

MCon: Building an Elastic Cloud for Android Applications with Framework Consolidation

Jiaxing Qiu¹, Zijie Zhou¹, Zhenhua Li^{1✉}, Hongyi Wang¹, Zizhang Liu¹, Yunhao Liu¹
Yumeng Liang², Jie Wu^{2,3}, Wen Xia⁴, Tianyin Xu⁵

¹Tsinghua University ²China Telecom Cloud Computing Research Institute ³Temple University

⁴Harbin Institute of Technology, Shenzhen ⁵University of Illinois Urbana-Champaign

Abstract

Cloud execution of mobile apps—running unmodified apps inside full mobile OSes on commodity servers—is gaining adoption for uses such as GUI agent training and large-scale app testing. These workloads are both temporally dynamic (from second-scale agent interactions to hour-long tests) and highly bursty (up to hundreds of concurrent jobs), demanding an elastic platform that can provision mobile environments both quickly and at high density. Existing stacks fall short because each tenant has a separate copy of the mobile framework (a large collection of shared libraries and system services), which is slow to start and heavy to keep.

We present MCon, the first container system with the *framework consolidation* architecture: the framework is run as a shared, multi-tenant service instead of a per-tenant component. The core challenge is to practically enable per-tenant isolation on a large framework with no native multi-tenant support. MCon achieves this from *outside* the framework: we virtualize the IPC control path through which apps access shared framework services, and combine it with per-tenant virtual device slices to construct private views of framework and device resources. Compared to the best existing stacks, MCon achieves sub-second cold tenant allocation (15× as fast), with 70%–82% less host memory and 1%–33% less GPU memory usage. MCon has been adopted by Infinigence AI.

CCS Concepts: • Human-centered computing → Mobile computing; • Computing methodologies → Mobile agents; • Software and its engineering → Virtual machines.

Keywords: mobile computing, container, elasticity, GUI agent, inter-process communication, namespace, virtualization

ACM Reference Format:

Jiaxing Qiu, Zijie Zhou, Zhenhua Li, Hongyi Wang, Zizhang Liu, Yunhao Liu, Yumeng Liang, Jie Wu, Wen Xia, Tianyin Xu. 2026. MCon: Building an Elastic Cloud for Android Applications with

Framework Consolidation. In *Symposium on Operating Systems Principles (SOSP '26)*, September 29–October 2, 2026, Prague, Czech Republic. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3830418.3843866>

1 Introduction

Advances in autonomous mobile GUI agents [59, 82, 89] are driving demand for cloud execution of mobile apps, where unmodified apps run inside mobile OSes in containers or VMs on commodity servers. A dominant use is the post-training of GUI agents, *e.g.*, reinforcement learning (RL) that scales through massive parallel *rollouts*: each rollout boots a fresh mobile instance, launches an app, lets the agent attempt a short sequence of GUI actions, records the outcome, and resets [12, 80]. Similar patterns arise in large-scale automated app testing and CI [30, 41, 57].

Rather than long-running virtual phones, these mobile instances are increasingly treated as ephemeral compute environments—they are highly dynamic in time (from second-scale interactions to hour-long tests) and bursty in quantity (*e.g.*, hundreds of rollouts arriving concurrently). The underlying platform must therefore be *elastic* along two axes: provisioning environments at *low latency*, and sustaining them at *high density* to keep parallel jobs cost-effective.

However, today’s mobile stacks fall short on both axes. On the latency axis, provisioning is slow: on state-of-the-art stacks [14, 37, 73, 74], booting 32 instances on a commodity server takes over five minutes, inflating end-to-end latency for short, second-scale jobs by orders of magnitude [28]. On the density axis, capacity is low: the best stacks saturate at roughly 32 instances per server before exhausting host memory (§7.4), far short of the hundreds of rollouts typically needed, so excess jobs queue and per-job cost climbs.

Both shortfalls trace to the same root cause: the structure of mobile systems. Besides the kernel, an app cannot run until the *mobile framework*—a large userspace stack of shared libraries and privileged services [76, 88]—has initialized. This framework is both slow to start and resource-hungry: under typical workloads of top-50 popular apps, it takes tens of seconds to initialize and consumes 80% of memory and 35% of CPU cycles per instance on average.

Yet existing systems merely port server virtualization techniques to mobile stacks: they either launch one VM per instance [37, 73], or treat the framework as a black



This work is licensed under a Creative Commons Attribution 4.0 International License.

SOSP '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2585-2/26/09

<https://doi.org/10.1145/3830418.3843866>

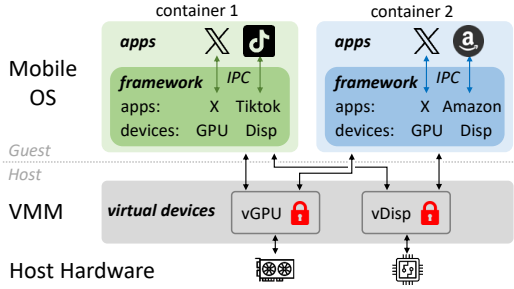


Figure 1. Architecture of container-based mobile stacks.

box and pack many containerized instances into a single long-running VM [14, 74] (Figure 1; a VM is necessary for hardware-accelerated graphics because server GPUs lack drivers for mobile OSes [29, 78]). Either way, the framework is duplicated per instance, which explains both shortfalls at once: its initialization dominates provisioning latency, while its large footprint caps density and forces jobs to queue.

The framework bottleneck also makes elasticity techniques from serverless and VM platforms hard to transfer. MicroVMs such as Firecracker [1] minimize VMM and guest-kernel boot, but in mobile stacks these already boot in under two seconds; the tens of seconds that dominate provisioning are spent in the framework, *above* the layer microVMs optimize. Snapshot-, fork-, and pre-warming techniques [23, 70, 75] do reach this bottleneck and can cut framework initialization latency, but only for a *fixed* device profile: robust agents and thorough tests need to generalize across hundreds of distinct profiles (e.g., display size, DPI, and refresh rate) that reshape app layout [46, 82], so a warm pool or resident snapshot per profile fragments capacity [53] (§2.1). Moreover, none of these techniques improve *density*: each instance still carries a full, duplicated framework in memory.

We present MCon, the first elastic container system natively designed for mobile systems, based on the *framework consolidation* architecture. As shown in Figure 2, instead of treating the mobile framework as a standalone component per tenant, we run the framework as a shared multi-tenant service: apps from multiple tenants are consolidated onto a single framework, while still allowing per-tenant control over hardware profiles. Our architecture improves elasticity in two ways: (1) without cold-booting and duplicating full mobile frameworks per tenant, container provisioning is much faster; (2) containers also have a smaller resource footprint and can run at higher density.

To realize shared frameworks, two major challenges exist:

- **State isolation:** The mobile framework is designed for a single trusted device and offers little isolation of app state. It assumes a global namespace (e.g., only one copy of any app package is allowed in the system) and loose visibility (e.g., any app can see the screen of other apps).

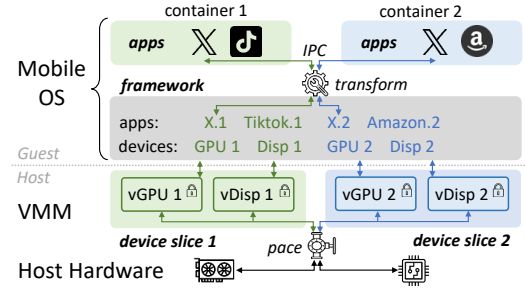


Figure 2. Architecture of MCon.

- **Performance interference:** As MCon enables tenants to run at high densities, high-throughput devices receive commands from many more tenants than they are designed for, causing bottlenecks in their global synchronization mechanisms like mutexes, as shown in Figure 1.

At the core, the challenges stem from a fundamental tension between consolidation and isolation in a complex system stack. Consolidation (reusing the framework) reduces resource overhead, but introduces interference and security issues in that shared layer. Worse, once the framework is consolidated, per-tenant semantics are erased in the underlying layers (the device stack), pushing those layers toward further consolidation and causing interference to propagate downward. While a natural solution is to refactor the framework to be truly multi-tenant, doing so is intractable in practice: it requires changing system APIs to be multi-tenant aware, which would break compatibility with existing apps.

To reconcile the tension, our core insight is that framework isolation does not require changing the framework itself; instead, it can be achieved *externally* by virtualizing the interface through which apps perceive the system. In mobile OSes, apps run in sandboxes and interact with the framework via a stable IPC channel named Android Binder [33]. This microkernel-style architecture creates a key opportunity: by transforming traffic at the IPC boundary, we can achieve per-tenant isolation *before* app requests reach the framework. This external virtualization not only allows for practical consolidation, but also brings tenant identity to the underlying device stack, enabling performance isolation.

As shown in Figure 2, MCon adopts a hybrid architecture driven by the distinct characteristics of each layer in the mobile stack. For the framework, which is slow to initialize and resource-heavy, we achieve effective consolidation with framework namespaces (§4): each tenant on the shared framework is provided with a private, virtualized view of framework and device resources. Conversely, for the para-virtualized device layer, which is lightweight but performance-critical, we prioritize isolation via slicing (§5): each tenant gets its own set of virtual devices to restrict performance interference and failure propagation.

Finally, MCon has a security-conscious design. MCon targets single-provider deployments (e.g., internal training and

CI/testing clusters), where tenants may be buggy, curious, or noisy, but not malicious. The framework, kernel, and host are trusted; framework compromise can affect all co-located tenants. Thus, MCon does not target mutually distrustful public workloads. Within this threat model, MCon enhances tenant protection with UID-based kernel controls and default-deny IPC handling, while limiting noisy-neighbor interference using resource quotas and frame pacing.

We integrate the above designs and implement MCon on Android¹ and vSoC [73], a mobile emulator with efficient virtual devices. We evaluate MCon against representative mobile containers (Redroid [74], Anbox Cloud [14]) and emulators (Google Android Emulator [37], vSoC [73]) with microbenchmarks and an end-to-end benchmark on top-50 popular apps. Compared to the best stacks, MCon delivers sub-second cold tenant allocation (15× as fast), and reduces per-tenant host memory use by 70%–82% and GPU memory use by 1%–33%, with a small 4% latency overhead for IPC. MCon has been adopted by Infinigence AI for training mobile agents, accelerating end-to-end RL training speed by 56%, and reducing mobile infrastructure cost by 60%.

