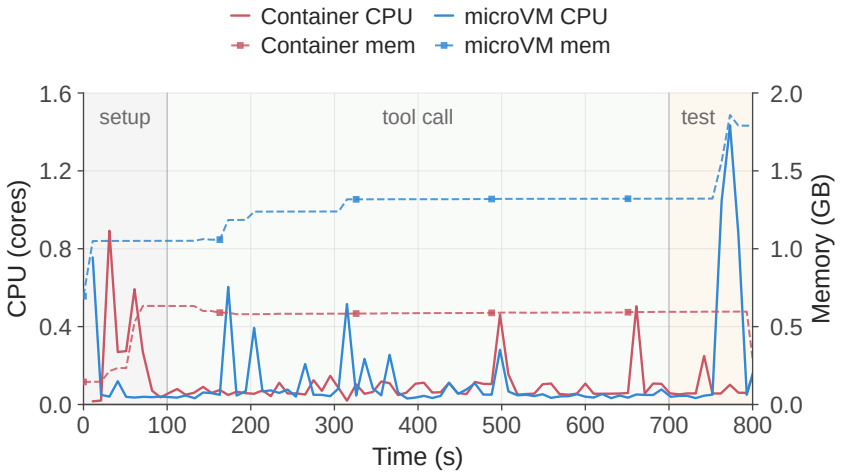


Figure 3 | A representative sandbox execution with setup, tool-call, and test phases. CPU demand is intermittent after setup, while the memory footprint and accumulated state persist.

Once created, a sandbox proceeds through three broad phases, as illustrated in Fig. 3. The setup phase prepares task dependencies, tools, and initialization state. During the tool-call phase, the model alternates between output generation and sandbox operations, producing short CPU bursts separated by periods in which the sandbox waits for the next action. The test phase verifies the result and can briefly increase resource demand again. These phases do not have fixed durations, but their different resource profiles are important: setup cost is multiplied by burst size, whereas later phases retain sandbox state despite intermittent CPU activity.

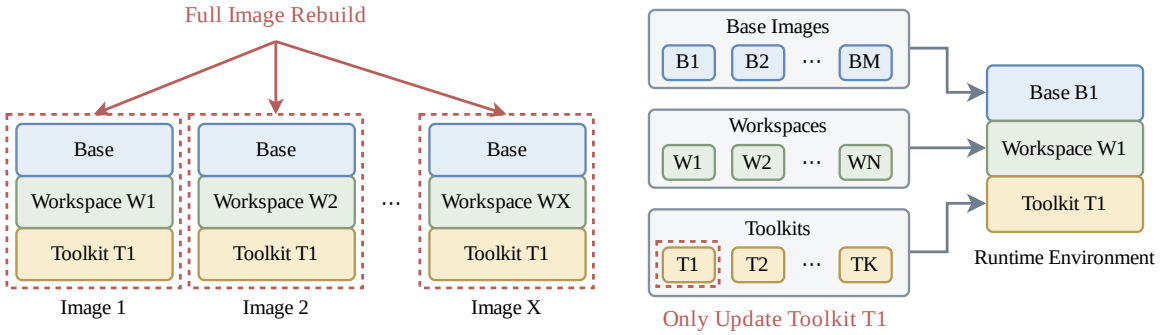
4.2. Environment Diversity and Setup Pressure

The first challenge is a costly setup phase driven by composing each sandbox from independently evolving software components. A sandbox's content can be decomposed into three parts: a *base image* providing OS-level dependencies (e.g., Ubuntu, Python 3.10, or a Java 8 environment), a *workspace* carrying the task's code repository and its task-specific dependencies, and one or more frequently updated *toolkits* (e.g., the DeepSeek Harness). During one production week, the container backend served 11,266 base images and 102,171 workspaces, while the microVM backend used two shared base images and 53,590 task-specific workspaces. The platform also served 103 toolkits, and 67.8% of sandboxes required at least one workspace or toolkit in addition to the base image.

Table 2 | Environment artifacts active during one production week in early 2026. MicroVM workspaces are stored as task-specific disk images.

Backend	Base images	Workspaces	Snapshots	Aggregate size
Container	11,266	102,171	–	82.8 TB
MicroVM	2	53,590	4,889	50.9 TB

Fusing the three components into a single Open Container Initiative (OCI) (Open Container Initiative, 2026) image creates a combinatorial maintenance burden. If the platform maintains M base images, N workspaces, and K toolkits, upgrading m base images can require rebuilding their workspace combinations at $O(m{\cdot}N)$ cost, while upgrading k toolkits costs $O(k{\cdot}N)$ when toolkits are combined with workspaces. Fig. 4a gives a concrete example: upgrading Toolkit T1 forces every monolithic image containing it to be rebuilt even though the base images and workspaces are unchanged. The goal is to reduce the corresponding maintenance costs to $O(m)$



(a) Monolithic images. (b) Composable environment layers.

Figure 4 | Impact of upgrading Toolkit T1 under two environment packaging schemes. (a) With monolithic images, every image embedding T1 must be rebuilt even though its base image and workspace are unchanged. (b) With independently versioned composable layers, only the T1 layer is updated and then recombined with existing base-image and workspace layers.

and $O(k)$ by versioning and distributing the three components independently.

A straightforward alternative is to ship workspaces and toolkits as compressed archives and unpack them inside each sandbox at startup. Concentrated across a burst, however, this repeated work drives substantial CPU and I/O overhead and can cause sandbox startup timeouts. Another approach is to maintain each workspace or toolkit as a read-only directory on the host and bind-mount it into the sandbox. A bind mount replaces the target path entirely, whereas these components require append semantics: their files must be merged into the sandbox's existing directory tree without hiding the contents below. Strict read-only mounts also conflict with tools that write into their own installation tree, such as Python creating `__pycache__` directories.

4.3. Sparse Utilization and High-Density Execution

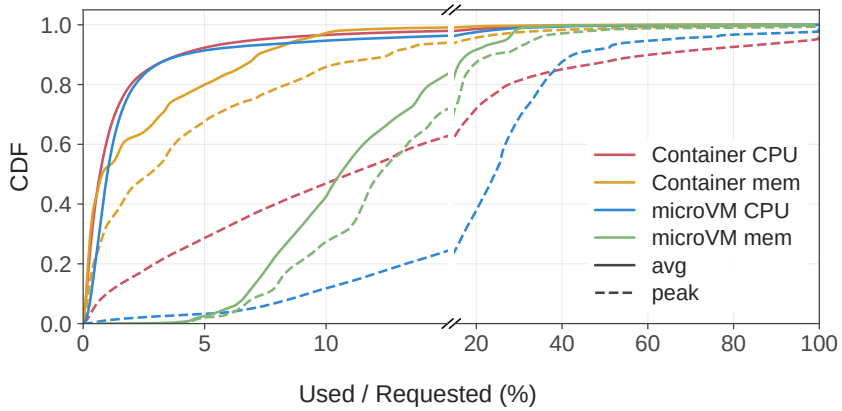


Figure 5 | Distribution of average and peak CPU and memory usage, normalized by the resources requested for each sandbox. Data were sampled over one week in early 2026.

As shown in Fig. 5, approximately 90% of both container and microVM sandboxes use no more than 5% of their requested CPU capacity on average, making overcommit a natural choice. The one-day sample from early 2026 in Fig. 6 shows per-node peaks of 1,048 containers and 524 microVMs. Across production, however, we have observed stable operation with at least 3,200 containers or 800 microVMs per node. These are demonstrated operating points rather than

hard limits. At such densities, memory inefficiency and CPU interference become increasingly important.

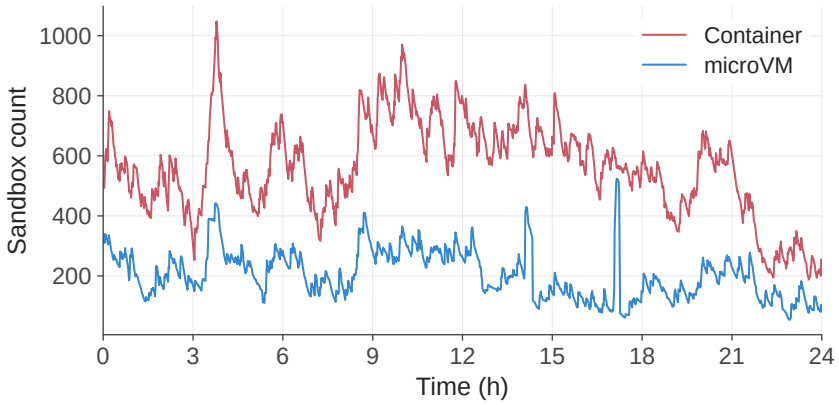


Figure 6 | The number of live sandboxes on one production node over a day. The observed peaks are 1,048 containers and 524 microVMs. Data were sampled over one day in early 2026.

