

# Sandboxing AI Tools

*Containers, MicroVMs, and Permission Scopes for  
Agents*

Trex Team

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Published by NobleTrex Press



© 2026 by NOBTREX LLC. All rights reserved.

For permissions and other inquiries, write to:  
P.O. Box 3132, Framingham, MA 01701, USA

This book is for educational and informational purposes only. The author and publisher make no guarantees about the accuracy or completeness of its contents and accept no liability for any loss or damage resulting from its use. Software tools such as large language models have been applied to assist in the writing and editing of this book. All product names, logos, and trademarks are the property of their respective owners. References to third-party products or names are for identification only and do not imply endorsement.

# Contents

|          |                                                                                                    |           |
|----------|----------------------------------------------------------------------------------------------------|-----------|
| <b>1</b> | <b>Security Foundations for Tool-Using Agents</b>                                                  | <b>5</b>  |
| 1.1      | Threat Model for Agent-Invoked Tool Execution . . . . .                                            | 6         |
| 1.2      | Isolation, Authorization, and Auditing as Separate Controls . . . . .                              | 14        |
| 1.3      | Least Privilege as a System Constraint (Default-Deny Everywhere) . . . . .                         | 22        |
| 1.4      | Defining Assets and Data Boundaries for AI Tool Runs                                               | 29        |
| <b>2</b> | <b>Designing Permission Scopes for Agents</b>                                                      | <b>37</b> |
| 2.1      | Modeling Permission Scope Across Filesystem, Network, Compute, Identity, and Device Boundaries . . | 37        |
| 2.2      | Designing Temporal Scope with TTLs, Just-in-Time Grants, and Revocation Boundaries . . . . .       | 47        |
| 2.3      | Choosing Between Per-Invocation, Per-Agent, and Per-Workflow Permission Models . . . . .           | 56        |
| 2.4      | Specifying, Reviewing, and Testing Scope Policy as an Enforceable Artifact . . . . .               | 64        |

|          |                                                                                |            |
|----------|--------------------------------------------------------------------------------|------------|
| <b>3</b> | <b>Linux Isolation Primitives as the Building Blocks</b>                       | <b>73</b>  |
| 3.1      | Namespaces and the Anatomy of Process Containment                              | 74         |
| 3.2      | cgroups v2 for Resource Scoping and Abuse Resistance                           | 83         |
| 3.3      | Designing Filesystem Views for Untrusted Tool Workloads                        | 91         |
| 3.4      | Shared-Kernel Risk Boundaries and What These Primitives Cannot Solve . . . . . | 100        |
| <b>4</b> | <b>Containers as a Sandbox Boundary (and How to Harden Them)</b>               | <b>109</b> |
| 4.1      | OCI Container Boundary Properties and Common Escape Classes . . . . .          | 110        |
| 4.2      | Configuring Hardened Defaults for Container Tool Runs                          | 119        |
| 4.3      | Choosing Rootless Containers for Agent Tool Runners .                          | 128        |
| 4.4      | Modeling the Container Runtime Stack as Part of the TCB                        | 136        |
| 4.5      | Operating Deterministic Denies and Safe Failure Paths                          | 144        |
| <b>5</b> | <b>Kernel Hardening Inside and Around the Sandbox</b>                          | <b>153</b> |
| 5.1      | Syscall Allowlisting with seccomp: From Generic to Workload-Specific . . . . . | 153        |
| 5.2      | Failure Handling and Observability for seccomp Denials                         | 163        |
| 5.3      | SELinux/AppArmor: MAC as a Second Line of Defense                              | 171        |
| 5.4      | Landlock for Unprivileged Self-Confinement . . . . .                           | 181        |
| <b>6</b> | <b>MicroVMs for Stronger Isolation and Multi-Tenant Safety</b>                 | <b>191</b> |