This work makes the following contributions:

- We propose an elastic mobile container architecture that allows multiple tenants to reuse one mobile framework while recovering per-tenant control and isolation.
- We design and implement MCon, which addresses the challenges of the architecture by isolating per-tenant resources from outside the framework itself.
- We demonstrate MCon’s elasticity, achieving sub-second tenant provisioning and significantly improving container density compared to state-of-the-art mobile stacks.

MCon is open source at <https://github.com/VirtualSoC/mcon-docs>. This work does not raise any ethical concerns.

2 Background and Motivation

To handle heterogeneous hardware and strict energy budgets, mobile systems differ fundamentally from server systems. They rely on a *mobile framework*—a large userspace stack of shared libraries and system services that mediates interactions between apps and the kernel. However, its heavyweight design and single-user assumptions hinder efficient cloud deployment. In this section, we analyze why existing mobile solutions fall short on elasticity (§2.1), explore the opportunities and challenges of framework consolidation (§2.2), and present our key idea: external framework isolation (§2.3).

2.1 Towards Framework Consolidation

Emerging workloads require elastic mobile clouds. Training mobile GUI agents with reinforcement learning [59, 82, 89]

¹Though iOS is closed-source and has a different ecosystem, it adopts a similar architecture with a large userspace framework and IPC-based app-framework interactions [10], so our design is applicable in principle.

relies on massive parallel *rollouts*, each launching a fresh instance for a short agent interaction; because complex app GUIs limit per-rollout coverage, useful coverage needs hundreds of concurrent instances [12, 80]. Automated testing [30, 41, 57] similarly needs many isolated instances per test, and cloud gaming [17, 56, 66, 85] scales container pools for bursty, variable-length player sessions. These workloads demand elasticity along two axes: instances must be provisioned at *low latency* for short, bursty jobs, and sustained at *high density* to keep large fleets cost-effective.

Adapting current mobile stacks to these workloads is challenging. State-of-the-art containers (e.g., Redroid [74]) encapsulate the mobile userspace inside a Linux container running atop a long-running VM with a para-virtualized GPU. VM and para-virtualization are indispensable because server GPUs (particularly NVIDIA) lack native drivers for mobile OSes, and open-source alternatives (e.g., Nouveau [65]) are too unstable for production use [22].

Consequently, each tenant pays the full cost of booting and hosting the framework. In Android 13, it spans over 60 long-running processes hosting 242 system services [35]. Initializing this stack—probing hardware, loading libraries, and building service states [88]—costs seconds of CPU time per instance and inflates provisioning latency, while keeping it resident consumes gigabytes of memory and caps density.

Worse, the usual remedy also fails for mobile workloads. Pre-warming and snapshotting [23, 70] reuse an *initialized template* and thus assume a fixed environment, but mobile workloads demand diverse device profiles (e.g., screen sizes, DPI, and refresh rates) that existing stacks fix at boot time. Each profile therefore needs its own warm pool or resident snapshot set (an Android 13 snapshot averages 3.2 GiB), fragmenting capacity across profiles and inflating idle cost [50, 52]. Moreover, these techniques still instantiate a framework per instance and thus cannot raise density.

2.2 Opportunities and Challenges

These limits motivate a new architecture, *framework consolidation*: the mobile framework is run as a persistent service shared among multiple tenants, while each tenant retains its own apps, configurations, and device profiles. Consolidation offers two immediate elasticity benefits. First, provisioning is much faster: adding a tenant now only creates a tenant context rather than booting a full framework. Second, resource efficiency improves because framework costs are amortized across tenants. However, framework consolidation also introduces fundamental challenges.

Functionality. Mobile frameworks are not designed for cloud multi-tenancy. Android and OpenHarmony do provide

a basic isolation mechanism, multi-user², intended for sharing a single device among mutually trusted people (e.g., a family tablet, or passengers in a smart car) [19, 36]. However, with multi-user, critical system services still assume a global namespace. For instance, apps are labeled using globally unique strings termed package names (e.g., `com.example.app`), so only one installation of a given package can exist at a time. As a result, two tenants cannot install or run different versions of the same app, making use cases such as A/B testing impossible. Similar uniqueness applies to other global identifiers (e.g., Android ContentProvider authorities).

Moreover, because multi-user assumes mutually trusted co-users, device-wide state such as screens and process lists stays observable across users, and all users must share a single device profile (screen size, DPI, refresh rate) and device set, which is unsuitable for cloud multi-tenancy.

Performance. Framework consolidation has mixed performance implications: it improves provisioning but risks introducing noisy-neighbor effects. High-throughput devices (especially GPU and display) now multiplex command streams from many tenants, creating performance collapse and jitter in virtual devices and host drivers (§5.1). Unfortunately, with a shared framework, the underlying device stack loses per-tenant context, making per-tenant command streams impossible to separate. Even inside the framework itself, tenants can contend for framework resources via heavy APIs (e.g., enumerating sensors), degrading per-tenant performance.

Security. With framework consolidation, the trusted computing base (TCB) includes the mobile framework, enlarging the attack surface compared with kernel-only sharing. This is a tradeoff between elasticity and security we consciously make. That said, we must still prevent cross-tenant leaks, state corruption, and resource interference. For instance, framework APIs expose device-wide capabilities (e.g., screen capture, process list) that assume a single user; if not carefully controlled, apps may interfere with each other and lead to unexpected behavior.

Reliability. Consolidation creates a larger failure domain: a crash in the shared framework or underlying device stack instantly disrupts all tenants on it. If the framework crashes, apps from all tenants must stop until the framework restarts (since the framework holds stateful app contexts). Worse, virtual device failures can impact or even crash the entire VM, causing even greater disruption. A well-designed system should therefore carefully balance performance and reliability, and mitigate cascading failures across layers.

2.3 Insight: External Isolation

A natural idea to reconcile consolidation with per-tenant isolation is to add multi-tenancy to the mobile framework,

²Higher-level features such as Work Profiles [40] and Private Space [4] are also specialized user profiles built on the same multi-user stack, so they inherit its assumptions and limitations.

just like multi-user. However, this is intractable in practice: it requires invasive changes to framework APIs, breaks compatibility with existing apps, and creates a maintenance nightmare given the framework’s huge, evolving codebase (the Android framework has 7.5M lines of code as of 2025 [35]).

Instead, we tap into the unique architecture of mobile systems. Unlike server programs that access most system functionality via syscalls, mobile apps run in sandboxes [3] to protect user data. For example, every Android app executes in a dedicated process with a unique Linux UID, isolated from both the system and other apps [11, 26]. To cross the sandbox for privileged operations (e.g., accessing storage or cameras), they must interact with the framework. For instance, when an Android app needs to read a photo in user storage, it cannot invoke `open()` on the file path without permission; it must request access via the framework’s `MediaStore` API [6, 54]. The framework validates permissions and returns a file descriptor to the photo.

Crucially, since apps and system services run in separate userspace processes, these interactions cross a stable IPC boundary (e.g., Android Binder [33] or iOS XPC [10]). This IPC boundary yields a key insight: by transforming IPC data, we can enforce isolation *externally* by virtualizing the interface through which apps perceive the system.

As an example, consider two tenants that install different versions of an app with package name `X`. Android’s global package namespace would normally make the installations collide. When tenant *i* invokes a package API, MCon rewrites `X` to `X.i` before the request reaches the framework, and maps results back to `X` on return. The shared framework therefore sees distinct packages, while each tenant sees the ordinary single-device interface it expects. Neither the app nor the framework needs modification.

3 System Overview

MCon has these design goals: (i) per-tenant isolation: each tenant should be isolated from others in terms of functionality, performance, and security; (ii) fast provisioning: new tenants should be provisioned without booting a full framework; (iii) high density: the system should support many tenants per host; (iv) maintainability: the system should be easy to patch in case of OS upgrades or security issues.

As shown in Figure 2, MCon adopts a hybrid architecture tailored to each layer. For the mobile framework, which is heavy and slow to initialize, we prioritize efficiency via *consolidation*: multiple tenants share a single framework instance. The mobile framework and kernel run in a VM managed by the QEMU/KVM hypervisor. Conversely, for the para-virtualized device layer, which is lightweight but performance-critical, we prioritize isolation via *slicing*: each tenant gets dedicated device instances to eliminate contention. At the bottom, the hypervisor and virtual device slices run on a KVM-enabled Linux host. This architecture is not Linux-specific: MCon can also support microkernel-based systems [15]

Interface/Resource	Examples	MCon Mechanisms	Security Property
Framework IPC	AIDL calls, providers, broadcasts, accessibility, IME, clipboard, package visibility	Typed IPC proxy (§4.1); namespace operations, policies and default-deny (§4.2)	Confidentiality, integrity
Framework resource	CPU, memory, and expensive framework APIs	cgroups and IPC rate control (§4.3)	Availability
Direct kernel path	File syscalls, Unix sockets, procfs, sysfs	UID-based kernel isolation (§4.2)	Confidentiality, integrity
Device data plane	OpenGL ES, network I/O, media, DMA-BUF	Namespace routing (§4.2) and device slices (§5.2)	Confidentiality, availability
Device resource	GPU time, VRAM, network bandwidth	Frame pacing and GPU quota (§5.3)	Availability
Vendor API	Vendor services, private protocols	IPC proxy (§4.1) or separate vendor integration	Implementation specific

Table 1. Cross-tenant interfaces and resources, with their corresponding isolation mechanisms and security properties.

that provide cgroup-like CPU and memory controls, equivalent per-user protection, and virtual device interfaces.

To realize this architecture, MCon uses two techniques: (i) *framework namespaces* (§4), which virtualize the app-framework interface to enforce logical isolation of framework resources; and (ii) *virtual device slicing* (§5), which partitions high-throughput virtual devices into per-tenant slices to restrict performance interference and failure propagation.