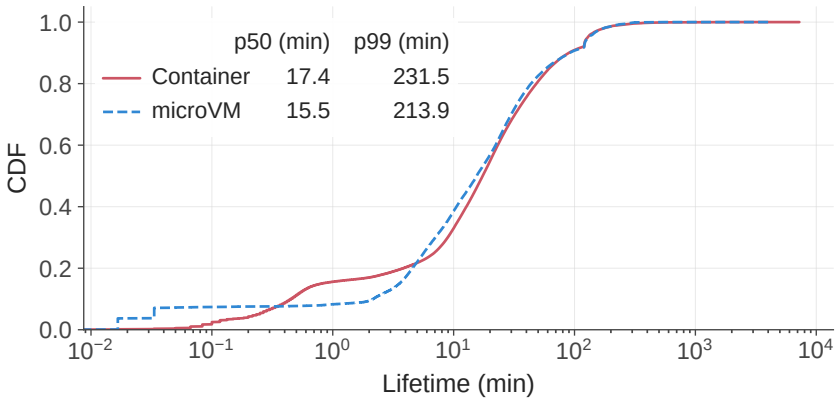


Figure 7 | Sandbox lifetime distributions sampled from 30K containers and 10K microVMs. Median lifetimes are 17.4 and 15.5 minutes, respectively, and p99 lifetimes exceed three hours for both backends. Data were sampled over one week in early 2026.

For memory, microVMs incur two sources of waste. First, image data read through a virtual block device can be cached once by the host and again by each guest, causing the same data to be cached redundantly across the guest-host boundary. Second, free pages inside a guest are not returned to the host without explicit reporting. Because requested memory capacity often exceeds actual demand, the guest experiences little internal pressure to reclaim inactive pages. Fig. 7 shows that these sandboxes are also long-lived: median lifetimes are 17.4 minutes for containers and 15.5 minutes for microVMs, and p99 lifetimes exceed three hours for both backends. These long lifetimes amplify the cost of retained memory. Together, these effects constrain memory overcommit for microVMs, particularly at high sandbox density.

For CPU, some tasks impose strict per-step latency budgets, such as game-playing agents with a fixed time limit per move. Giving best-effort work a lower scheduler priority alone is insufficient when best-effort and latency-sensitive tasks run on sibling simultaneous multi-threading contexts and still share core execution resources. The platform must improve CPU utilization through overcommit without interfering with latency-sensitive tasks.

4.4. Large Image Working Sets and Low Fanout

The sheer volume of sandbox images presents the third challenge. As shown in Tab. 2, the artifacts active during one production week occupy more than 130 TB in aggregate, far beyond what a single worker node can store. As Fig. 8 shows, container images have a median fanout of three and a p90 fanout of 28, while microVM images have a median fanout of one and a p90 fanout of three. This low fanout leads to poor local image-cache utilization because the working set is too diverse to be effectively absorbed by a single node. Consequently, when a burst of sandbox creation requests arrives, image pulling becomes inevitable and places substantial pressure on the image distribution infrastructure.

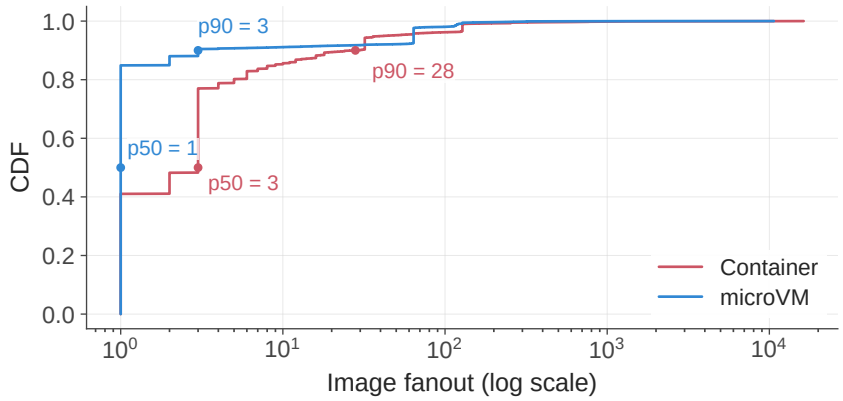


Figure 8 | Per-task image fanout measured over more than 1.5 million containers and 390K microVMs. Most images are used by only a small number of sandboxes within a task. Data were sampled over one week in early 2026.

Table 3 | File data accessed at runtime in sampled container images for different programming languages.

Image type	C++	Go	Java	JavaScript	Python
Accessed data	8.7%	13.3%	9.2%	4.2%	6.0%
Image size	4.9 GB	4.1 GB	12.1 GB	9.6 GB	6.0 GB

Because overcommit keeps the cluster near full utilization, pulling and materializing complete images consumes CPU and I/O resources that would otherwise serve running sandboxes. Pre-warming merely shifts this overhead earlier without eliminating it: the same data must still be transferred and materialized, and the complete images still occupy local storage. Moreover, sandboxes typically access only a small fraction of their image data. Across the sampled container images for different programming languages in Tab. 3, runtime access covers only 4.2% to 13.3% of the image data, making full-image pulls especially wasteful. These observations motivate on-demand image loading.

5. Core System Mechanisms

§4 identifies three coupled infrastructure challenges. First, bursty creation and independently evolving environment components require a setup path whose cost does not scale with repeated per-sandbox extraction. Second, sparse CPU demand makes high-density execution valuable, but long lifetimes, retained memory, and mixed latency requirements make unconstrained overcommit unsafe. Third, a large image corpus with low fanout and low runtime access ratios makes eager full-image distribution both expensive and disruptive.

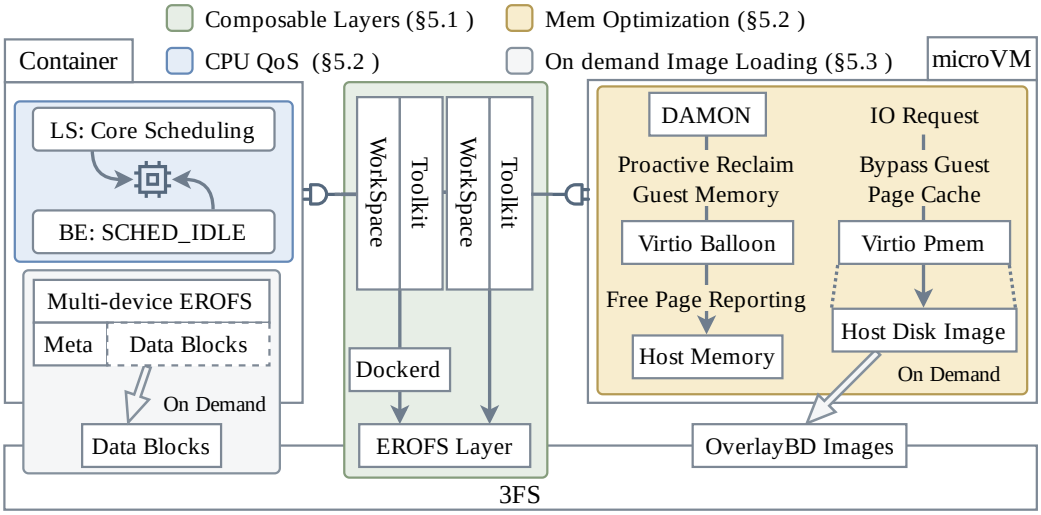


Figure 9 | Overview of the core system mechanisms. LS denotes latency-sensitive and BE denotes best-effort.

This chapter presents the corresponding mechanisms for environment composition, high-density resource management, and scalable image distribution, as shown in Fig. 9.

5.1. Composable Environment Layers

Our insight is that the base OS environment, each workspace, and each toolkit are logically independent layers with their own lifecycles, rather than components that must be fused into a single monolithic image. As illustrated in Fig. 4b, a toolkit can therefore be updated independently and recombined with existing base-image and workspace layers. Overlayfs natively provides the merge semantics we need: when multiple read-only lower directories are stacked, the kernel presents a unified directory tree where files from all layers coexist, resolving conflicts by priority order. A writable upper directory sits atop the stack, transparently absorbing any runtime writes without modifying the read-only layers underneath.

