

MYKOS: performance by design

Making complete customer environments available quickly and keeping them responsive

Engineering milestones, roadblocks, and design decisions Released implementation baseline: 1.5.20261024-146

Performance became a central engineering goal of MYKOS: get a complete customer environment ready quickly, then keep management and monitoring from getting in its way. The target is a usable system with its services, accounts, customer configuration, data, and connectivity intact. A fast container launch is only one part of that result.

The main gains came from changing when work happens. Prepare and boot an OS once, preserve the state that can be reused, compose an available backup into that prepared runtime, and collect shared observations instead of repeatedly entering guests. Keep expensive installation work in a recoverable preparation pipeline. Make image edits reusable through the GUI. Network correctness and coordinated capture make those shortcuts dependable.

This account follows those milestones and the difficult problems behind them. The project owner's experience is that the system is exceptionally responsive, including customer-backup workflows that become usable in seconds. Those are observed outcomes; comparable cold/warm and concurrent-load benchmarks remain future measurement work.

“Whole-instance” is the managed unit. Native Linux system containers run a complete userspace sharing the host kernel; Windows, macOS, and Ubuntu desktop environments use QEMU-backed VMs inside Podman wrappers. Their execution and isolation models remain distinct.

The performance milestones

Milestone	Work removed from the repeated path
Reuse already-booted state	Avoid repeating supported VM boot and initialization work; restore paired memory, device, and disk state.
Restore the customer environment from an available Linux backup	Avoid rebuilding accounts, passwords, ownership, and customer setup manually or copying every data tree before startup.
Edit and capture images through the GUI	Reuse an existing prepared environment for changes instead of reinstalling the OS for each iteration.
Pipeline unattended image preparation	Complete provisioning once, retain stage progress and completed captures, and recover from late failures.
Observe from the host and share collected state	Avoid repeated guest polling commands for each dashboard/API consumer; publish timestamped observations.
Preserve network identity and recovery correctness	Prevent delayed IP loss, broken DNS, unsafe fallback, and inconsistent captures from undoing the startup gains.

The first five milestones reduce repeated work. The remaining engineering work keeps the resulting environment usable and recoverable. The order is an architectural account, not a measurement of engineering hours or a claim that every mechanism accelerates every workload.

1. Reusing an already-booted instance

The obstacle. An installed image could contain all the right files and still repeat the complete guest boot path. A filesystem commit does not preserve running processes, guest RAM, or emulated device state. Keeping the disk alone lost the work already performed during installation, initialization, and boot.

The decision. Treat prepared execution state as part of the reusable image. For supported VM paths, MYKOS asks QEMU to capture memory and device state, waits for migration completion and the source pause, flushes the participating disks, and copies the corresponding storage using required reflinks. A manifest records the OS/image identity, CPU environment, RAM size, QEMU version, and capture metadata. The source resumes after capture.

Creation gives each customer ID private storage. A one-time restore request starts the instance from its prepared state, and the helper checks restoration rather than treating a running wrapper as sufficient proof. The useful discovery was that repeating a boot is often unnecessary when the required state already exists.

What this buys us. Operator experience recorded a substantial startup improvement from reusing already-booted state. The mechanism explains that improvement, but there is no controlled cross-OS benchmark establishing a universal speedup. A persisted warm capture also differs from keeping every future instance resident in RAM: it avoids repeated guest boot work while still paying clone, restore, identity, and readiness costs.

The tradeoff. Memory/device state imposes compatibility and storage requirements. It is not a portable replacement for every cold boot, and external sessions or mounted data can change while a capture is retained. Native Linux checkpoints use CRIU separately; the QEMU result must not be generalized to all native containers.

Performance consequence. Installation and supported boot preparation move out of the repeated creation path; restore and readiness remain real costs.

2. A Linux backup became a working customer system in seconds

The objective. Bring an available Linux backup into a usable environment with customer accounts, passwords, file ownership, application data, and configuration already in place. The observed result was restoration in seconds for the tested workflow, instead of manually rebuilding the customer's setup. The important engineering contribution is the composition process that makes that possible.

The obstacle. A backup contains numeric identities and configuration inherited from another machine. Base-image users can collide with customer UIDs or GIDs. A complete customer `/etc` can also replace settings needed by the new runtime; inherited PAM limits exposed this when restored consoles could not follow the expected login path. Copying files alone did not recreate a working system.