Threat Model. MCon targets single-provider, semi-trusted cloud environments, such as agent training clusters, automated testing farms, and enterprise CI/CD pipelines. These deployments prioritize elasticity and density; to provide them, MCon trusts the host OS, hypervisor, mobile kernel, shared framework, and framework namespaces, which together form its trusted computing base (TCB). In particular, a compromised framework affects every tenant using that framework instance, so MCon is not intended for public clouds that execute mutually distrustful workloads.

The in-scope adversary is a curious, buggy, or noisy tenant that runs arbitrary unprivileged mobile apps. A curious tenant may attempt to read another tenant’s files, monitor their screen or clipboard, query their package list, or change system settings. A buggy tenant may send malformed data via IPC. A noisy tenant may consume excessive IPC or hardware resources and degrade others’ performance. We exclude attacks that compromise the TCB (e.g., privilege-escalation or code-execution exploits) as well as malicious host administrators and hardware side channels.

Within this model, MCon provides security along three dimensions. First, *confidentiality* prevents a tenant from observing another tenant’s private framework state, such as files, screens, clipboards, and package lists. Second, *integrity* prevents a tenant from affecting another tenant’s state, like causing package conflicts or modifying system-wide configuration. Third, *availability isolation* limits performance interference through shared CPU, memory, framework IPC, and device resources. MCon enforces these properties through a combination of framework namespaces (§4), kernel sandboxing, resource controls, and device slicing (§5).

Interface Coverage. Table 1 summarizes the interfaces and resources MCon mediates. Framework IPC is the control path to all shared and privileged resources and can be fully mediated by framework namespaces. However, high-throughput paths (e.g., graphics, network) often interact directly with

kernel drivers for performance; while their control planes remain managed by the framework [26, 72] and thus by IPC, MCon also extends isolation to the device stack to prevent noisy-neighbor effects. Some operations like accessing temporary files use kernel syscalls directly; MCon protects them by reusing Android’s built-in UID-based isolation.

Finally, some vendor systems may include custom APIs. Standard vendor services that follow modern practices like Android’s Project Treble [48] also use framework IPC and can be ported to MCon by filling out the IPC transformation policies for their APIs (§4.2). For the minority of services that hold per-tenant context and use private protocols, vendors should implement multi-tenancy on their own.

4 Framework Namespaces

Framework namespaces logically isolate tenants on a shared framework by transforming IPC messages to construct private views of system resources (e.g., package names, settings, device IDs). The framework still operates as if on a single-user device, unaware of the abstraction. The abstraction exposes a minimal control API: (i) `ns.new()` creates a namespace and returns a handle; (ii) `ns.bind(slice)` associates the namespace with a virtual device slice; and (iii) `ns.destroy(handle)` tears down the referenced namespace. The MCon container manager uses these APIs to manage tenant lifecycles. We detail their use in §5.2.

We next describe the design principles behind framework namespaces, including how IPC messages are captured (§4.1), transformed (§4.2), and regulated (§4.3).

4.1 IPC Proxy

The IPC proxy intercepts and transforms IPC traffic. Instead of modifying complex system-service internals or manipulating binary IPC data [27] whose protocols are unstable across OS versions, MCon leverages a feature of mobile IPC: as mobile systems are often written in multiple languages (Java, Swift, C++, etc.), IPC interfaces are defined in generic Interface Definition Languages (IDLs, like AIDL [31] for Android and MIG [9] for iOS). Therefore, MCon injects the proxy into the IDL compiler, intercepting IPC calls without changing service code (details in §6).

Crucially, we place the proxy logic in the framework processes rather than the app processes. If the proxy were located in the app’s address space, a malicious tenant could bypass it by manually crafting raw IPC packets.

Category	Tenant-Scoped Objects	Operation
Names	Package Names, Providers, URIs	<u>Virtualize</u> : Append tenant ID suffix (e.g., app → app.1) to prevent collisions.
Visibility	Processes, Tasks, Services, Debug APIs	<u>Filter</u> : Prune lists to only show objects belonging to the requesting tenant.
Events	Broadcasts, Notifications, Intents	<u>Filter</u> : Restrict event delivery to the originating tenant's namespace.
Data Sharing	Clipboard, Drag-and-Drop	<u>Virtualize</u> : Maintain separate data buffers for each tenant.
Identifiers	IMEI, Serial, MAC Address	<u>Virtualize</u> : Return unique, consistent generated IDs per tenant.
Configuration	Settings, Properties, Accessibility	<u>Virtualize/Reject</u> : Maintain per-tenant view or reject global changes.
Device I/O	Input Events, Device Handles	<u>Route</u> : Direct I/O streams to the specific virtual device slice.

Table 2. Policy categories in MCon.

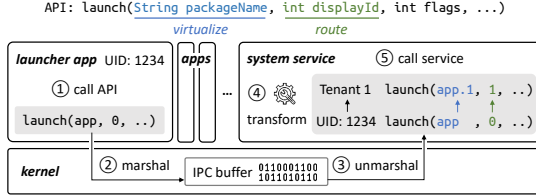


Figure 3. Data flow of an API that launches apps. Framework namespaces take effect in ④.

4.2 Operations and Policies

With the IPC proxy in place, we focus on managing transformation rules for the vast, evolving mobile APIs. A static hardcoded rule set would be unmaintainable as APIs evolve. To address this, MCon decouples isolation *mechanism* from *policy*. We distill isolation logic into a small set of generic operations (mechanism), and apply them to a bounded set of relevant framework objects (policy).

Mechanism: Generic Operations. Despite the diversity of framework APIs, the necessary transformations fall into four generic operations on common framework objects:

- Virtualize: Change names to create private resource views;
- Filter: Change visibility to hide other tenants' resources;
- Route: Direct traffic to the correct device slice;
- Reject: Deny access to global or privileged settings³.

Figure 3 illustrates the workflow of a simplified framework API that launches apps, with two policies, each acting on one framework object (virtualize on `packageName`, and route on `displayId`). As a result, when a launcher app from tenant 1 (identified by its UID) requests to launch app on display 0 (the default display), the IPC call is transformed to launch app.1 on the display assigned to tenant 1 (display 1 in our example). Virtualize ensures that app.1 is always private to tenant 1, and that the framework never confuses it with apps from other tenants, while route ensures that the app's UI always appears on the correct display slice.

Policy: Object-Based Analysis. Policies determine how operations apply to each framework API. While the large API surface seems daunting, it can be transformed in an

automated and principled way, using object-based analysis (instead of per-API). Our observation is that cross-tenant interference does not arise from arbitrary APIs, but from a bounded set of *tenant-scoped* framework objects. An object is tenant-scoped if a standalone device maintains a single, system-wide instance of it (e.g., package name, process list, clipboard) that implicitly serves the device's sole user. When several tenants share one framework, this single instance would otherwise be shared across all of them and cause cross-tenant collisions. Such objects therefore require per-tenant treatment to preserve normal device semantics. Table 2 summarizes the categories of these objects.

Accordingly, MCon defines a policy as a (tenant-scoped object → operation) mapping. The policy engine traverses IPC arguments and return values at run time, and applies the corresponding operation whenever it encounters a matching object. While most arguments are typed objects (e.g., `PackageInfo`) and hence the mapping is straightforward, a small subset of arguments are primitive-typed (e.g., `String` or `int`) and need effort to be bound to the correct object type. The engine uses stable naming conventions as a heuristic, since these names are part of the stable public AIDL interface and usually do not change (see §6). For composite objects, the engine introduces a recurse meta-operation that indicates the object should be inspected recursively. To avoid repeated runtime recursions, MCon pre-computes a *policy tree* for each object type at compile time, so the proxy can directly locate the nested fields that require transformation.

Coverage and Completeness. By design, MCon classifies the complete set of framework APIs at build time: *transform* if it carries tenant-scoped objects, and *pass-through* otherwise. Nevertheless, the correctness of the classification depends on (1) identifying all tenant-scoped objects and (2) mapping API arguments to the correct object types. We empirically validate both through expert review and automated testing. Expert review involves two steps: first, we audit which framework objects represent tenant-scoped state. Android 13 exposes 878 framework object types, of which 79 are tenant-scoped; the rest represent global or immutable state (e.g., OS version). Second, we check if the naming heuristics for primitive arguments are correct. The naming heuristics initially produced 6 false positives and 92 false negatives; these are identified and corrected with 98 explicit policies.

³Reject effectively *deprivileges* tenants relative to standalone device owners. Similar to how Linux containers can use but not modify kernel functionality, a MCon tenant cannot change global framework settings like Wi-Fi (they are always-on). This deprivileging does not break normal app semantics: third-party apps cannot programmatically modify these settings anyway [7].

We complement the static review with compatibility and adversarial testing. For compatibility, we use Android’s Compatibility Test Suite (CTS) [38] that ensures MCon preserves standard Android behavior. Separately, we develop an adversarial test suite that combines hand-written boundary cases with systematic fuzzing across the transformed API surface (§7.7). We compare results with the unmodified stack and iteratively adjust policies until $\Delta_{\text{result}}(\text{MCon, unmodified}) = 0$.

Of the 10,511 APIs in Android 13, 2473 (23.5%) are transformed and 8038 (76.5%) are pass-through. Because the framework API surface is stable across versions, upgrading requires only minor adjustments (see §6).

Refinements. We further refine the operations and policies to improve efficiency and security.

First, we enforce a *default-deny* policy. When building MCon, the signature and classification of every framework API is recorded; at runtime, an IPC call is accepted only if its signature is known at build time. A new or signature-changed method is denied until reviewed. It is a safeguard designed to close two gaps: a *version-migration gap*, where a new or changed API is accidentally missed during an upgrade, and *runtime tampering*, where a compromised service presents an API signature absent at build time (where denying is precisely the desired behavior).

Second, we introduce *many-to-one aliases* for *virtualize* to enable safe reuse of shared resources across tenants. A notable example is app binaries (Figure 4): if we install the same app package to multiple tenants (common in app testing), naively virtualizing the package name (so that X becomes X.1, X.2, etc.) would create multiple copies of the same binaries, wasting storage and memory resources. To solve this, when the package binaries are identical, we map the binary paths from all tenants to the same X.base path for read-only sharing. App configuration and data are still virtualized.