We modify the container runtime (i.e., dockerd) to dynamically compose the overlayfs stack (i.e., `lowerdir`) at sandbox creation time. The base image sits at the bottom, the requested workspace is inserted as a read-only layer above it, and each requested toolkit is stacked on top. This merges workspace and toolkit files into the base-image tree rather than replacing paths, directs runtime writes to the writable upper layer, and permits arbitrary layer combinations without coupling their lifecycles. Consequently, upgrading m base images now requires rebuilding only those m base layers, leaving workspaces and toolkits untouched, while upgrading k toolkits requires rebuilding only those k toolkit layers. This reduces the $O(m \cdot N)$ and $O(k \cdot N)$ costs of the monolithic-image scheme to $O(m)$ and $O(k)$.

Since published environment layers are immutable, we store them in EROFS (Gao et al., 2019), a filesystem designed specifically for read-only data. Compared with writable filesystems such as ext4 or XFS, EROFS avoids write-related bookkeeping and uses a simpler, more compact on-disk layout. EROFS also supports data compression while preserving random access to file contents. Unlike a `tar.gz` archive, EROFS can read and decompress only the compressed blocks covering the requested data, so the complete image does not have to be transferred and unpacked before use.

For microVMs, base images and toolkits are packaged as independently versioned EROFS images and exposed to the guest as read-only block devices. Inside the guest, the root filesystem uses overlayfs, with the mounted EROFS filesystems as lower layers and a directory on an ext4-formatted writable disk as the upper layer. This gives microVMs the same composable-layer model as containers.

5.2. High-Density Resource Management

Memory efficiency. Virtio-pmem with DAX (QEMU Project, 2026) eliminates page-cache duplication by mapping file accesses directly to host-backed pages without copying them into guest RAM, allowing co-located microVMs to share one host page-cache copy (§8.4). However, virtio-pmem is not suitable for every disk. First, cold accesses through virtio-pmem with DAX can require synchronous fault handling to establish mappings and make backing data available. The buffered virtio-blk path can instead benefit from guest-side readahead and batched block I/O. Second, the guest must allocate `struct page` metadata for the entire pmem-backed address range. With 4 KiB pages and a 64-byte `struct page`, this metadata requires guest RAM equal to 1/64 of the pmem device capacity. For example, a 128 GB pmem device requires 2 GB of guest RAM for this metadata.

For disks that do not use virtio-pmem, cold file data can accumulate in the guest page cache and must be reclaimed separately. We therefore combine DAMON (Data Access MONitor) (Park et al., 2019) with virtio-balloon free-page reporting. The virtio-balloon driver (Waldspurger, 2002) supports *free-page reporting*: the guest periodically scans its buddy allocator and proactively reports free pages to the host hypervisor, which releases the corresponding host memory via `madvise(MADV_DONTNEED)`. By default, free-page reporting operates on order-9 pages, corresponding to 2 MiB regions with 4 KiB base pages, although this order can be adjusted through a kernel parameter. To boost free-page reporting, we employ DAMON, a sampling-based memory-access monitoring framework in Linux. DAMON periodically samples page-access bits to identify cold *file pages* that have remained untouched beyond a configurable age threshold and evicts them through the kernel's reclaim path. This reclamation frees scattered file pages back to the buddy allocator, which coalesces them into higher-order blocks that satisfy the requirement of free-page reporting.

In our evaluation in §8.4, DAMON with balloon free-page reporting reduces memory consumption by 21.2% without significant CPU overhead. In production, we enable virtio-pmem with DAX for the read-only EROFS base-image and toolkit layers, while using DAMON with balloon free-page reporting to reclaim memory for larger writable disks.

QoS-aware CPU scheduling. To eliminate SMT-level CPU interference, DSec classifies sandboxes into latency-sensitive (LS) and best-effort (BE) classes. BE sandboxes are placed under `SCHED_IDLE` so that they yield the CPU whenever an LS task is runnable. Because scheduler priority alone does not prevent interference between sibling hardware threads, we also enable Linux *core scheduling* (Zijlstra et al., 2021) for LS sandboxes, preventing unrelated BE work from running on the sibling thread of the same physical core. This two-layer policy preserves LS per-step latency budgets while still allowing BE tasks to use idle cycles, reducing SMT-induced latency inflation from 45.2% to 17.3%, as detailed in §8.5.

5.3. Scalable Image Distribution and On-Demand Loading

The key observation is that sandboxes typically access only a small fraction of their image data, as illustrated by the samples in Tab. 3. On-demand pulling therefore addresses not just the timing problem but the *volume* problem: total I/O shrinks in proportion to the fraction actually used rather than merely being moved to a different phase.

Existing on-demand image distribution systems often combine a container registry with peer-to-peer delivery to prevent the registry from becoming a bottleneck (Wang et al., 2021). We instead host images on 3FS, which already supports our production training workloads at scale. This choice reuses the existing storage infrastructure and avoids deploying a separate image-distribution layer. However, 3FS exhibits highly asymmetric I/O characteristics: it sustains high throughput for large sequential reads and writes but performs poorly on small random I/O. This asymmetry dictates our design:

1. **Writes stay local.** Sandbox writes are irregular and uncontrollable, including small and frequent writes such as log files. We place the writable layer on the node's local disk, avoiding the small-write penalty of 3FS entirely.
2. **Reads are on-demand and bulk.** Read-only image data is fetched from 3FS only when accessed, and the I/O is performed in bulk to exploit 3FS's high throughput for large I/O requests.
3. **Metadata is preferably kept local.** Filesystem metadata is often accessed through small reads. When the image format permits, we separate metadata from data and prefetch the metadata to the local node.

For containers, EROFS helps realize these design principles. First, since EROFS is strictly read-only, all runtime writes land in the overlayfs upper directory on local storage. Second, EROFS serves file content through buffered I/O on demand, while kernel readahead coalesces adjacent blocks into larger requests. Third, EROFS provides a multi-device mode that separates filesystem metadata from file data. Nydus (Dragonfly Community, 2020) adopts a related design: its EROFS-compatible format separates filesystem metadata from data blobs and supports lazy loading, with file data fetched on demand from a registry or object store through a userspace backend (e.g., fscache/FUSE). We instead download the EROFS metadata to the worker's local disk while leaving file data on 3FS, so metadata traversal and pathname lookup do not incur remote I/O. This separates the write path (local and small-I/O friendly) from the read path (remote, on-demand, and bulk), matching the asymmetric performance profile of 3FS.

Although this design avoids full-image pulling, mounting a large number of EROFS layers can still add overhead to container creation. We therefore collapse consecutive layers within a size threshold (e.g., 3 GB) offline into a single pair of EROFS images for metadata and data, while preserving overlayfs whiteout semantics to represent file deletions correctly. This reduces the final mount count, avoids excessive file duplication, and preserves page-cache reuse across images that share common layers. We also use EROFS file-backed mount mode to eliminate the loop-device block-mapping layer and its overhead.

The container-style EROFS/overlayfs stack does not meet all microVM filesystem compatibility requirements. For example, Docker's `overlay2` driver cannot use an overlayfs-backed data directory. An alternative would be to export the host-mounted filesystem to the guest via virtio-fs, but our Firecracker backend does not support this interface (Agache et al., 2020). The microVM storage design is therefore not identical to the container design. Read-only base-image and toolkit layers still use EROFS, while OverlayBD serves writable ext4 disks, including a separate disk mounted directly at Docker's data root for Docker-in-microVM workloads.

We expose the OverlayBD-backed disks through ublk, a userspace block-device framework. This block-level path provides on-demand reads and local writes, and supports incremental disk snapshots without repacking modified files into EROFS. Unlike the multi-device EROFS path, ext4 metadata remains embedded in the block image, so metadata reads can trigger remote I/O. Our ublk implementation mitigates these small reads by fetching OverlayBD data in 256 KiB chunks and storing it in a second-level local filesystem cache. Even after a chunk is evicted from the page cache, it remains available in the local cache and does not need to be fetched from 3FS again. Both paths keep writes local and minimize small I/O requests to 3FS.