The resolution. Start with a compatible prepared Linux OS, read the backup account databases, preserve customer numeric identities, and relocate conflicting image accounts within the instance. Reconcile `passwd`, `group`, `shadow`, and `gshadow` into private runtime files before binding the customer trees. Existing customer password hashes can remain usable without asking an operator to recreate those passwords. Root credentials follow the base-image policy; they are not blindly replaced by the backup. Accounts without usable shadow entries are handled explicitly, including locking when required.

Customer data keeps its numeric ownership rather than undergoing a wholesale ownership rewrite. Private machine settings supply the new runtime's hostname, machine identity, selected boot settings, and PAM configuration. Compatible customer paths are mounted directly, avoiding a mandatory full data copy before startup. Together these steps preserve the customer environment while keeping the new instance's identity explicit.

Why it can take seconds. OS preparation has already happened, and the backup is already available. The critical path becomes configuration/account reconciliation, mounts, and instance startup. This is an operator-observed outcome, not a timed guarantee or a claim that transferring an arbitrary backup, installing its OS, or validating every application takes seconds. OS compatibility, storage latency, data preparation, and application recovery still matter.

The tradeoff and lesson. Direct mounts are writable live data. Use a working copy when the recovery source must remain unchanged, and coordinate application protection with external writers. The recoverable unit is the customer system's data plus identities plus configuration on a compatible runtime. This is the clearest reason that preserving a whole environment can matter more than repackaging its individual services.

Performance consequence. Direct composition removes the mandatory full-copy step and reuses the customer setup instead of reconstructing it.

3. GUI image editing shortened the preparation loop

The obstacle. Engineers needed to open an existing prepared image in the GUI, experiment in its running environment, and retain a useful change without repeatedly rebuilding the OS. An unrestricted shared base console also risked concurrent edits, accidental shutdown, and uncertainty about which changes future instances would receive.

The resolution. The GUI opens a controlled base-image session. A per-base lock prevents competing console ownership and coordinates with conflicting image operations. The engineer makes changes in the actual prepared environment and captures them before exiting. Logout or window close resets the generic base to the latest captured image; uncaptured edits are discarded. Session command guards intercept common shutdown/reboot commands so the normal workflow remains capture, then exit.

A successful capture changes the starting point for newly created instances. It does not rewrite the disks of an existing customer instance merely because that instance is stopped and started. This distinction was confirmed during operator testing: restart retained an instance's prior filesystem, while recreation used the updated image.

The tradeoff and lesson. "Sandbox" here describes a controlled editing/reset workflow, not a security boundary that contains an administrator. The command guards prevent common mistakes; they do not make an administrative shell untrusted-code isolation. The useful innovation is an understandable transaction: edit, explicitly capture what should persist, then reset the working base on disconnect. Reproducible build inputs still matter for rebuilding and auditing the image later.

Performance consequence. Engineers iterate on prepared state and publish the useful change, shortening the path from experiment to reusable image.

4. Unattended preparation became a recoverable pipeline

The obstacle. Starting an installer was only the first stage. Media preparation, OS setup, account provisioning, reboots, first-boot tasks, callback delivery, readiness, capture, and image commit each had different completion conditions. An installer exiting or a VM responding did not prove the final image was ready. A late failure could otherwise force the expensive stages to run again or destroy an earlier working image.

The resolution. MYKOS pipelines these stages through OS-specific build workers, persisted build state, scoped callbacks, and explicit readiness and capture checks. Callback tokens have expiry and consumption state; resuming unfinished work re-arms the relevant callback deliberately. macOS's successive boot/provisioning stages require separate handoffs rather than treating the first callback as final success.

Completed VM capture and commit progress are retained. The retry path validates the builder and capture identity, avoids repeating completed embedding or commit work, and publishes the canonical prepared storage after the required commit succeeds. A failed publication step has a rollback path for the previous prepared storage. Transient installer credentials and installation-only state are removed at the appropriate completion boundary.

The lesson. The innovation is reliable handoff and recovery across asynchronous stages, rather than a long shell script that assumes every previous command finished the job. This does not claim unrestricted concurrent builds or interchangeable pipelines for every OS. The success condition is an installed, provisioned, validated image whose supported startup path actually restores and runs.

Performance consequence. Reusable preparation and stage-aware retry avoid repeating expensive completed installation and capture work.

5. Monitoring stayed outside the normal guest execution path

The obstacle. Repeated guest commands for dashboard or API polling add execution work and can interfere with the environment being investigated. Multiple consumers can multiply that cost and disagree about the same instance.