Finally, to minimize the scope of virtualization, MCon reuses the underlying OS’s multi-user support as a base layer. By mapping each tenant to a distinct user ID, we offload basic file system and process isolation to the kernel and framework. For instance, even if a bug in a shared system service incorrectly authorizes access to another tenant’s file path, the kernel’s UID-based access control will still block the actual file operation.

The multi-user mechanism is a defense-in-depth layer in MCon and is not fundamental. In cases where the underlying framework lacks multi-user support (e.g., slim IoT OSes), the IPC proxy can be extended to virtualize IDs and storage paths. Note that we currently do not support nested multi-user (i.e., multiple users in one tenant) because there is typically no need in cloud environments; just start another tenant instead.

4.3 Framework Performance Isolation

While IPC transformations provide logical isolation, they do not solve performance interference. A “noisy neighbor” can

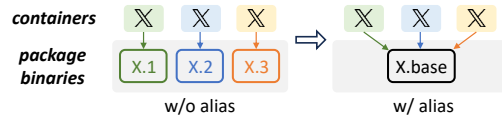


Figure 4. Many-to-one package alias improves storage efficiency.

monopolize shared system services, causing latency spikes or denial-of-service for others. To address this, MCon uses a multi-layer resource control strategy.

To partition standard compute resources (CPU, memory), we rely on cgroups [51]. To reduce contention on hot system paths, MCon implements a transparent IPC result cache within the proxy. We target read-only IPCs that are expensive but return stable results (e.g., `getInstalledPackages()`, which scans information about all apps). The proxy caches these results per-tenant, serving subsequent calls instantly without waking the system service. To ensure consistency, the proxy monitors relevant state-change events (e.g., package installation broadcasts) to actively invalidate stale cache.

For requests that cannot be cached, we enforce fair usage via a token bucket [44] regulator. Tokens are deducted from a tenant’s bucket based on the CPU time consumed by its requests. If a tenant exhausts its budget, subsequent non-critical IPCs are delayed, ensuring that no single tenant can starve the entire framework.

5 Virtual Device Slicing

With per-tenant device commands distinguishable in framework namespaces, we next turn to the para-virtualized device layer. To prevent performance interference and failure propagation, high-throughput devices (e.g., GPU, display) require a different approach from consolidation. In this section, we quantify interference bottlenecks in existing virtual device architectures (§5.1), detail our process-isolated device slices (§5.2), and introduce a complexity-aware frame pacing mechanism to regulate GPU performance jitter (§5.3).

5.1 Why Device Slices?

As discussed in §1, sharing virtual devices creates severe bottlenecks at high densities. To quantify this, we benchmark the virtual GPUs of three open-source systems: vSoC [73], Google Android Emulator (GAE) [37], and Redroid [74]. We use the hardware configuration in §7.1. We use an app that generates randomized OpenGL ES commands across 1 to 128 concurrent graphics contexts to simulate tenant density, allowing us to stress-test the GPU backends beyond the density limits of existing platforms. The app is run within a single large tenant (36 vCPUs, 120 GiB RAM), and we report the average of five 5-minute runs.

As Figure 5 shows, existing systems suffer from severe performance collapse and jitter at high concurrency. At 128 contexts, the aggregate FPS of vSoC and GAE collapses to just 2.2%–3.5% of peak performance, and the coefficient of

Algorithm 1: Complexity-Aware Frame Pacing

Input: Frame complexity $C = N_{\text{frag}} + \alpha N_{\text{vert}} + \beta N_{\text{comp}}$, Measured frame time T_{real} , Previous sleep $T_{\text{prev_sleep}}$ and error E_{prev}
Global: Conversion factor K
Param: PI gains K_p, K_i , Custom offset T_{custom}
 $K \leftarrow \text{sliding_min}(T_{\text{real}}/C)$ // 1. Update conversion factor
 $T_{\text{est}} \leftarrow C \times K$ // 2. Estimate hardware time
 $E \leftarrow T_{\text{est}} - T_{\text{real}} + T_{\text{custom}}$ // 3. Calculate error
 $T_{\text{sleep}} \leftarrow \max(T_{\text{prev_sleep}} + K_p(E - E_{\text{prev}}) + K_i E, 0)$ // 4. PI
if $T_{\text{sleep}} > 0$ **then**
 | $\text{sleep}(T_{\text{sleep}})$ // 5. Apply pacing
end
 $T_{\text{prev_sleep}} \leftarrow T_{\text{sleep}}; E_{\text{prev}} \leftarrow E$ // 6. Update history

the shared physical GPU (§5.1). Unlike well-explored CPU isolation [49, 51], GPU performance isolation remains challenging for elastic mobile workloads. Existing hardware partitioning schemes (e.g., SR-IOV [71], NVIDIA MIG [69]) offer strong isolation but lack the flexibility for elastic multi-tenancy, often supporting only a few static partitions (typically ≤ 8 [67]). Conversely, driver-level scheduling (e.g., time-sliced vGPUs [68], DRM scheduler [60]) often relies on proprietary host and guest drivers (e.g., NVIDIA) that are opaque and unavailable to mobile OSes.

To overcome these limitations, MCon operates at a higher, graphics API level. Instead of trying to control the opaque GPU hardware scheduler, we regulate the rate at which tenants *submit* frames. The idea is that jitter occurs when GPU commands queue up in drivers where we lack control; by forcing commands to wait in userspace when the hardware queue fills up, we regain control over frame pacing. This approach works with any GPU that supports standard graphics APIs, and scales to any number of tenants.

The core challenge is estimating GPU processing time for each frame to determine the right pacing rate. Naive wall-clock measurements (e.g., `GL_TIME_ELAPSED`) are unreliable because they include CPU-side bubbles caused by scheduling delays and contention. For example, a 10 ms GPU job might appear to take 50 ms if the CPU is busy. To address this, MCon employs a *minimum-filter* accounting algorithm. The key principle is that while a frame may take longer during contention, it can never take less time than the hardware physically requires to process the pixels. Therefore, by tracking the minimum observed time normalized by frame complexity over a sliding window, we can filter out transient noise and recover the true hardware processing time.

Algorithm 1 details the logic. For each frame, MCon estimates its ideal processing time (T_{est}) by scaling its complexity (C) with a learned conversion factor (K). To cover diverse workloads, MCon defines C as a composite metric from fragment, vertex, and compute shader invocations: $C = N_{\text{frag}} + \alpha N_{\text{vert}} + \beta N_{\text{comp}}$. The hardware-specific weights α and β normalize vertex and compute work into equivalent fragment invocations, calibrated by a brief microbenchmark

at hypervisor startup. Next, K is learned online via a sliding-window minimum of the ratio between measured wall-clock time and complexity, effectively rejecting outliers caused by contention. Finally, a PI controller computes the required sleep duration (T_{sleep}) to align the frame interval with the target policy (T_{custom}).

The parameter T_{custom} enables flexible, policy-driven isolation. By default, we set $T_{\text{custom}} = \overline{T_{\text{est}}} - T_{\text{est}}$ (where $\overline{T_{\text{est}}}$ is the average across all tenants) to equalize the GPU time allocated to each tenant. For latency-critical system services (e.g., the display compositor), we set $T_{\text{custom}} \rightarrow -\infty$ to bypass pacing. Alternatively, operators can enforce strict frame rate caps (e.g., 60 FPS) by setting $T_{\text{custom}} = \frac{1}{60}s - T_{\text{est}}$.

GPU Memory Isolation. Complementing temporal isolation, MCon enforces spatial isolation for GPU memory. Since GPU drivers often lack native per-process memory quotas, we track allocation APIs (e.g., `glBufferData`, `glTexImage2D`) within the virtual device slice. MCon rejects allocations that exceed the assigned quota with an out-of-memory error, preventing a single tenant from exhausting global GPU memory.

6 System Implementation

MCon has three main components: a guest OS, a hypervisor, and a container manager. The guest OS is derived from the Android Open Source Project (AOSP) [35] and currently supports Android 13–15. Since AOSP uses a Linux-based kernel, MCon uses Linux UID isolation and cgroups, but these are implementation choices; MCon can be easily ported to a microkernel with equivalent protection and resource accounting mechanisms. Inside the guest, the IPC proxy hooks into Java and C++ bindings generated by the Android Interface Definition Language (AIDL) compiler [31]. For each IPC interface, we generate a decorator that wraps the original implementation, transforms incoming calls, and forwards the sanitized call to the service. By operating on typed objects (e.g., Java Strings) instead of raw bytes, MCon improves robustness and portability across Android versions.

Porting across versions is inexpensive in practice. Migrating the policy engine from Android 13 to 15 took roughly 20 developer-hours in total: ~7h reviewing new and changed framework APIs, ~3h updating policies and container tests (11 policies and 2 tests in all), and ~10h debugging failures of the compatibility test suite and container tests. The effort stays small because maintainers reason about the stable set of tenant-scoped objects rather than the full set of 10,511 APIs—only 5 tenant-scoped objects were changed between Android 13 and 15. Our default-deny policy (§4.2) further lowers the burden by flagging every new or signature-changed API automatically at build time, so maintainers review an auto-generated diff instead of manually scanning all APIs.

The hypervisor extends vSoC [73], a QEMU-based emulator optimized for virtual device performance. It supports both x86-64 and AArch64; on x86-64, we enable ARM app

compatibility via the `ndk_translation` library [34]. The virtual devices are virtio-based para-virtualized devices built on vSoC [73], including GPU, display, codec, input, audio, net, and sensors. The container manager runs in the hypervisor process, exposing a Human Monitor Protocol (HMP) API [24] via the QEMU console for lifecycle management (e.g., tenant creation). We also provide a Python controller to automate batch operations across tenants.

MCon is implemented in 39K lines of code: 3K in the Android framework (IPC proxy and policies), 1K in the guest kernel (virtual device drivers), and 35K in the hypervisor (device slicing and management).

7 Evaluation

We evaluate MCon’s elasticity, per-mechanism contributions, robustness, overhead, and production performance.