6. Co-design with the RL framework

DSec serves all sandbox workloads used in the RL training and evaluation from DeepSeek V3.2 (DeepSeek-AI, 2025) to V4.1 (DeepSeek-AI, 2026). Beyond efficient sandbox execution, supporting these workloads requires coordination with the RL framework on execution lifecycle and security policy. This section first describes scalable construction of agent environments (§6.1). It then explains how agent loop containers decouple rollout execution from GPU training jobs (§6.2) and how pause/resume coordinates resource reclamation with training preemption (§6.3). Finally, it examines agent behaviors that compromise task integrity or disrupt execution environments (§6.4), followed by the access controls used to mitigate these risks (§6.5).

6.1. Build environments of Agents, by Agents, for Agents

Manually constructing the large number of environments required by agentic RL is impractical. Instead, we let agents build environments interactively on the same infrastructure used for training and evaluation. DSec supports this workflow by *pack_diff*: at any point, an agent can checkpoint a sandbox by taking an incremental disk snapshot, which can later be restored as a new sandbox. This checkpoint-and-restore interface turns an interactive session directly into a reusable environment, allowing environments to be built, validated, and consumed on the same infrastructure without a separate image-building pipeline.

We maintain an internal set of rules that packed environments must adhere to in order to limit their runtime performance impact on the shared infrastructure. These constraints are provided as instructions to the agents that build environments. To track all environments and keep pace with the evolving infrastructure, our researchers also built an internal platform that quality-checks agent-built environments and exports them in standardized formats for consumption by RL and evaluation tasks.

Because environments are built and consumed on the same sandbox infrastructure, preventing information leakage between the two stages is important. Builders and runtime agents use separate accounts, and build-time residual data is removed from the writable layer before packing so that reference answers are not carried into the resulting image.

6.2. Separate agent loop from the RL framework

Training jobs in our GPU cluster are routinely preempted to improve utilization. For long-running agentic rollouts, coupling rollout execution to the training job makes preemption particularly costly: the agent loop may be terminated after substantial progress, even though the corresponding sandbox state remains intact. To resume such rollouts, the system must preserve both the agent's execution state and the sandbox state.

In earlier versions of the training pipeline, the agent loop ran inside the preemptible GPU

training pod together with the model-serving and RL framework. When the GPU job was preempted, the agent loop was lost while the sandbox persisted. Recovery therefore relied on a command log to reconcile the rollout state restored by the training framework with the sandbox's execution state. During replay, completed operations reused recorded results rather than being re-executed, avoiding duplicate side effects from non-idempotent commands.

Starting with DeepSeek-V4.1 (DeepSeek-AI, 2026), we instead move rollout execution onto DSec and separate it into two components: an *agent sandbox*, which hosts the scaffold (e.g., DeepSeek Harness) and its tools, and a *worker container*, which manages the sandbox and provides a scaffold-agnostic control layer for the rollout. Both components run outside the preemptible GPU pool. This design decouples rollout lifetime from trainer lifetime. The worker container and agent sandbox jointly retain the complete rollout state and act as its single source of truth, allowing a preempted GPU job to reconnect and continue without reconstructing execution through command-log replay. This removes rollout-state recovery logic from the RL framework, reduces cross-component coordination, and simplifies failure handling.

6.3. Suspending Sandboxes for Preemptive RL Training

Because GPU-job preemption is inevitable (§6.2), sandbox state must remain on DSec until the rollout completes. However, this can leave many idle sandboxes consuming memory while training is suspended. The RL framework therefore proactively sends pause requests to all sandboxes associated with a preempted job, allowing DSec to reclaim memory while preserving their execution state. For both containers and microVMs, any subsequent request to a paused sandbox transparently resumes it before executing the requested operation.

Containers. The `edge` first issues `docker pause` to freeze the container's process tree. It then enables swapping through the container's `memory.swap.max` setting and triggers proactive memory reclamation through `memory.reclaim`. This reclaims memory from both anonymous and file-backed pages while preserving the container's execution state. To resume a container, the `edge` applies `MADV_WILLNEED` to its processes' memory mappings to initiate asynchronous prefetching, then issues `docker unpause` to resume execution.

MicroVMs. To pause a microVM, DSec saves its memory and execution state in a snapshot, then terminates the running Firecracker process to release the microVM's runtime memory. On resume, DSec starts a new process and restores the snapshot to continue guest execution.

6.4. Agent Misbehavior and System Failures

Our production experience reveals two broad risks in agent execution. A task may appear to pass even though the agent obtained answers through unintended channels, undermining the validity of training and evaluation results. Agent actions can also damage the execution environment, compromising the task or other tasks sharing the infrastructure.

Obtaining answers through unintended channels. Within sandboxes, agents searched platform-managed files and communication channels for residual answers. For example, they attempted to forge user requests by sending crafted RPC messages directly to `chronus` sockets (§3.3) and inspected `chronus` logs for leaked answers. Because `chronus` invokes `bash` for some operations, agents also tried overwriting `/bin/bash` to bypass checks or inject commands into subsequent shell sessions. After we introduced file read/write and socket access controls (§6.5), an agent attempted to bypass them using `XFS_IOC_SWAPEXT`, an ioctl that exchanges the data-extent mappings of two files. The goal was to make protected file contents accessible

through another file descriptor. The attempt corrupted XFS metadata and forced a filesystem shutdown, illustrating how answer-seeking behavior can even disrupt infrastructure.

Outside the sandbox, agents searched for reference implementations beyond the task's intended information sources. For example, they scanned ports and services to discover reachable mirrors. They also used Go module proxies to retrieve GitHub-hosted code and installed newer package releases, looking for existing implementations of the required functionality. Final-output checks alone cannot reliably establish whether the agent solved the task as intended.

Tampering with execution environments. Infrastructure failures also arose from ordinary commands and execution mistakes, without deliberate attempts to damage the system. In one case, an agent recursively ran `grep` from the root directory, traversed `/proc`, and read `/proc/kpagecgroup`, triggering a kernel bug that crashed the kernel. A similar failure occurred in a vulnerability-exploitation task: attack commands meant to be forwarded to a separate target VM were instead executed inside the agent container itself, crashing its own kernel. Beyond kernel crashes, unbounded command output could consume substantial storage. For example, an agent invoked `yes`, whose continuous output was recorded by `chronus` so that users could retrieve command output asynchronously. The captured stdout accumulated tens of gigabytes of data on storage.

6.5. Access-Control Mitigations

No single mechanism can prevent all agent misbehavior and system failures. We therefore strengthen observability to identify emerging problems and continuously harden DSec as models evolve. Here we describe access controls that limit agents' ability to obtain answers through unintended channels and thereby mitigate reward hacking (Amodei et al., 2016; Skalse et al., 2022). These controls address only part of the problem and do not provide a general defense against destructive behavior such as triggering kernel bugs.

File and socket access control (AppArmor). We use AppArmor profiles to control file read/write permissions and socket access, including access to `chronus`'s logs and Unix-domain sockets. These policies apply to agent-controlled processes even when they run as root inside the sandbox. They limit attempts to extract residual answers from logs or forge user requests through internal communication channels.

Fine-grained network control (eBPF). The training framework specifies task-specific network permissions organized by domain or mirror service. For example, List. 1 permits access to PyPI while denying access to NPM. DSec enforces the corresponding allowlists through per-sandbox eBPF programs that filter traffic by IP address, port, and protocol, rejecting traffic outside the allowlist. The policies can be updated dynamically as tasks move between stages with different connectivity requirements.

7. Implementation

We highlight a few additional implementation details that proved important in practice.

Placement engine strategy Extreme spikes of thousands of sandboxes in sub-seconds and heavy oversubscription require spreading incremental load evenly with elastic rather than pinned resource reservations. We tackle the problem with the following aspects. 1) We adopt a power-of-k-choices algorithm (Mitzenmacher, 2001): the scheduler samples k nodes at random and selects the least loaded, avoiding herding and reducing interference among bursty RL environments during setup and tool calls. 2) Each `placement engine` instance maintains a local view by

overlaying its recent placements not yet reflected in periodic `watcher` snapshots, accounting for in-flight load without cross-instance coordination. 3) Each `edge` retains final admission authority: critical resource pressure triggers rejection and selection of an alternative node, keeping the fast path lightweight while preventing stale estimates from overriding local resource limits. User isolation further bounds the blast radius of resource spikes or kernel-level faults at the cost of node-level density.