The resolution. Host-side collectors inspect runtime metadata, resource statistics, files, and control sockets, then publish timestamped results for the web application and API. Normal host monitoring avoids executing guest programs during polling. Missing observations remain unknown rather than being converted into healthy values. Explicit terminals and diagnostic commands remain separate workflows.

The lesson. Readiness, inventory, resource limits, and sampled utilization need distinct meanings. A cached observation needs a timestamp. Privileged mutation belongs behind validated helpers rather than being an incidental side effect of a status request.

Performance consequence. Shared host observations reduce repeated guest execution; freshness and collection cost remain measurable constraints.

6. Stable network ownership kept fast-started guests usable

The obstacle. The affected guests had a configured static Podman address and a live interface, yet their actual IPv4 address and default route disappeared later. Checking only Podman metadata made the configuration look healthy.

The cause. A network reload recreated the interface. Guest NetworkManager claimed `eth0`, attempted DHCP, and removed the runtime address after DHCP failed. The failure was delayed, which made an immediate post-change connectivity check inadequate. The triggering reload occurred during an earlier DNS repair; fixing one layer exposed a competing owner in another.

The resolution. A native-only, read-only NetworkManager drop-in marks Podman's interface unmanaged. Podman remains responsible for addressing, whether the selected mode uses private-bridge static IPAM or optional macvlan DHCP supplied by the host proxy. The incident repair restored the existing addresses and gateway from the host network namespace without restarting the affected containers.

Private-bridge allocation also uses deterministic MAC identities, collision checking under the shared lifecycle lock, and address claims retained by stopped containers. These prevent assignment conflicts; the ownership fix prevents the guest from discarding an otherwise valid assignment.

Evidence and lesson. Recorded checks confirmed restored addresses, gateway reachability, DNS resolution, and retention after a delayed recheck with container IDs and start times unchanged. The general rule is one authoritative owner per interface, plus verification of the actual namespace state after asynchronous managers have settled.

7. DNS needed its own tunnel policy

The obstacle. A guest could have a working address and apparently established VPN while name resolution failed. Local resolver addresses, bridge DNS, full-tunnel selectors, and fallback routes do not necessarily lead to the same place.

The resolution. The released native full-tunnel path redirects non-loopback UDP and TCP port-53 requests to `8.8.8.8` and applies the VPN output policy. Shared modes install their corresponding DNS and return-path rules. The DNS decision is explicit instead of relying on whatever resolver happened to be inherited by the image.

That destination is an implementation choice, not a general enterprise DNS architecture. Private split-horizon names require a reachable resolver that knows those zones; reaching a public resolver does not prove customer-domain resolution. Encrypted DNS protocols also fall outside a plain port-53 redirect.

Evidence and lesson. Project records include real namespace DNS queries and guest `getent` checks. The test must cover the resolver path and expected answers while the tunnel is up, after reconnect, and during loss. DNS failures should not be “fixed” by silently allowing protected application traffic onto the ordinary WAN.

8. Shared VPNs needed correct return and failure paths

The obstacle. A native guest, a VM's internal OS, its Podman wrapper, and the host occupy different networking contexts. A tunnel that works for its endpoint does not automatically carry sibling or VM traffic. Source NAT can also change the address that IPsec selectors expect to protect.

The resolution. MYKOS distinguishes independent guest tunnels, a shared native gateway, a shared host gateway, and host-only site mode. Shared native routing pins the selected gateway identity, diverts protected bridge traffic through policy routing, treats DNS deliberately, and keeps the gateway's outer IKE/NAT-T traffic from being diverted back into itself. Host-shared routing preserves the selectors at the host IPsec boundary.

Host participation is conditional and explicit. Managed host full-tunnel routing requires the expected policy and ESP state; management replies, local traffic, and tunnel transport need exceptions. On loss, protected guest traffic remains blocked or unreachable while the managed host internet diversion is removed so ordinary host WAN access can return. Removing a profile deliberately releases its protection and is a different event from an unexpected outage.

Evidence and lesson. Recorded isolated packet tests exercised encrypted forwarding, clear-text rejection, peer/SA loss, and recovery. They do not certify every customer peer configuration. The architecture has to specify who encrypts, which source address the peer sees, how replies return, and what happens when the selected gateway disappears.