|          |                                                                                           |            |
|----------|-------------------------------------------------------------------------------------------|------------|
| 6.1      | Why VM Boundaries Matter for Untrusted Tools . . . .                                      | 192        |
| 6.2      | Designing MicroVM Lifecycles for Fast Boot and Ephemeral Tool Runs . . . . .              | 201        |
| 6.3      | Hardening the Firecracker Execution Model: Jailer, API Surface, and Guest Trust . . . . . | 209        |
| 6.4      | Minimizing Guest/Host Interfaces and Mediating I/O by Scope . . . . .                     | 216        |
| <b>7</b> | <b>Composing Layers and Building the Tool Runner Control Plane</b>                        | <b>225</b> |
| 7.1      | Choosing Defense-in-Depth Compositions for Mixed-Risk Tool Fleets . . . . .               | 226        |
| 7.2      | Designing the Tool Runner Control Plane for Policy, Launch, and Mediation . . . . .       | 235        |
| 7.3      | Implementing Default-Deny Egress with Proxies, DNS Policy, and Metadata Shields . . . . . | 244        |
| 7.4      | Issuing Ephemeral Workload Identity and Delivering Secrets Per Invocation . . . . .       | 254        |
| 7.5      | Operating Audit and Forensics Pipelines Without Turning Logs into a Data Leak . . . . .   | 262        |
| 7.6      | Enforcing Supply Chain and Cluster Policy at Admission Time . . . . .                     | 270        |

# Introduction

Artificial intelligence agents extend their computational utility by interacting with external systems through tool execution. This capability, while necessary for resolving complex tasks, fundamentally alters the security posture of the application. When non-deterministic models dictate the execution of generated code, file system operations, or network requests, systems become highly vulnerable to prompt injection, malicious payloads, and unauthorized lateral movement. Addressing these critical vulnerabilities requires rigorous containment strategies that assume the model, the inputs it processes, and the tool outputs it generates are inherently untrusted.

This book provides a systematic, engineering-focused approach to securing agent-invoked tool execution. Effective security in this domain relies on disentangling three distinct control layers: isolation, authorization, and auditing. By establishing a precise threat model, engineers can design architectures where least privilege operates as a hard system invariant rather than a post-deployment configuration preference. This necessitates defining explicit data boundaries, identifying high-value assets, and translating threat vectors into enforceable permission scopes. These scopes must dictate constraints across multiple dimensions-including compute resources, network access, workload identity, and device interfaces-and operate within strict temporal

limits using short-lived execution windows and immediate revocation protocols.

Enforcing these scopes requires a precise application of operating system primitives. The text details the usage of Linux namespaces, cgroups v2, and filesystem isolation patterns that form the technical basis for process containment. Containers serve as the baseline execution boundary for most tool workloads, but standard container configurations are insufficient against targeted evasion attempts. By applying stringent hardening techniques-such as dropping Linux capabilities, mandating read-only root filesystems, and utilizing rootless execution-practitioners can mitigate extensive risk. Furthermore, mandatory access controls like SELinux and AppArmor, combined with seccomp syscall filtering and Landlock for unprivileged self-confinement, restrict processes well beyond standard discretionary permissions to minimize the shared-kernel attack surface.

For environments requiring multi-tenant safety or where shared-kernel risks are unacceptable, microVMs provide a hardware-backed isolation boundary. Technologies such as Firecracker isolate untrusted tool execution within dedicated guest kernels, fundamentally changing the risk profile of kernel exploits. The text addresses the specific design constraints of microVMs for AI agents, focusing on fast boot times, minimal device exposure, proxy-mediated network egress, and ephemeral storage. This ensures that strict isolation does not degrade the latency or efficiency of the underlying AI application.

Ultimately, these isolation primitives must be integrated into a cohesive tool runner control plane. This involves deploying automated brokers that validate tool requests, evaluate declarative policies, issue just-in-time ephemeral credentials, and safely mediate inputs and outputs. It also requires comprehensive supply chain verification, deterministic failure handling, and high-fidelity observability. Telemetry must be designed to generate actionable audit trails-recording authorization de-

cisions, blocked syscalls, and resource limit breaches-without logging sensitive prompt data or exposing downstream artifacts.