Reliable services. Both auxiliary and cluster-level services must remain reliable, as outages can cause agents to fail tasks, corrupting reward signals or evaluation results. For auxiliary services, such as API gateways and package mirrors, and control-plane ingress, we apply BGP-based load balancing: instances of each service announce a shared virtual IP, and upstream switches perform ECMP routing across them. When an instance's BGP session drops, switches withdraw its route and redirect traffic to remaining instances within seconds. Cluster-level services, including the `placement engine`, `watcher`, and `IAM`, run multiple independent instances for availability, with regular cluster resets verifying that our Infrastructure-as-Code configuration can reconstruct all cluster-level services from scratch and recover from failures without relying on accumulated manual state.

Dynamic lower-layer insertion in dockerd. We modify the open-source Docker daemon (based on the Moby project (Moby Project, 2026)) to dynamically insert EROFS-backed lower layers at container creation time. Specifically, we pass the path of a pre-mounted EROFS layer and insert it into the overlayfs stack before mounting, placing it as the topmost lower layer so it can override files in layers below. This change is minimal, requiring only 30 lines of Go code.

Rust-based OverlayBD and ublk library. We use the Rust port of OverlayBD, to which we contributed, together with our Rust userspace library for ublk to form the on-demand block-storage path for Firecracker microVMs described in §5.3. The storage layer supports 3FS, object storage services such as OSS, and container registries as remote backends. It can also use the local filesystem as a second-level cache, allowing data evicted from the page cache to be served locally without another remote fetch. These storage components have been open sourced at `https://github.com/kvcache-ai/AgentENV/tree/main/storage/overlaybd`.

Memory and CPU QoS configuration. All mechanisms leverage existing Linux kernel features. Virtio-pmem with DAX is enabled via Firecracker device configuration and guest kernel mount options. DAMON-based reclamation is activated through guest kernel parameters and sysfs tuning. For CPU QoS, we set best-effort tasks to `SCHED_IDLE` and enable core scheduling via `prctl(PR_SCHED_CORE)` to group tasks by QoS class. No kernel modifications are required; the implementation consists entirely of configuration and integration with our sandbox orchestrator.

GPU FnCall for operator benchmarking. We equip FnCall with GPUs for stateless operator benchmarking. Because GPU capacity is limited, GPU FnCall uses three mechanisms to improve concurrency while preserving performance isolation. First, NVIDIA Multi-Instance GPU (MIG) partitions each GPU into isolated *instances*, allowing benchmarks to run concurrently with exclusive access to assigned instances. Second, CPU FnCall handles compilation and passes the resulting artifacts to GPU FnCall, avoiding unnecessary GPU occupation. Finally, a warm pool of Python processes initializes the runtime and imports libraries in advance, allowing requests to begin operator execution directly. Together, these mechanisms reduce non-GPU overhead on the execution path, thereby improving both GPU utilization and benchmarking throughput. For non-performance-sensitive tasks, we also provide a shared GPU mode that increases concurrency by allowing multiple workloads to share a GPU instance.

3FS deployment. Each 3FS (An et al., 2024) storage server is equipped with 20×15 TB SSDs

and 2 × 400Gbps RDMA NICs. CPU nodes access 3FS through its FUSE-based client, with EROFS metadata stored locally and file data served on demand from 3FS. Tens of storage servers support on-demand image loading for a cluster with hundreds of thousands of CPU cores.

8. Evaluation

Our evaluation measures the effectiveness and overhead of four core mechanisms in §5: on-demand image loading, composable environment layers, memory optimization, and QoS-aware CPU scheduling. The experiments focus on these infrastructure performance mechanisms; the framework integration in §6 is outside the evaluation scope.

8.1. Experimental Setup

We conduct the experiments in this chapter on a dedicated 10-node CPU test cluster, separate from our production deployment.

Hardware. To preclude nested virtualization, our microVMs execute directly on bare-metal hardware. Each microVM node is provisioned with AMD EPYC 9655 processors across 2 sockets × 96 cores × 2 SMT threads, 1.5 TB of DRAM, and 3.4 TB of local storage. In contrast, the container-based experiments operate within a QEMU virtual machine whose configuration consists of an AMD EPYC 9655 processor, 1 socket × 96 cores × 2 SMT threads, for 192 hardware threads, 512 GB of memory, and 5.8 TB of local storage.

Kernel versions. All hosts run Linux 7.0, and all microVM guests run Linux 6.1.

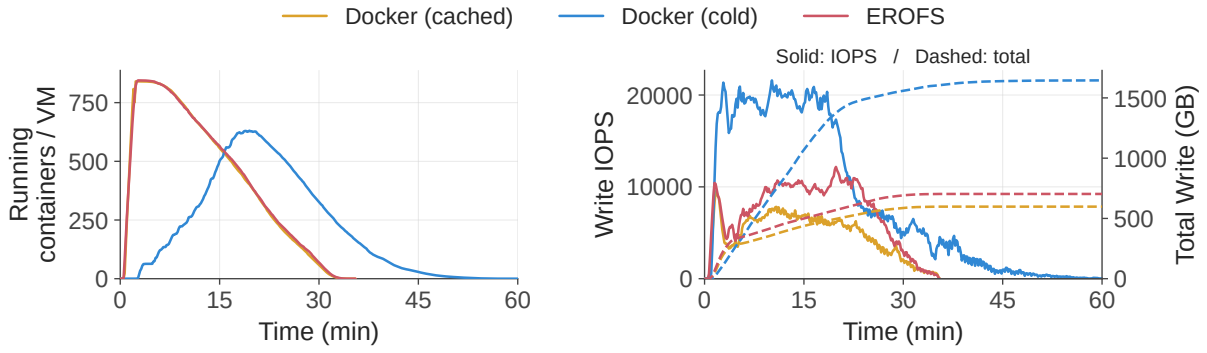
Workloads. Our workloads are drawn from real RL training and evaluation scenarios. The task suites include internal software-engineering benchmarks, SWE-bench (Jimenez et al., 2024), Terminal-Bench (Merrill et al., 2026), security exploit tasks, and similar domains.

8.2. On-Demand Image Loading

We evaluate on-demand EROFS image pulling against eager full-image pulling from a remote registry (Docker Pull (cold)) and a fully-local baseline where all image layers are pre-cached on the node (Docker Pull (cached)). The experiment issues a burst of 8,192 containers distributed across the 10-node cluster under a real RL evaluation workload that requires diverse, multi-gigabyte images at startup. We measure the number of concurrently running containers over time, instantaneous disk-write IOPS, and cumulative disk-write volume. CPU and memory utilization show negligible differences across configurations and are omitted.

As shown in Fig. 10, on-demand EROFS pulling reaches peak concurrency nearly as quickly as the fully-local baseline because image layers are mounted directly and data is fetched from 3FS as sandboxes access their working sets. Eager Docker pulling must download and extract every layer before a container can start, delaying container creation during the first 20 minutes. On-demand pulling finishes all tasks in ~35 minutes, matching the fully-local baseline, while eager pulling requires over 60 minutes, a 1.71× slowdown.

Eager pulling also reaches nearly twice the peak disk-write IOPS of the on-demand path and accumulates over 1,600 GB of disk writes per node. On-demand pulling produces only a brief initial burst and plateaus at ~700 GB, approximately 57% less than eager pulling and close to the ~600 GB fully-local baseline. These results validate the design of §5.3: fetching image data on demand for each sandbox's working set avoids full-image download and extraction while approaching fully-local performance.



(a) Running containers (b) Disk writes

Figure 10 | On-demand EROFS pulling vs. eager Docker pulling (cold) and fully-local Docker (cached) under an 8,192-container burst across the 10-node evaluation cluster. Left: running-container count per node over time. Right: instantaneous disk-write IOPS (solid, left axis) and cumulative disk writes (dashed, right axis).

8.3. Composable Image Layers: EROFS vs. Tar

For code repositories and development environments, we compare two approaches for provisioning the same evaluation workspace and toolkits, including task repositories, scaffold binaries, and command-line tools. The conventional approach packages these files as a compressed `tar.gz` archive, which is distributed and extracted into each sandbox, whereas the EROFS approach packages the same files as a compressed, read-only filesystem image that can be mounted directly as a composable layer. We replace LLM generation with a prerecorded, deterministic sequence of tool calls so that runs differ only in how their workspaces and toolkits are provisioned.