7.1 Experimental Setup

Baselines. We compare MCon against two state-of-the-art mobile container systems, Redroid [74] and Anbox Cloud [14], and two widely used emulators, vSoC [73] and Google Android Emulator (GAE) [37]. We use the latest stable versions as of September 2025. Unless otherwise noted, we use a standard mid-range smartphone profile: Full-HD (1920×1080) resolution, 420 DPI, and a 60 Hz refresh rate. Since we target cloud workloads, all systems render offscreen.

Hardware. All experiments run on a commodity server with dual-socket Intel Xeon Silver 4210 CPUs (2.20 GHz), an NVIDIA RTX A4000 GPU with 16 GiB VRAM, 128 GiB of DDR4 ECC host memory, a 4 TiB NVMe SSD, and Gigabit Ethernet. We select x86-64 because it remains the dominant architecture in cloud data centers and the only one fully supported by all baselines (specifically GAE).

Software Configuration. The host OS is Ubuntu 24.04 LTS with NVIDIA driver 575.51.02 (OpenGL 4.6). For container-based systems (MCon, Redroid, and Anbox), we provision a large VM with 36 vCPUs and 120 GiB of memory to host multiple tenants. For emulator-based baselines (GAE and vSoC), which are one-VM-per-tenant, we configure each VM with 1 vCPU and 2 GiB of memory. We choose this configuration because it is the minimum needed to run the top-50 apps without frequent errors or crashes.

Workloads. Our workload uses the top-50 popular Android apps spanning various categories (e.g., social, entertainment, productivity, shopping, maps, AI) [64]. Since Anbox Cloud does not support Google Play Services, dependent apps (e.g., Gmail) are excluded for fairness. To simulate interaction, we inject random input events using the Android Application Exerciser Monkey [39]. Each tenant runs one app at a time. Each experiment is repeated five times.

Density Scaling. We evaluate system scalability by exponentially increasing the tenant density ($N \in \{1, 2, 4, 8, \dots\}$) until the system fails. We report metrics at each density level

and the *maximum density* achieved. Maximum density is the highest concurrency where all tenants complete the workload without failures (e.g., boot loops, crashes, or system freezes) across five repeated trials.

7.2 Tenant Provision Latency

Rapid tenant provisioning is fundamental to cloud elasticity, enabling platforms to react to bursty workloads.

Methodology. We measure provisioning latency as the interval from issuing an allocation request to the tenant becoming operational. A tenant is considered operational when two conditions are met: (i) the OS reports `sys.boot_completed=1`, and (ii) the launcher app has started. To eliminate network variance, the controller and target run on the same host.

We evaluate three cases: (1) *Concurrent cold boot*: cold booting N tenants to capture worst-case latency. Specifically, a cold boot means starting a VM for emulators, booting a full framework for container baselines, and creating tenant context and hotplugging virtual devices for MCon. (2) *Cold boot under load*: measuring the latency of a single new allocation while N active MCon tenants are running top-50 apps. (3) *Concurrent warm-pool*: provisioning N simultaneous requests from P device profiles (sweeping P from 1 to 1000, as thousands of models are used in real-world testing [57]).

We let each baseline allocate as many warm instances as possible. For MCon, we pre-allocate a pool of N framework namespaces; each allocation request creates and binds a device slice to a free namespace. For each P , we use the DeviceAtlas device-model popularity dataset [20] to define a skewed distribution over the P profiles, pre-warm according to the distribution, then independently sample the profile requested by each tenant from that distribution. Issuing N simultaneous boot requests to heavyweight systems may saturate host I/O and CPU and lead to watchdog timeouts. To ensure fairness, we test different request issuance rates for each baseline, and report the highest-throughput setting.

Results. Figure 7 shows total time to provision N tenants, and MCon outperforms all baselines across all density levels. With one tenant, MCon achieves sub-second provisioning (0.99 s), a 15–58× speedup over other systems. MCon also maintains near-linear scalability: even with 137 concurrent requests, each tenant takes only 2.3 s to provision.

Profiling reveals MCon’s latency breakdown: the multi-user isolation mechanism (for defense-in-depth) costs the most (60% on average). This is because each provision sets up and starts an Android user sequentially, causing queuing. When multi-user is disabled, the per-tenant latency decreases dramatically to 0.40 s on average, and to 0.55 s at 64 tenants. In contrast, spawning the virtual device slice is fast and parallelizable, consuming only 4% of the total time. The remaining 36% is attributed to device hotplugging and app startup.

Baseline systems exhibit a high latency floor (>15 s) due to the need to boot full frameworks. While they benefit slightly

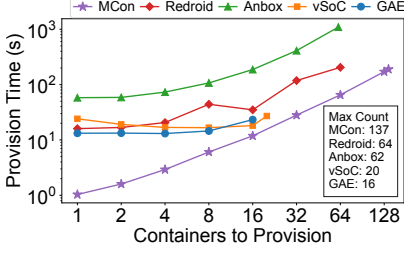


Figure 7. Concurrent provision time.

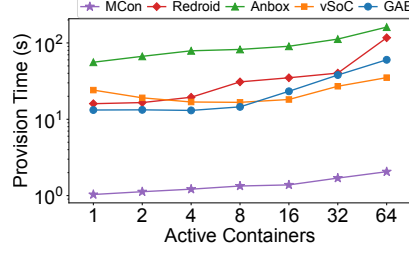


Figure 8. Provision time under load.

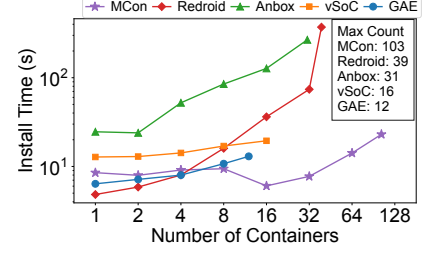


Figure 9. App deployment time.

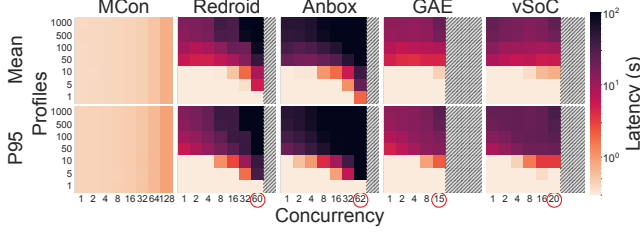


Figure 10. Concurrent provision time with warm pools. Circled numbers are the max supported concurrency of a given system.

from hardware parallelism at moderate densities (8–16 tenants), their performance degrades exponentially at high densities due to resource exhaustion. At 64 and 62 concurrent provisions, respectively, Redroid and Anbox Cloud induce massive system load, with max 1-minute load averages of 383 and 61.2 (only 8.45 for MCon).

MCon successfully provisions up to 137 concurrent tenants, whereas the best baseline fails beyond 64 tenants. Failure in MCon is due to GPU memory exhaustion: small UI elements from many tenants cause GPU memory fragmentation, eventually leaving no contiguous memory for new virtual GPU contexts. Failures in baselines are driven primarily by host memory exhaustion and watchdog timeouts.

In the provisioning-under-load scenario (Figure 8), MCon maintains its advantage, 17–78 $\times$ as fast as baselines with 64 active MCon containers. Provisioning new tenants in MCon also has negligible impact on existing workloads. In contrast, Redroid exhibits performance interference, causing a 1–9% FPS drop in running apps, as its heavy initialization process contends for CPU and GPU resources.

Figure 10 reports both mean and P95 warm-pool provisioning latency with popularity-skewed profiles. At $P=1$, every request matches the homogeneous pool, so baselines provision tenants nearly instantly (<10 ms). As P grows, skew preserves warm capacity for popular profiles but cannot cover the long tail: each baseline can hold only tens of profile-specific instances, and requests miss once no idle instance has the requested profile. The effect appears first in P95 latency and then in the mean; at $P \geq 50$, both approach the cold-boot latencies in Figure 7. In contrast, MCon’s mean and P95 latency remain sub-second across all evaluated profile counts, because any pre-allocated namespace can bind to any requested device slice.

7.3 Application Deployment Time

Rapid app deployment is important for CI/CD and testing, where each run often pre-installs apps across many tenants.

Methodology. We measure end-to-end app deployment latency from when the controller starts issuing installation commands to when all the top-50 apps are installed on all N tenants. The N tenants install the apps in parallel; within each tenant, the 50 apps are installed sequentially because AOSP does not support parallel installation [5]. To prevent I/O skew, all the top-50 app packages are pre-staged in a RAM disk on the host.

Results. As Figure 9 shows, for a single tenant, all systems complete in tens of seconds (12 s on average), as deployment is bound by Android Package Manager’s sequential installation. However, as tenant count increases, a divergence emerges. MCon’s total deployment time grows slowly, increasing to only 1.7 $\times$ when scaling from 1 to 64 tenants. In contrast, baselines exhibit $O(N)$ scaling, as they perform N times the I/O to copy and install packages for N isolated instances. Consequently, at 32 tenants, MCon completes deployment 9.0 $\times$ as fast as the best baseline.

The near- $O(1)$ scalability of MCon comes from framework namespaces: instead of copying the package per tenant, MCon only performs a logical mapping of a shared package into each tenant’s virtual view. Therefore, deployment cost scales with the number of *distinct* package binaries M rather than the number of tenants N . This benefits common use cases like agent training and app testing, where a small app set is installed across many tenants ($M \ll N$); the performance gains disappear when $M \approx N$ (e.g., a public multi-customer CI service where every tenant deploys a unique build), a workload largely outside MCon’s scope.

7.4 Application Throughput

App throughput under varying tenant densities is a key cost-efficiency indicator for performance-sensitive workloads.

Metrics. Following standard industry practice [29, 83], we use Frames Per Second (FPS) as the primary metric for app smoothness. FPS data is collected via the Android `dumpsys gfxinfo` command. We also monitor CPU and GPU usage, host and GPU memory use, and stability metrics including Mean Time Between Failures (MTBF) and recovery time.

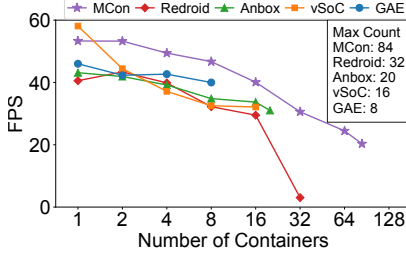


Figure 11. App throughput.