9. Live capture had to preserve one consistent moment

The obstacle. Saving RAM and then copying disks after tasks resume can combine different moments. Native checkpointing adds kernel objects, mounts, open descriptors, and runtime protocol compatibility to that problem.

The resolution. The QEMU path copies flushed disks while the source is paused at capture completion. The native path uses a scoped CRIU RPC relay and a post-dump filesystem hook so copying occurs before tasks resume. The relay preserves the vendor RPC exchange and descriptor passing instead of globally replacing CRIU. Native preflight rejects known unsupported state, including open BPF program descriptors.

Startup captures and intentional snapshots have separate lifetimes. Launch captures support reuse; intentional recovery points remain retained until explicitly removed. Image deletion checks references and active use, and failed recovery paths preserve relevant disks for diagnosis.

The limit. A successful checkpoint is not proof of application consistency across every external dependency. Native compatibility is conditional; refusing an unsupported capture is preferable to publishing a recovery point that cannot restore. Restoring and checking the application is the meaningful test.

10. Full OS builds needed a boundary around installer side effects

The obstacle. Installation tools expect machine-like control over services, devices, and runtime directories. During image creation, a `tmpfiles` operation could create `/run/nologin` through a host-bound runtime path and block host logins. Hardware firmware and microcode also added weight that the container workload did not need.

The resolution. Generated kickstart copies exclude those unnecessary packages and exclude both `nologin` paths from the affected `tmpfiles` commands. The operator's original kickstarts remain intact. Native instances receive the `cgroup` visibility needed for correct reporting of their configured 2 GiB limit.

macOS provisioning presented a related lifecycle problem: account creation, home-directory creation, first-boot execution, installation callbacks, and the final capture had to finish in the right order. The recorded successful path used native account tools and explicit verification, with private transient provisioning material removed before capture.

Working images were preserved while failed installation paths were investigated.

The lesson. Installation-time access and artifacts need their own boundaries. A complete userspace is useful, but carrying every installer side effect into the host or reusable image is not.

11. DHCP remained separate from routed VPN traffic

The obstacle. A routed site-to-site VPN does not automatically make remote hosts members of one Ethernet broadcast domain. Initial DHCP discovery and the later routable application traffic are different problems. Forwarding ordinary IP packets through IPsec does not by itself deliver local discovery broadcasts to a remote DHCP server.

How it is handled. The released default keeps allocation local: MYKOS assigns private-bridge addresses. Optional macvlan mode obtains a LAN lease through the host's CNI or Netavark DHCP proxy. Neither mechanism establishes that guests obtain leases from the remote site through the VPN.

A remote-DHCP design needs an explicitly supported relay on the client network, a suitable server scope and relay identity, routes and VPN selectors for the relay exchange, and correct return delivery. A lease also does not establish that its offered address is usable on the client's actual routed subnet. The standard relay model is described in [RFC 2131](#).

Where the boundary remains. Remote DHCP over the MYKOS VPN is not established as a completed, validated feature in the released evidence. Meraki specifically documents that its MX DHCP relay does not support a server behind a non-Meraki VPN peer. That limits that topology; it does not mean every independent relay design is impossible. See [Meraki's relay requirements](#).

The engineering lesson is to choose address ownership and broadcast/relay behavior before troubleshooting the encrypted transport. VPN reachability and remote DHCP support must be tested separately.

Why whole-instance containers are a defensible choice here

The strongest case starts with the performance goal and the workload. A customer system may depend on service supervision, local accounts, numeric ownership, file layouts, printing, shares, scheduled jobs, package versions, and interactions among several processes. For support, migration, and reproduction, those relationships are often the thing being preserved. Splitting them into independently redesigned application containers changes the environment before the investigation begins.

A whole-instance boundary lets the customer ID be the unit of creation, networking, access, capture, and recovery. MYKOS can prepare that integrated unit once and reuse it when the next customer environment is needed. It can preserve a familiar administration model while moving installation and boot work into reusable preparation. Native system containers avoid a separate guest kernel; VM-backed environments supply the OS boundary needed by their workloads. The warm-state optimization comes from coordinated capture and restore, not simply from choosing a larger container.

The common application-container guidance remains useful. Docker recommends separating concerns because independently managed services are easier to scale and reuse; its own guidance treats a single-process container as a rule of thumb, with exceptions. System containers are also an established category: Incus documents full Linux distributions sharing a host kernel. Neither fact proves MYKOS's implementation correct, but they show that one application per container is not a universal prohibition on complete userspaces. Sources: [Docker building guidance](#), [Incus instance types](#).