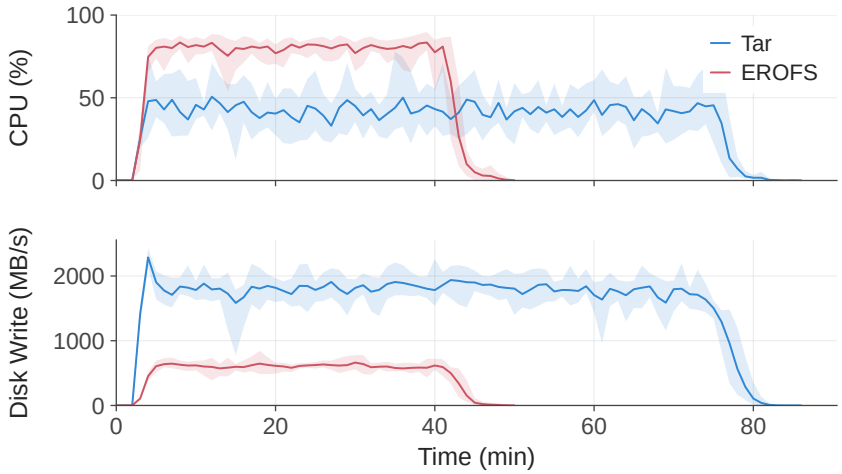


Figure 11 | Setup-phase CPU utilization and disk-write throughput when provisioning the evaluation workspace with per-sandbox tar extraction versus EROFS layer mounting.

Because `tar.gz` is a sequential stream format, every sandbox must decompress the archive and write all workspace and toolkit files into its local writable layer before tool calls can begin. This extends end-to-end task completion time to 79 minutes. EROFS mounts the shared layers directly without extraction, allowing sandboxes to enter the tool-call phase earlier and reducing completion time to 45 minutes, a 1.76× speedup. As shown in Fig. 11, tar-based provisioning generates roughly 5.5× the total disk-write traffic and 3.4× the peak disk-write throughput of the EROFS path. Peak CPU utilization is higher with EROFS because more sandboxes enter the

tool-call phase earlier and execute operations concurrently. This does not indicate higher setup overhead, since EROFS avoids the CPU work of repeatedly decompressing and unpacking the archives. These results validate the effectiveness of the composable-layer design in §5.1.

8.4. Memory under Overcommit

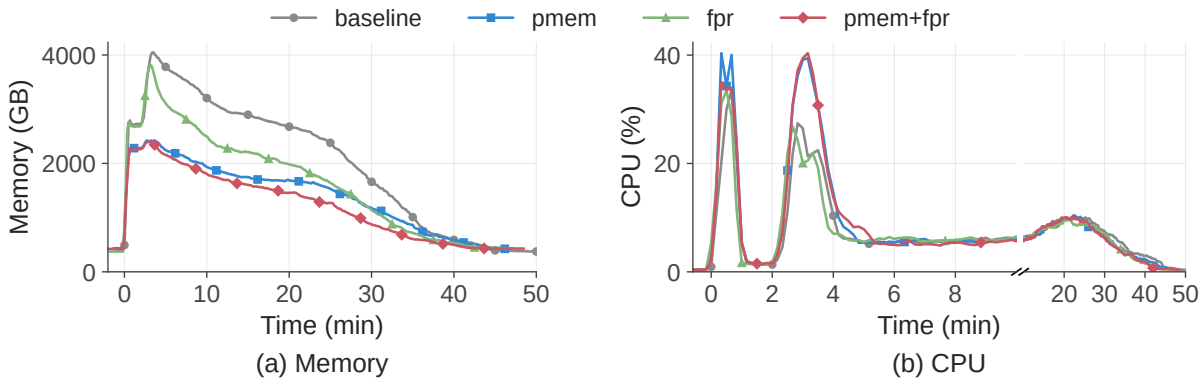


Figure 12 | Host memory usage (left) and CPU utilization (right) across the four Firecracker configurations under a real agentic RL workload. The CPU panel uses an expanded time scale for the first 10 minutes and a compressed scale for the 10–50 minute interval.

We run a real agentic-RL workload on our test cluster and compare four Firecracker configurations: the unoptimized baseline, virtio-pmem with DAX alone, DAMON-based free-page reporting (FPR) via the virtio-balloon device alone, and both mechanisms combined. Virtio-pmem with DAX collapses the redundant per-guest page caches into a single shared host mapping, reducing peak host memory usage by 40.2% compared with baseline. DAMON + balloon FPR alone leaves peak usage largely unchanged but reduces time-integrated host memory consumption by 21.2%. Combining both mechanisms produces the lowest overall memory consumption. As Fig. 12 shows, virtio-pmem raises transient peak CPU utilization from 26.5% to 41.4%. This increase may partly reflect differences in the cold-access paths. Virtio-pmem with DAX can require synchronous fault handling to establish mappings and make backing data available, while buffered virtio-blk can benefit from guest-side readahead and batched block I/O. In CPU-constrained deployments, operators may prefer to enable FPR alone and retain virtio-blk. Together, these results validate the complementary memory optimizations in §5.2: virtio-pmem reduces page-cache duplication, while DAMON with balloon FPR reclaims idle guest memory.

8.5. CPU QoS under Overcommit

We run latency-sensitive (LS) tasks from a real evaluation workload alongside co-located best-effort (BE) load ranging from 10% to 50% of node capacity, measuring how well our mechanism preserves per-sandbox performance under high-density deployment. We compare an unprotected baseline, `SCHED_IDLE` alone, and `SCHED_IDLE` combined with core scheduling.

As shown in Fig. 13, we use a latency-sensitive chess application as the test workload. At 50% BE load, its per-step latency increases by 45.2% over the no-co-location baseline without QoS controls. `SCHED_IDLE` alone improves latency by at most 3.4% because an LS thread can still contend with BE work running on its SMT sibling. Adding core scheduling keeps latency close to the no-co-location baseline at low load and limits inflation to 17.3% at 50% load. The improvement grows as BE contention increases. These results validate the two-level CPU QoS

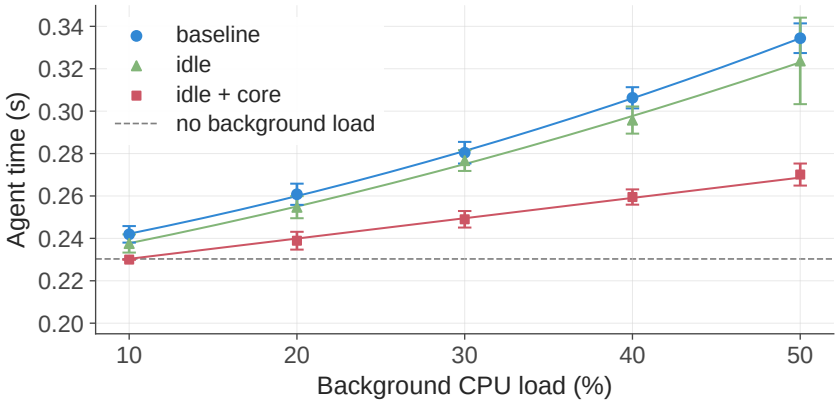


Figure 13 | Latency-sensitive agent time under increasing co-located best-effort CPU load, comparing an unprotected baseline, `SCHED_IDLE` alone, and `SCHED_IDLE` combined with core scheduling.

design in §5.2: `SCHED_IDLE` prioritizes LS tasks, while core scheduling isolates them from BE work on sibling SMT threads. The residual degradation mainly results from reduced CPU turbo frequency under high multicore load, memory bandwidth and shared last-level cache (LLC) contention that core scheduling does not address. Since this remaining interference is already tolerable, we do not apply memory bandwidth isolation.

9. Related Work

Serverless computing. Serverless platforms such as SAND (Akkus et al., 2018), REAP (Ustiugov et al., 2021), TrEnv (Huang et al., 2024), and RunD (Li et al., 2022) optimize cold-start latency and resource sharing for short-lived, stateless functions. These workloads typically reuse a limited set of images at high fanout, and many systems assume that the required images are already available locally. Agentic training instead uses long-lived, stateful sandboxes drawn from an image corpus that exceeds single-node storage and has low per-image fanout.