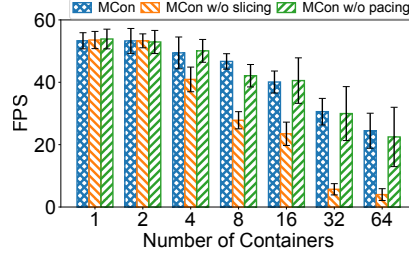


Figure 12. Performance breakdown.

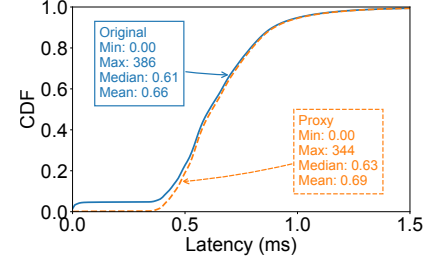


Figure 13. IPC latency overhead.

Workload Patterns. We run the top-50 apps in two patterns: (i) *Random*: each tenant independently selects a random app, and (ii) *Round-Robin*: all tenants execute the same app simultaneously, cycling through the apps in different rounds. Each round consists of a startup phase to ensure the app is started in all tenants, followed by a 5-minute measurement window to capture steady-state performance. We conduct 100 back-to-back rounds for each configuration (system, pattern, density) to ensure statistical significance.

Compatibility Results. Out of the 50 apps, MCon, Redroid, Anbox Cloud, vSoC, and GAE can respectively run 48, 48, 45, 48, and 46 apps successfully. The two apps that fail on MCon also fail on other platforms; one fails because ARM binary translation is at fault, the other because of anti-emulation checks. This demonstrates that, with careful design and implementation, framework namespaces do not break application compatibility; incompatible apps are excluded from the final results of each system.

Anti-emulation is a hard problem shared by all emulator- and container-based stacks (e.g., banking apps refuse to run in a virtual device), and is largely orthogonal to MCon. Nor do MCon’s mechanisms introduce a distinguishing fingerprint: framework namespaces present an ordinary single-user framework, device slices expose the same virtio devices as vSoC, and the IPC proxy adds only ~4% mean latency (§7.8), within normal device-to-device variance, so IPC timing analysis cannot readily distinguish MCon from other emulators or physical devices.

Throughput Results. Figure 11 presents the average FPS of successfully executed apps. At low densities (1–8 tenants), all systems deliver comparable performance (standard error < 6.8), close to the 60 FPS ceiling. Here, hardware resources are abundant and virtualization efficiency is the main factor.

MCon outperforms the baselines beyond 16 tenants. On our server, MCon sustains relatively high FPS for up to 84 tenants. In contrast, baselines degrade rapidly. Redroid, the most competitive baseline, sees its aggregate throughput collapse from 16 to 32 tenants (90% FPS drop). Profiling reveals that this collapse is driven by memory pressure: at 32 tenants, Redroid exhausts the available host memory, triggering aggressive swapping and OOM kills. MCon avoids this cliff through its memory-efficient architecture; at 84 tenants, it still has host memory headroom, but further scaling fails

when fragmented VRAM cannot satisfy allocations for new virtual GPU contexts. MCon’s FPS degrades gracefully with increasing density, primarily due to GPU compute saturation (we observe 100% GPU utilization at ≥ 16 tenants).

Thus, on our server, MCon removes the baseline host memory wall and shifts the hard capacity bottleneck to VRAM fragmentation. Unfortunately, this fragmentation cannot be addressed within MCon itself: Android renders through OpenGL ES, which delegates VRAM placement to the host graphics driver, and the proprietary NVIDIA driver exposes no application-controlled defragmentation mechanism. Operators can instead raise the allocation ceiling with a higher-memory GPU: when an NVIDIA RTX 4090D with 24 GiB of VRAM is used, MCon supports up to 106 tenants.

Reliability Results. At 1 and 64 tenants, MCon framework MTBF is 243 and 34 hours, respectively; a framework restart interrupts all co-located tenants. Device-slice failures affect one tenant and have MTBFs of 69 and 18 hours. MCon automatically recovers either component, but live app state is lost, so affected apps restart and in-flight work is retried. Average per-tenant uptime is 99.99% at 1 tenant (99.92%–99.99% for baselines), and 99.65% at 64 tenants. Combining uptime with event rates implies a mean outage, including app restart, of ~20 seconds per failure at 1 tenant and ~2.5 minutes at 64 tenants. Thus, density is not a free lunch; operators should select the highest workload-tested density satisfying both per-tenant availability and outage allowances.

7.5 Density and Resource Consumption

We next quantify resource consumption to identify density bottlenecks. Since memory capacity typically dominates server costs [32], we focus on memory footprints when tenants are idle or running apps.

Idle Tenant. We first measure the overhead of provisioning idle tenants. MCon incurs a one-time fixed cost of 2.1 GiB host memory and 80 MiB GPU memory to initialize the framework and the VM stack. Beyond this, the *marginal cost* for each additional tenant is minimal: only 357 MiB of host memory and 69 MiB of GPU memory. This results from our namespace design, where new tenants only require lightweight virtual device structures and a private data view.

In contrast, baseline systems exhibit high linear resource growth. Redroid, Anbox Cloud, vSoC, and GAE require 1.7–3.0 GiB of host memory and 70–125 MiB of GPU memory for

every new tenant. Consequently, MCon reduces memory consumption by 85%–92% (host) and 1%–45% (GPU), compared to state-of-the-art alternatives.

Active Workloads. During the application benchmark in §7.4, an average top-50 app consumes 245 MiB of host memory and 46 MiB of GPU memory. Such overhead is intrinsic to apps and remains constant across systems. With the app overhead added, MCon retains a decisive advantage. The total per-tenant footprint (system+app) in MCon is 70%–82% lower than baselines in host memory and 1%–33% lower in GPU memory. This explains the density gap in §7.4: baselines hit host-memory limits at moderate densities, while MCon scales to 84 tenants before VRAM fragmentation limits further scaling.

7.6 Performance Breakdown

We now isolate the contributions of MCon’s core mechanisms when running the top-50 apps.

Framework Consolidation. Consolidation removes the host memory wall that caps the baselines (§7.5): by amortizing one framework across tenants, MCon gains enough memory headroom to reach 84 tenants on our server before encountering a separate VRAM-fragmentation ceiling. Consolidation alone, however, reduces throughput: with slicing disabled, MCon delivers 14%–26% lower FPS than the baselines at a density of 16. This degradation stems from the synchronization bottleneck in §5.1.

Virtual Device Slicing. Slicing is what makes MCon performant under high density. Figure 12 shows that slicing becomes more effective as density rises (6.3× FPS improvement at 64 tenants). Profiling reveals that the top-50 apps issue 55–940 (average 133) GPU commands per frame. Without slicing, these commands would overwhelm a single driver with 200K commands per second at 64 tenants; slicing instead routes them to independent device slices. Slicing has negligible impact on density, since each slice starts in <50 ms and adds <1% memory (§5.2).

Complexity-Aware Frame Pacing. Disabling pacing does not degrade average throughput but hurts performance stability. As shown in Figure 12, FPS *stdev.* increases by 69%–107% at 16–64 tenants without pacing.

To stress test the pacing algorithm, we use two rare, extremely volatile workloads: (i) random 10× particle bursts in a physics simulation (same shader, 10× more per-frame GPU work), and (ii) a shader-switching benchmark (10 shader changes per second). Under (i), the baseline without pacing suffers a 2.1× *stdev.* of MCon; that is because frame complexity is *measured* rather than predicted in our algorithm, so it absorbs sudden spikes in intra-frame GPU work. With (ii), the algorithm produces a 15% higher *stdev.* than the baseline; that is because per-invocation cost shifts and the global *K* needs to re-converge to adapt to the new workload. When

Issue class	Trigger	#
Namespace escape	App-supplied .X suffix	1
Null dereference	Null field in recursive transform	3
Resource exhaustion	Deep nesting / oversized array	2

Table 3. Previously unknown issues found by fuzzing.

frequent shader switches dominate, the estimate can be further improved by keying *K* per shader program (observable via `glUseProgram`) instead of using a single global *K*, which we leave as future work.

7.7 Adversarial Testing

We evaluate the robustness of framework namespaces with a test suite that combines 19 hand-written adversarial tests for boundary cases we anticipate, with mutation-based fuzzing for the rest. The 19 hand-written tests include common attacks like out-of-namespace file reads, cross-tenant screen capture, and global clipboard access. The fuzzer uses the hand-written cases and captured IPC traces as seeds and stress-tests transformed IPC interfaces with generated malformed calls. All calls are issued from an unprivileged in-tenant app—the buggy or curious adversary in our threat model (§3). Because MCon transforms typed objects (§6), the fuzzer mutates typed objects rather than binary, so inputs survive deserialization checks and actually reach the proxy. The following areas are covered:

- *Namespace confusion*: inject sibling or non-existent tenant suffixes (.X);
- *Strings*: empty, null, oversized, and path-traversal strings;
- *Structural*: null, deeply nested, and oversized objects;
- *Identity spoofing*: forged user IDs, binder objects, etc.

Results. The fuzzer runs for 48 hours and executes 2.4M transactions, reaching 1879 of 2473 (76%) transformed APIs. The remaining APIs are still mediated by framework namespaces (§4.2), but are not exercised at runtime, because they are unused by the top-50 apps or require unavailable hardware. Six previously unknown issues are uncovered and fixed (Table 3). The most security-relevant is a *namespace-escape* case: the proxy passes through an app-supplied package name that already carried a .X suffix, causing the framework to treat it as a non-namespaced app. The remaining five are robustness defects: three null dereferences in recursive transforms and two resource-exhaustion paths (deep nesting and oversized arrays), fixed with null guards and depth/size caps. After hardening, a re-run confirms that every malformed transaction is safely rejected or virtualized.