Objection	Engineering answer for MYKOS
A complete userspace is larger and needs more patching	True. Remove unnecessary build payloads and maintain the OS. Accept the remaining cost where reproducing the integrated environment is the requirement.
Several services lose independent scaling and deployment	True. This model prioritizes an environment's integrity. Independently scalable application components remain better candidates for application containers.
A container is not a VM security boundary	True. Native containers share the kernel, and the released native path is rootful and privileged. Use the trusted administrative host model deliberately; choose a stronger boundary for mutually untrusted workloads.
Warm state can become stale or incompatible	Validate the capture's prerequisites and readiness, refresh it after relevant changes, and account for external dependencies. Do not assume a restored process has a valid remote session.
Stateful recovery is harder than replacing an application image	Preserve private storage and intentional snapshots, coordinate data protection, and test actual restoration. An image update is not a customer-data migration.
A large container is automatically an antipattern	Judge the unit against the requirement. Reproducing a multi-service customer system is a different requirement from deploying one independently scalable service.

The performance claim is about less repeated preparation and less operational interference, not inherently faster application instructions. The decision is conditional and testable: whole-instance environments are justified when preserving the integrated system provides more value than independent service deployment. They are not a reason to package every new application this way.

Planned milestone: continuous compliance status and alerts

Automated security compliance scans are planned. The intended workflow is to scan against the selected compliance policy, show pass/fail status, and alert when a previously passing system falls out of compliance. This is future work requested by the project owner, not a capability validated in the released baseline.

The engineering chapter will cover policy selection, scan scheduling and cost, meaningful status freshness, and regression alerts after we implement and test those parts together. The performance goal carries into that work: measure scan overhead and schedule it deliberately so assurance does not undermine the responsiveness being protected. Existing authentication and isolation mechanisms do not stand in for that future assessment.

Measuring the performance goal

Use application-ready time as the main startup endpoint: the required customer service works with its data, identity, and connectivity. Keep container/wrapper launch, OS restoration, and actual application readiness as separate timestamps.

Question	Measurement that can answer it
How much boot work does warm restore avoid?	Compare cold boot, persisted warm restore, and access to an already-running instance on the same OS, RAM, storage, and host.
How quickly does a customer backup become useful?	Start with an available compatible backup; time reconciliation, mounts, startup, and the first successful customer-service check. Report transfer/preparation separately.
Does the dashboard affect the workload?	Compare application latency/throughput with normal observation enabled and disabled; record collection CPU, I/O, frequency, and data freshness.
What happens as instance count grows?	Run concurrent starts and steady workloads; report throughput, readiness failures, resource pressure, and median and tail latency across repeated runs.
Is the faster path dependable?	Recheck addresses after interface recreation and delayed manager activity; test DNS/VPN recovery and restore real snapshots with application-data checks.

The current evidence supports the mechanisms and operator-observed responsiveness. These measurements would turn those observations into repeatable performance results. No general speedup factor, density limit, or compliance-scanning overhead has been established here.

Evidence and implementation map

This account uses the released source, recorded incident/validation notes, and the preserved operator notes. Historical intent is not treated as proof of current implementation. No engineering-hour totals or general startup benchmark are claimed.

Topic	Principal implementation or evidence
Warm state and paired storage	mykos_vm.py: capture_live, restore_manifest, wait_for_restore; operator warm-start notes
Stable guest addresses	mykos_network.py, mykos-podman-nm.conf; recorded live IP-loss repair and delayed checks
DNS and shared VPNs	pos/vpn-updown, mykos_vpn_gateway.py, mykos_vpn_host.py; isolated packet-test records
GUI editing lifecycle	mykos_base_session.py, image/base console handlers; recorded capture-before-logout and reset tests
Backup composition	mykos_backup_identity.py, mykos_backup_restore.py, mykos-stage-pos
Consistent snapshots	mykos_snapshots.py, mykos_criu.py; compatibility preflight and post-dump hook
Installation pipeline	mykos_image_build.py; mykos_vm.py pending-capture and commit paths; OS bootstrap callbacks
Image build boundaries	Generated kickstart preparation; macOS account/bootstrap scripts and recorded clone checks
Observation	mykos_host_monitoring.py and timestamped collector/API state
Remote DHCP	Released local IPAM/proxy modes; RFC 2131 and Meraki relay documentation; remote relay not validated