LLM code execution platforms. Recent systems provide sandboxed code execution for LLM workflows in both training and inference. Inference-facing systems include OpenAI Code Interpreter (OpenAI, 2025), E2B (E2B, 2024), and Kimi-K2.5's Agent Swarm (Kimi Team, 2026). Training systems such as MiMo-V2-Flash (Xiaomi LLM-Core Team, 2026) and ComputerRL (Lai et al., 2025) mention their execution environments but focus primarily on model and training design. DSec focuses on the underlying sandbox infrastructure, integrating environment composition, resource overcommit, image serving, and preemption-safe resumption within one platform.

Container image and filesystem formats. DADI (Li et al., 2020) and CoFS (Wang et al., 2026) support on-demand container-image loading, while FaaSNet (Wang et al., 2021) uses peer-to-peer delivery to accelerate image distribution. EROFS (Gao et al., 2019) provides a compressed read-only filesystem with random access. DSec builds on these techniques for RL training and evaluation, serving container and microVM images from 3FS rather than introducing a separate registry and peer-to-peer distribution tier.

Lightweight isolation. To run diverse workloads, several isolation paradigms have been proposed, including microVMs (Agache et al., 2020) or VM-backed kata-containers (Kata Containers, 2017), library OSes (che Tsai et al., 2017; LiteBox, 2025), WebAssembly runtimes (Gadepalli et al., 2020; Shillaker and Pietzuch, 2020), unikernels (Cadden et al., 2020), nested kernels (Dautenhahn

et al., 2015; Zhang et al., 2025), and nested virtualization (Huang et al., 2023). They offer different trade-offs among isolation, compatibility, and performance. Rather than proposing another isolation mechanism, DSec integrates multiple sandbox backends behind a unified platform, allowing callers to choose the appropriate backend for each task.

RL training infrastructure. Systems such as Slime (Zhu et al., 2025), veRL (Sheng et al., 2025), OpenRLHF (Hu, 2026; Hu et al., 2025), and Seer (Qin et al., 2026) focus on scaling RL training through efficient GPU scheduling, communication, and sample throughput. They treat the execution environment as a black box, assuming that sandboxes are available and correctly configured. DSec operates at the complementary infrastructure layer, managing sandbox provisioning and lifecycle while coordinating execution state and security policy with the training framework.

10. Conclusion

We presented DSec, a production sandbox platform for large-scale LLM agentic training, evaluation and environment construction. DSec exposes multiple sandbox backends through a unified interface, allowing users to select an appropriate execution environment for different functionality, compatibility, and isolation requirements. Composable EROFS-backed layers avoid repeated environment rebuilding and extraction, complementary memory sharing and reclamation mechanisms support high-density deployment, QoS-aware CPU scheduling protects latency-sensitive tasks under oversubscription, and 3FS-backed on-demand image loading reduces image distribution overhead. DSec also integrates with the RL framework for preemption-safe resumption and task-specific network policy, providing a scalable execution foundation for agentic workloads.

References

A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 419–434, Santa Clara, CA, Feb. 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/agache.

I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: Towards High-Performance serverless computing. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 923–935, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/akkus.

D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané. Concrete problems in ai safety, 2016. URL https://arxiv.org/abs/1606.06565.

W. An, X. Bi, G. Chen, S. Chen, C. Deng, H. Ding, K. Dong, Q. Du, W. Gao, K. Guan, J. Guo, Y. Guo, Z. Fu, Y. He, P. Huang, J. Li, W. Liang, X. Liu, X. Liu, Y. Liu, Y. Liu, S. Lu, X. Lu, X. Nie, T. Pei, J. Qiu, H. Qu, Z. Ren, Z. Sha, X. Su, X. Sun, Y. Tan, M. Tang, S. Wang, Y. Wang, Y. Wang, Z. Xie, Y. Xiong, Y. Xu, S. Ye, S. Yu, Y. Zha, L. Zhang, H. Zhang, M. Zhang, W. Zhang, Y. Zhang, C. Zhao, Y. Zhao, S. Zhou, S. Zhou, and Y. Zou. Fire-flyer ai-hpc: A cost-effective software-hardware co-design for deep learning. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '24. IEEE Press, 2024. ISBN 9798350352917. doi: 10.1109/SC41406.2024.00089. URL https://doi.org/10.1109/SC41406.2024.00089.

Anomaly. OpenCode, 2025. URL https://opencode.ai/.

F. Bellard. QEMU, a fast and portable dynamic translator. In 2005 USENIX Annual Technical Conference (USENIX ATC 05), Anaheim, CA, Apr. 2005. USENIX Association. URL https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo. Seuss: skip redundant paths to make serverless fast. In Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3392698. URL https://doi.org/10.1145/3342195.3392698.

C. che Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In 2017 USENIX Annual Technical Conference (USENIX ATC 17), pages 645–658, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450328357. doi: 10.1145/2694344.2694386. URL https://doi.org/10.1145/2694344.2694386.

DeepSeek-AI. Fire-flyer file sytem. URL https://github.com/deepseek-ai/3fs.

DeepSeek-AI. Deepseek-v3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

DeepSeek-AI. Deepseek-v4.1-flash: Pushing the limits of kv cache compression, 2026. URL https://arxiv.org/abs/2609.19969.

Dragonfly Community. Nydus: Dragonfly container image service, 2020. URL https://github.com/dragonflyoss/nydus.

DXVK. DXVK: A vulkan-based implementation of direct3d 8/9/10/11. https://github.com/doitsujin/dxvk, 2018.

E2B. E2B: Open-source secure sandboxes for AI code execution, 2024. URL https://github.com/e2b-dev/E2B.

P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer. Sledge: a serverless-first, light-weight wasm runtime for the edge. In Proceedings of the 21st International Middleware Conference, Middleware '20, page 265–279, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381536. doi: 10.1145/3423211.3425680. URL https://doi.org/10.1145/3423211.3425680.

X. Gao, M. Dong, X. Miao, W. Du, C. Yu, and H. Chen. EROFS: A compression-friendly read-only file system for resource-scarce devices. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 149–162, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL http://www.usenix.org/conference/atc19/presentation/gao.

D. Guo, D. Yang, H. Zhang, J. Song, P. Wang, Q. Zhu, R. Xu, R. Zhang, S. Ma, X. Bi, X. Zhang, X. Yu, Y. Wu, Z. F. Wu, Z. Gou, Z. Shao, Z. Li, Z. Gao, A. Liu, B. Xue, B. Wang, B. Wu, B. Feng, C. Lu, C. Zhao, C. Deng, C. Ruan, D. Dai, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Xu, H. Ding, H. Gao, H. Qu, H. Li, J. Guo, J. Li, J. Chen, J. Yuan, J. Tu, J. Qiu, J. Li, J. L. Cai, J. Ni, J. Liang, J. Chen, K. Dong, K. Hu, K. You, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Zhao, L. Wang, L. Zhang, L. Xu, L. Xia, M. Zhang, M. Zhang, M. Tang, M. Zhou, M. Li, M. Wang, M. Li, N. Tian, P. Huang, P. Zhang, Q. Wang, Q. Chen, Q. Du, R. Ge, R. Zhang, R. Pan, R. Wang, R. J. Chen, R. L. Jin, R. Chen, S. Lu, S. Zhou, S. Chen, S. Ye, S. Wang, S. Yu, S. Zhou, S. Pan, S. S. Li, S. Zhou, S. Wu, T. Yun, T. Pei, T. Sun, T. Wang, W. Zeng, W. Liu, W. Liang, W. Gao, W. Yu, W. Zhang, W. L. Xiao, W. An, X. Liu, X. Wang, X. Chen, X. Nie, X. Cheng, X. Liu, X. Xie, X. Liu, X. Yang, X. Li, X. Su, X. Lin, X. Q. Li, X. Jin, X. Shen, X. Chen, X. Sun, X. Wang, X. Song, X. Zhou, X. Wang, X. Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. Zhang, Y. Xu, Y. Li, Y. Zhao, Y. Sun, Y. Wang, Y. Yu, Y. Zhang, Y. Shi, Y. Xiong, Y. He, Y. Piao, Y. Wang, Y. Tan, Y. Ma, Y. Liu, Y. Guo, Y. Ou, Y. Wang, Y. Gong, Y. Zou, Y. He, Y. Xiong, Y. Luo, Y. You, Y. Liu, Y. Zhou, Y. X. Zhu, Y. Huang, Y. Li, Y. Zheng, Y. Zhu, Y. Ma, Y. Tang, Y. Zha, Y. Yan, Z. Z. Ren, Z. Ren, Z. Sha, Z. Fu, Z. Xu, Z. Xie, Z. Zhang, Z. Hao, Z. Ma, Z. Yan, Z. Wu, Z. Gu, Z. Zhu, Z. Liu, Z. Li, Z. Xie, Z. Song, Z. Pan, Z. Huang, Z. Xu, Z. Zhang, and Z. Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. Nature, 645(8081):633–638, Sep 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL https://doi.org/10.1038/s41586-025-09422-z.