7.8 Overhead Analysis

We quantify the IPC overhead of framework namespaces by comparing end-to-end IPC latency and throughput with and without the IPC proxy. We define end-to-end latency as the round-trip time from a client (app) issuing a Binder transaction to receiving the response from the system service.

Since only one tenant can run without the proxy, we compare using a single tenant. We run the top-50 apps each for five minutes, instrumenting the Binder driver to capture precise timing for every IPC transaction.

Latency Results. Figure 13 presents the latency CDF. The proxy introduces a modest overhead: the mean IPC latency increases from 660 μ s to 688 μ s with the proxy, an average overhead of 28 μ s (4%) per transaction. While the 90th percentile IPC latency increases from 890 μ s to 901 μ s with the proxy, the 99th percentile decreases from 1371 μ s to 1245 μ s. This improvement is due to caching in framework namespaces, which avoids heavy IPC calls like package queries (the main contributor to tail latency).

To understand the costs, we decompose the 28 μ s average overhead: (i) policy table lookup (16.5 μ s, 58%), which matches IPC transactions to policies; (ii) deep-copying mutable framework objects (9.6 μ s, 34%); and (iii) transformation operations (e.g., `virtualize`) (2.3 μ s, 8%). The copying in (ii) is essential for correctness: system services often return direct references to internal data structures for performance; the proxy must clone these objects to not corrupt the framework’s internal state.

Throughput. The system sustains an average throughput of 560 and 558 transactions per second with and without the proxy (<1% difference). We do not observe a statistical difference in either tail or per-app throughput, indicating that the proxy introduces negligible overhead to IPC throughput.

7.9 Production Deployment

MCon is the Android-environment backend for GUI-agent RL training at Infinigence AI. A training run executes hundreds of *rollouts* back-to-back; each provisions a clean tenant, deploys the target app, lets the agent interact, and then resets. Production spans an NVIDIA RTX 4090 render pool and an NVIDIA A100 compute pool, with 128–512 Android tenants active per run. Rollouts last from 5 seconds to more than 5 minutes (about 60 seconds on average), and batched resets trigger up to 512 cold provisions every few minutes. Before MCon, preparing clean environments consumed more than 50% of rollout wall-clock time. MCon cuts preparation time to less than one-third and doubles maximum concurrency, together improving end-to-end RL training speed by 56% [64].

8 Related Work

Mobile Cloud Computing. Early work offloaded resource-intensive components like databases [16, 18, 42] to the cloud. Recent trends shift entire app execution to the cloud, treating mobile systems as compute environments. This supports app testing [30, 41, 57], agent training [12, 80], cloud gaming [13, 17, 56, 66, 85], and cloud phones [2, 47]. MCon addresses elasticity challenges in hosting such workloads.

API Virtualization. API virtualization and namespaces are well established in both mobile [8, 77, 86] and PC/server

computing [21, 61]. However, while related work virtualizes POSIX or narrow domain-specific interfaces, MCon builds an end-to-end architecture that tackles a large and evolving framework API surface, heterogeneous device profiles, and cross-layer performance isolation—all without breaking unmodified apps.

Mobile Containers. Mobile containers have evolved along two paths. On-device solutions create multiple personas for a single user, using kernel-level namespaces (e.g., Cells [8]) or native OS features like multi-user [36] and Work Profiles [4, 40]. In contrast, cloud-based solutions [14, 74, 84] follow the server container paradigm [21, 61] and share the host kernel. Recent work optimizes aspects like GPU performance [29, 73, 87], distributed rendering [55, 79], reliability [57, 76], framework security [58, 77, 86], and storage [54]. However, these systems do not fit elastic multi-tenant clouds: multi-user and Work Profiles [4, 36] target trusted co-users on one device, retaining a global namespace, cross-user visibility, and a single shared device profile (§2.2); Cells [8] and other work still duplicate the heavy framework per instance. MCon addresses this orthogonal problem.

Elastic Clouds. Rapid provisioning is key to cloud platforms, impacting QoS [45, 63] and infrastructure costs [43]. Server platforms boost startup via snapshots [23, 70], microVMs [1], unikernels [62], or application-level tooling [53]. These approaches achieve millisecond latency by minimizing environments for stateless functions or individual binaries. However, in mobile contexts the bottleneck is fundamentally different: the heavy userspace framework dominates provisioning time and limits density, which VM- and kernel-level optimizations do not address. This mobile-specific bottleneck creates the elasticity challenges that MCon addresses.

9 Conclusion

This paper presents MCon, the first architecture that brings elasticity to mobile containers. Unlike approaches that directly port server virtualization to mobile OSes, MCon leverages mobile systems’ structure to consolidate the resource-heavy framework across tenants. For controlled, single-provider clusters, we show that tenant isolation need not sacrifice efficiency, by virtualizing the IPC boundary with framework namespaces and partitioning hardware access with virtual device slices. Our evaluation shows that MCon delivers sub-second provisioning and, under popular workloads, reduces host memory use by 70%–82% and GPU memory use by 1%–33%. Together, these results make MCon a practical foundation for next-generation mobile cloud computing.

Acknowledgments

We are grateful to our shepherd, Yubin Xia, for his invaluable feedback. We thank the anonymous reviewers for their insightful comments. This work is supported in part by the National Natural Science Foundation of China under grants 62332012, 62472245, and 62472127.

References

- [1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proc. of USENIX NSDI*, pages 419–434, 2020.
- [2] Alibaba. Cloud Phone - Virtual Phone | Alibaba Cloud, 2025. https://www.alibabacloud.com/product/elastic_cloud_phone.
- [3] Android Developers. Application Sandbox, 2025. <https://source.android.com/docs/security/app-sandbox>.
- [4] Android Developers. Private Space, 2025. <https://source.android.com/docs/security/features/private-space>.
- [5] Android Developers. Android PackageManagerService Source Code, 2026. <https://cs.android.com/android/platform/superproject/main/+main:frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java>.
- [6] Android Developers. MediaStore API Reference, 2026. <https://developer.android.com/reference/android/provider/MediaStore>.
- [7] Android Developers. Privacy Changes in Android, 2026. <https://developer.android.com/about/versions/10/privacy/changes>.
- [8] Jeremy Andrus, Christoffer Dall, Alexander Van’t Hof, Oren Laadan, and Jason Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proc. of ACM SOSP*, pages 173–187, 2011.
- [9] Apple. Apple-Oss-Distributions/Xnu, 2025. <https://github.com/apple-oss-distributions/xnu>.
- [10] Apple. Apple XPC, 2025. <https://developer.apple.com/documentation/xpc>.
- [11] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proc. of ACM CCS*, pages 217–228, 2012.
- [12] Hao Bai, Yifei Zhou, Mert Cemri, Jiayi Pan, Alane Suhr, Sergey Levine, and Aviral Kumar. DigiRL: Training In-The-Wild Device-Control Agents with Autonomous Reinforcement Learning. In *Proc. of NeurIPS*, pages 12461–12495, 2024.
- [13] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. *SIGOPS Oper. Syst. Rev.*, 39:277–290, 2005.
- [14] Canonical. Anbox Cloud - Scalable Android in the Cloud, 2025. <https://canonical.com/anbox-cloud>.
- [15] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, Hongyang Yang, Hui Wang, Jie Yin, Yu Peng, and Fengwei Xu. Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel. In *Proc. of USENIX OSDI*, pages 465–485, 2024.
- [16] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proc. of ACM EuroSys*, pages 301–314, 2011.
- [17] CloudMoon. CloudMoon Mobile Cloud Gaming, 2025. <https://web.cloudmoonapp.com/>.
- [18] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. of ACM MobiSys*, pages 49–62, 2010.
- [19] Savino Dambra, Leyla Bilge, Platon Kotzias, Yun Shen, and Juan Caballero. One Size Does Not Fit All: Quantifying the Risk of Malicious App Encounters for Different Android User Profiles. In *Proc. of USENIX Security*, pages 5683–5700, 2023.
- [20] DeviceAtlas. DeviceAtlas: Web Usage by Device Model and Country, 2026. <https://deviceatlas.com/device-data/explorer/>.
- [21] Docker Developers. Docker: Accelerated Container Application Development, 2025. <https://www.docker.com/>.
- [22] Micah Dowty and Jeremy Sugerman. GPU Virtualization on VMware’s Hosted I/O Architecture. *SIGOPS Oper. Syst. Rev.*, 43:73–82, 2009.
- [23] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proc. of ACM ASPLOS*, pages 467–481, 2020.
- [24] Fabrice Bellard. QEMU Monitor — QEMU Documentation, 2025. <https://qemu-project.gitlab.io/qemu/system/monitor.html>.
- [25] Fabrice Bellard. Vhost-User Protocol — QEMU Documentation, 2025. <https://qemu-project.gitlab.io/qemu/interop/vhost-user.html>.
- [26] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proc. of ACM CCS*, pages 627–638, 2011.
- [27] Foundryzero. Binder-Trace, 2025. <https://github.com/foundryzero/binder-trace>.
- [28] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proc. of ACM ASPLOS*, pages 3–18, 2019.
- [29] Di Gao, Hao Lin, Zhenhua Li, Chengen Huang, Yunhao Liu, Feng Qian, Liangyi Gong, and Tianyin Xu. Trinity: High-Performance Mobile Emulation through Graphics Projection. In *Proc. of USENIX OSDI*, pages 285–301, 2022.
- [30] GenyMobile. Genymotion - Android Emulator in the Cloud and for PC & Mac, 2026. <https://www.genymotion.com/>.
- [31] Google. AIDL Overview. <https://source.android.com/docs/core/architecture/aidl>.
- [32] Google. Well-Architected Framework: Cost Optimization Pillar | Cloud Architecture Center. <https://docs.cloud.google.com/architecture/framework/cost-optimization>.
- [33] Google. Android Binder IPC, 2025. <https://source.android.com/docs/core/architecture/hidl/binder-ipc>.
- [34] Google. Android NDK, 2025. <https://developer.android.com/ndk>.
- [35] Google. Android Open Source Project, 2025. <https://source.android.com/>.
- [36] Google. Android Support for Multiple Users, 2025. <https://source.android.com/docs/devices/admin/multi-user>.
- [37] Google. Google Android Emulator, 2025. <https://developer.android.com/studio/run/emulator>.
- [38] Google. The Compatibility Test Suite (CTS) Overview, 2025. <https://source.android.com/docs/compatibility/cts>.
- [39] Google. UI/Application Exerciser Monkey, 2025. <https://developer.android.com/studio/test/other-testing-tools/monkey>.
- [40] Google. Work Profiles | Android Enterprise, 2025. <https://developer.android.com/work/managed-profiles>.
- [41] Google. Firebase Test Lab, 2026. <https://firebase.google.com/docs/test-lab>.
- [42] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. COMET: Code Offload by Migrating Execution Transparently. In *Proc. of USENIX OSDI*, pages 93–106, 2012.
- [43] Tian Guo, Upendra Sharma, Prashant J. Shenoy, Timothy Wood, and Sambit Sahu. Cost-Aware Cloud Bursting for Enterprise Applications. *ACM Transactions on Internet Technology*, 13(3):10:1–10:24, 2014.
- [44] Juha Heinanen and Roch Guerin. A Single Rate Three Color Marker. Request for Comments RFC 2697, Internet Engineering Task Force, September 1999.
- [45] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. In *Proc. of USENIX ICAC*, pages 23–27, 2013.
- [46] Wenyi Hong, Weihang Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, and Jie Tang. CogAgent: A Visual Language Model for GUI Agents. In *Proc. of IEEE/CVF CVPR*, pages 14281–14290, 2024.

- [47] Huawei Cloud. Cloud Phone Host, 2026. <https://www.huaweicloud.com/intl/en-us/product/cph.html>.
- [48] Iliyan Malchev. Here Comes Treble: A Modular Base for Android, 2017. <https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html>.
- [49] Intel. Intel Resource Director Technology, 2025. <https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html>.
- [50] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In *Proc. of ACM SoCC*, pages 445–451, 2017.
- [51] Kernel Developers. Cgroups Linux Manual, 2025. <https://man7.org/linux/man-pages/man7/cgroups.7.html>.
- [52] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding Ephemeral Storage for Serverless Analytics. In *Proc. of USENIX ATC*, pages 789–794, 2018.
- [53] Tom Kuchler, Pinghe Li, Yazhuo Zhang, Lazar Cvetković, Boris Gornov, Tobias Stocker, Leon Thomm, Simone Kalbermatter, Tim Notter, Andrea Lattuada, and Ana Klimovic. Unlocking True Elasticity for the Cloud-Native Era with Dandelion. In *Proc. of ACM SOSP*, pages 944–961, 2025.
- [54] Zeyu Lei, Güliz Seray Tuncay, Beatrice Carissa Williem, Z. Berkay Celik, and Antonio Bianchi. ScopeVerif: Analyzing the Security of Android’s Scoped Storage via Differential Analysis. In *Proc. of ISOC NDSS*, 2025.
- [55] Linsheng Li, Bin Yang, Cathy Bao, Shuo Liu, Randy Xu, Yong Yao, Mohammad R. Haghighat, Jerry W. Hu, Shoumeng Yan, and Zhengwei Qi. DroidCloud: Scalable High Density AndroidTM Cloud Rendering. In *Proc. of ACM MM*, pages 3348–3356, 2020.
- [56] Yang Li, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Feng Qian, Jing Yang, Hao Lin, Yunhao Liu, Bo Xiao, Xiaokang Qin, and Tianyin Xu. Dissecting and Streamlining the Interactive Loop of Mobile Cloud Gaming. In *Proc. of USENIX NSDI*, pages 595–611, 2025.
- [57] Hao Lin, Jiaxing Qiu, Hongyi Wang, Zhenhua Li, Liangyi Gong, Di Gao, Yunhao Liu, Feng Qian, Zhao Zhang, Ping Yang, and Tianyin Xu. Virtual Device Farms for Mobile App Testing at Scale: A Pursuit for Fidelity, Efficiency, and Accessibility. In *Proc. of ACM MobiCom*, pages 1–17, 2023.
- [58] Yan Lin, Joshua Wong, Xiang Li, Haoyu Ma, and Debin Gao. Peep With A Mirror: Breaking The Integrity of Android App Sandboxing via Unprivileged Cache Side Channel. In *Proc. of USENIX Security*, pages 2119–2135, 2024.
- [59] Zichuan Lin, Feiyu Liu, Yijun Yang, Jiafei Lyu, Yiming Gao, Yicheng Liu, Zhicong Lu, Yangbin Yu, Mingyu Yang, Junyou Li, Deheng Ye, and Jie Jiang. UI-Voyager: A Self-Evolving GUI Agent Learning via Failed Experience. *arXiv preprint arXiv:2603.24533*, 2026.
- [60] Linux Kernel Community. DRM Memory Management — The Linux Kernel Documentation, 2025. <https://www.kernel.org/doc/html/latest/gpu/drm-mm.html>.
- [61] LXC Developers. Linux Containers LXC. <https://linuxcontainers.org/>.
- [62] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proc. of ACM ASPLOS*, pages 461–472, 2013.
- [63] Ming Mao and Marty Humphrey. Auto-Scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *Proc. of ACM SC*, pages 1–12, 2011.
- [64] MCon Authors. MCon Open Source Website, 2026. <https://github.com/VirtualSoC/mcon-docs>.
- [65] nouveau. Nouveau, 2025. <https://nouveau.freedesktop.org/>.
- [66] now.gg. Now.Gg Mobile Cloud, 2025. <https://now.gg>.
- [67] NVIDIA. NVIDIA A100 | Tensor Core GPU. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-nvidia-us-2188504-web.pdf>.
- [68] NVIDIA. NVIDIA Virtual GPU (vGPU) Software. <https://docs.nvidia.com/vgpu/index.html>.
- [69] NVIDIA. MIG User Guide — NVIDIA Multi-Instance GPU User Guide, 2025. <https://docs.nvidia.com/datacenter/tesla/mig-user-guide/>.
- [70] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proc. of USENIX ATC*, pages 57–70, 2018.
- [71] PCI-SIG. Specifications | PCI-SIG, 2010. <https://pcisig.com/specifications>.
- [72] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. In *Proc. of USENIX OSDI*, pages 1–16, 2014.
- [73] Jiaxing Qiu, Zijie Zhou, Yang Li, Zhenhua Li, Feng Qian, Hao Lin, Di Gao, Haitao Su, Xin Miao, Yunhao Liu, and Tianyin Xu. vSoC: Efficient Virtual System-on-Chip on Heterogeneous Hardware. In *Proc. of ACM SOSP*, pages 558–573, 2024.
- [74] Redroid. Redroid Documentation, 2025. <https://github.com/remote-android/redroid-doc>.
- [75] Mohammad Shahradd, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proc. of USENIX ATC*, pages 205–218, 2020.
- [76] Yuru Shao, Ruowen Wang, Xun Chen, Ahemd M. Azab, and Z. Morley Mao. A Lightweight Framework for Fine-Grained Lifecycle Control of Android Applications. In *Proc. of ACM EuroSys*, pages 1–14, 2019.
- [77] Wenna Song, Jiang Ming, Lin Jiang, Yi Xiang, Xuanchen Pan, Jianming Fu, and Guojun Peng. Towards Transparent and Stealthy Android OS Sandboxing via Customizable Container-Based Virtualization. In *Proc. of ACM CCS*, pages 2858–2874, 2021.
- [78] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *Proc. of USENIX ATC*, pages 109–120, 2014.
- [79] Dongjie Tang, Cathy Bao, Yong Yao, Chao Xie, Qiming Shi, Marc Mao, Randy Xu, Linsheng Li, Mohammad R. Haghighat, Zhengwei Qi, and Haibing Guan. CARE: Cloudified Android OSes on the Cloud Rendering. In *Proc. of ACM MM*, pages 4582–4590, 2021.
- [80] Daniel Toyama, Philippe Hamel, Anita Gergely, Gheorghe Comanici, Amelia Glaese, Zafarali Ahmed, Tyler Jackson, Shibl Mourad, and Doina Precup. AndroidEnv: A Reinforcement Learning Platform for Android. *arXiv preprint arXiv:2105.13231*, 2021.
- [81] VirGL Developers. VirGL 3D Graphics Library Documentation, 2025. <https://docs.mesa3d.org/drivers/virgl.html>.
- [82] Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-Agent: Autonomous Multi-Modal Mobile Device Agent with Visual Perception. *arXiv preprint arXiv:2401.16158*, 2024.
- [83] Shibo Wang, Shusen Yang, Xiao Kong, Chenglei Wu, Longwei Jiang, Chenren Xu, Cong Zhao, Xuesong Yang, Jianjun Xiao, Xin Liu, Changxi Zheng, Jing Wang, and Honghao Liu. Pudica: Toward Near-Zero Queuing Delay in Congestion Control for Cloud Gaming. In *Proc. of USENIX NSDI*, pages 113–129, 2024.
- [84] Waydroid. Waydroid | Android in a Linux Container, 2025. <https://waydro.id/>.
- [85] Jiangkai Wu, Yu Guan, Qi Mao, Yong Cui, Zongming Guo, and Xing-gong Zhang. ZGaming: Zero-Latency 3D Cloud Gaming by Image Prediction. In *Proc. of ACM SIGCOMM*, pages 710–723, 2023.
- [86] Xiaobo Xiang, Ren Zhang, Hanxiang Wen, Xiaorui Gong, and Baoxu Liu. Ghost in the Binder: Binder Transaction Redirection Attacks in Android System Services. In *Proc. of ACM CCS*, pages 1581–1597, 2021.
- [87] Qifan Yang, Zhenhua Li, Yunhao Liu, Hai Long, Yuanchao Huang, Jiaming He, Tianyin Xu, and Ennan Zhai. Mobile Gaming on Personal

- Computers with Direct Android Emulation. In *Proc. of ACM MobiCom*, pages 1–15, 2019.
- [88] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring. In *Proc. of USENIX ATC*, pages 387–400, 2012.
- [89] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. AppAgent: Multimodal Agents as Smartphone Users. In *Proc. of ACM CHI*, pages 70:1–70:20, 2025.