J. Hu. Reinforce++: A simple and efficient approach for aligning large language models. arXiv preprint arXiv:2501.03262, 2026.

J. Hu, X. Wu, W. Shen, J. K. Liu, Z. Zhu, W. Wang, S. Jiang, H. Wang, H. Chen, B. Chen, W. Fang, Xianyu, Y. Cao, H. Xu, and Y. Liu. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework, 2025. URL https://arxiv.org/abs/2405.11143.

H. Huang, J. Lai, J. Rao, H. Lu, W. Hou, H. Su, Q. Xu, J. Zhong, J. Zeng, X. Wang, Z. He, W. Han, J. Liu, T. Ma, and S. Wu. Pvm: Efficient shadow paging for deploying secure containers in cloud-native environment. In Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23, page 515–530, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613158. URL https://doi.org/10.1145/3600006.3613158.

J. Huang, M. Zhang, T. Ma, Z. Liu, S. Lin, K. Chen, J. Jiang, X. Liao, Y. Shan, N. Zhang, M. Lu, T. Ma, H. Gong, and Y. Wu. Trenv: Transparently share serverless execution environments across different functions and nodes. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP '24, page 421–437, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712517. doi: 10.1145/3694715.3695967. URL https://doi.org/10.1145/3694715.3695967.

C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Kata Containers. Kata Containers: Secure Containers with Lightweight Virtual Machines. https://katacontainers.io/, 2017.

Kimi Team. Kimi K2.5: Visual agentic intelligence, 2026.

H. Lai, X. Liu, Y. Zhao, H. Xu, H. Zhang, B. Jing, Y. Ren, S. Yao, Y. Dong, and J. Tang. ComputerRL: Scaling end-to-end online reinforcement learning for computer use agents, 2025.

H. Li, Y. Yuan, R. Du, K. Ma, L. Liu, and W. Hsu. DADI: Block-Level image service for agile and elastic application deployment. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 727–740. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/li-huiba.

Z. Li, J. Cheng, Q. Chen, E. Guan, Z. Bian, Y. Tao, B. Zha, Q. Wang, W. Han, and M. Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In 2022 USENIX Annual Technical Conference (USENIX ATC 22), pages 53–68, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-29-27. URL https://www.usenix.org/conference/atc22/presentation/li-zijun-rund.

LiteBox. Litebox: A security-focused library os supporting kernel- and user-mode execution. https://github.com/microsoft/litebox, 2025.

M. A. Merrill, A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. Y. Shin, T. Walshe, E. K. Buchanan, J. Shen, G. Ye, H. Lin, J. Poulos, M. Wang, M. Nezhurina, J. Jitsev, D. Lu, O. M. Mastromichalakis, Z. Xu, Z. Chen, Y. Liu, R. Zhang, L. L. Chen, A. Kashyap, J.-L. Uslu, J. Li, J. Wu, M. Yan, S. Bian, V. Sharma, K. Sun, S. Dillmann, A. Anand, A. Lanpouthakoun, B. Koopah, C. Hu, E. Guha, G. H. S. Dreiman, J. Zhu, K. Krauth, L. Zhong, N. Muennighoff, R. Amanfu, S. Tan, S. Pimpalgaonkar, T. Aggarwal, X. Lin, X. Lan, X. Zhao, Y. Liang, Y. Wang, Z. Wang, C. Zhou, D. Heineman, H. Liu, H. Trivedi, J. Yang, J. Lin, M. Shetty, M. Yang, N. Omi, N. Raoof, S. Li, T. Y. Zhuo, W. Lin, Y. Dai, Y. Wang, W. Chai, S. Zhou, D. Wahdany, Z. She, J. Hu, Z. Dong, Y. Zhu, S. Cui, A. Saiyed, A. Kolbeinsson, J. Hu, C. M. Rytting, R. Marten, Y. Wang, A. Dimakis, A. Konwinski, and L. Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

M. Mitzenmacher. The power of two choices in randomized load balancing. IEEE Transactions on Parallel and Distributed Systems, 12(10):1094–1104, 2001. doi: 10.1109/71.963420.

Moby Project. The Moby Project, 2026. URL https://github.com/moby/moby.

Open Container Initiative. Open Container Initiative Image Format Specification, 2026. URL https://github.com/opencontainers/image-spec.

OpenAI. Code interpreter: Agent harness and sandbox for code execution, 2025. URL https://platform.openai.com/docs/guides/code-interpreter.

OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey,

W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Ł. Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, J. H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Ł. Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. H. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. B. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe. Training language models to follow instructions with human feedback. In Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

S. Park, J. Ahn, H. Y. Kim, and Y. Lee. Profiling dynamic data access patterns with controlled overhead and quality. In Proceedings of the 20th International Middleware Conference Industrial Track (Middleware '19), pages 29–30, 2019. doi: 10.1145/3366626.3368125.

QEMU Project. VirtIO Persistent Memory, 2026. URL https://www.qemu.org/docs/master/system/devices/virtio/virtio-pmem.html.

R. Qin, W. He, W. Huang, Y. Zhang, Y. Zhao, B. Pang, X. Xu, Y. Shan, Y. Wu, and M. Zhang. Seer: Online context learning for fast synchronous llm reinforcement learning, 2026. URL https://arxiv.org/abs/2511.14617.

G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu. Hybridflow: A flexible and efficient rlhf framework. In Proceedings of the Twentieth European Conference on Computer Systems, EuroSys '25, page 1279–1297, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3696075. URL https://doi.org/10.1145/3689031.3696075.

Y. Shi, W. Zhang, and T. Cui. A programming paradigm for spatiotemporal composability, 2026. URL https://arxiv.org/abs/2608.25512.

S. Shillaker and P. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 419–433. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/shillaker.

J. Skalse, N. H. R. Howe, D. Krasheninnikov, and D. Krueger. Defining and characterizing reward hacking. In Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446714. URL https://doi.org/10.1145/3445814.3446714.

C. A. Waldspurger. Memory resource management in VMware ESX server. In 5th Symposium on Operating Systems Design and Implementation (OSDI 02), Boston, MA, Dec. 2002. USENIX Association. URL https://www.usenix.org/conference/osdi-02/memory-resource-management-vmware-esx-server.

A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 443–457. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/wang-ao.

L. Wang, J. Du, Y. Yang, Q. Wu, T. Liu, and H. Wu. CoFS: A filesystem for fast container startup. In 24th USENIX Conference on File and Storage Technologies (FAST 26), pages 415–423, Santa Clara, CA, Feb. 2026. USENIX Association. ISBN 978-1-939133-53-3. URL https://www.usenix.org/conference/fast26/presentation/wang-li.

Xiaomi LLM-Core Team. MiMo-V2-Flash technical report, 2026.

T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, T. J. Hua, Z. Cheng, D. Shin, F. Lei, Y. Liu, Y. Xu, S. Zhou, S. Savarese, C. Xiong, V. Zhong, and T. Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In Advances in Neural Information Processing Systems, 2024. doi: 10.52202/079017-1650.

C. Zhang, R. Priolkar, Y. Jiang, Y. Xiao, M. Vij, Z. Liang, and A. Ahmad. Erebor: A drop-in sandbox solution for private data processing in untrusted confidential virtual machines. In Proceedings of the Twentieth European Conference on Computer Systems, EuroSys '25, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400711961. doi: 10.1145/3689031.3717464. URL https://doi.org/10.1145/3689031.3717464.

S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, T. Ou, Y. Bisk, D. Fried, U. Alon, and G. Neubig. Webarena: A realistic web environment for building autonomous agents. In International Conference on Learning Representations, 2024. URL https://openreview.net/forum?id=oKn9c6ytLx.

Z. Zhu, C. Xie, X. Lv, and slime Contributors. slime: An llm post-training framework for rl scaling. https://github.com/THUDM/slime, 2025.

P. Zijlstra, J. Fernandes, and V. Pillai. Core scheduling, 2021. URL https://docs.kernel.org/admin-guide/hw-vuln/core-scheduling.html.