By synthesizing policy specification, container hardening, kernel restriction, and hardware-assisted virtualization, this text offers a definitive reference for AI security architecture. It equips system architects and security engineers with the technical methodologies required to safely deploy agentic AI systems in production environments, ensuring that tool execution remains deterministic, securely confined, and fully auditable.



# Chapter 1

## Security Foundations for Tool-Using Agents

*Agent security failures rarely begin with an exotic kernel exploit; they usually begin with a system that never decided what the model was allowed to touch, under whose authority, and with what evidence afterward. This chapter frames tool execution as a security architecture problem: first model the adversary and the run boundary, then separate containment from permission and proof, then make least privilege the default shape of the platform, and finally define the data boundaries that later chapters will enforce with concrete mechanisms.*

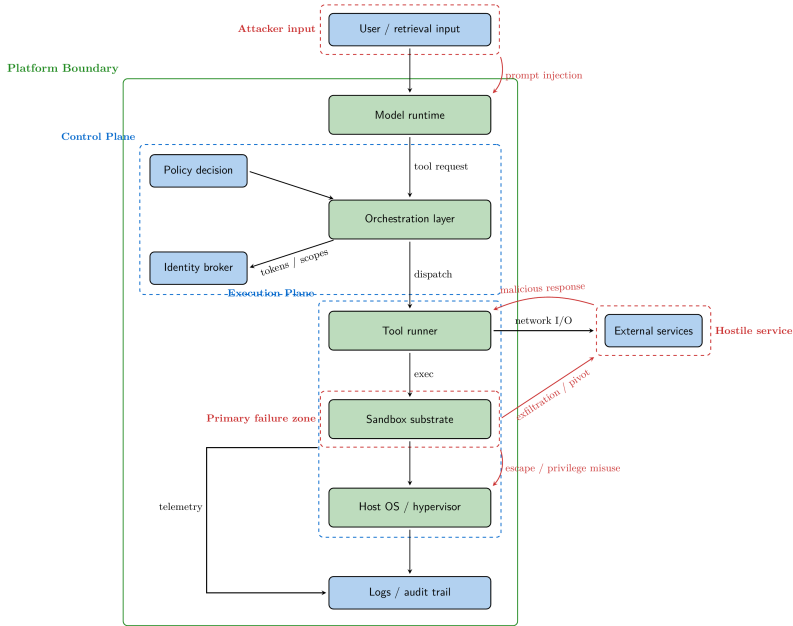
## **1.1. Threat Model for Agent-Invoked Tool Execution**

A tool call is not a textual act; it is a privilege transition. The moment an agent turns generated text into filesystem access, network traffic, credential use, or API mutation, you have crossed from language generation into distributed execution with real consequences. That boundary matters because the model itself may be perfectly coherent while the surrounding system is still vulnerable to adversarial input, mis-scoped authority, or unintended persistence. A useful threat model starts by naming the components that participate in that transition and the trust each one must place in the others.

### **The execution path and its boundaries**

A practical tool invocation usually moves through a chain like this: the model runtime produces a proposed action, the orchestration layer selects a tool, the tool runner mediates the request, the sandbox executes the work, and the host or hypervisor enforces the outer boundary. Identity and policy are often handled by a separate broker, while logs and telemetry capture evidence of what happened. External services sit beyond the platform boundary and may receive data, credentials, or side effects. Each hop creates a distinct trust boundary, and each boundary can fail independently.

## 1.1. THREAT MODEL FOR AGENT-INVOKED TOOL EXECUTION



Failure modes: credential abuse, data exfiltration, lateral movement, persistence, and denial of service can arise at different layers.

The diagram is easiest to reason about if you separate control from execution. The model proposes; the runner decides whether a request may proceed; the sandbox performs the work; the host or hypervisor constrains what the sandbox can touch; identity services issue whatever credentials are needed; logs preserve a trace. None of those elements alone defines the system's security posture. A secure platform composes them so that compromise of one layer does not automatically grant durable control over the others.

### Why the dangerous moment is the translation from text to action

Many agent failures begin with a mismatch between semantic trust

and operational trust. The model may read a web page, retrieve a document, or inspect an uploaded file and then emit a plausible action that actually encodes attacker intent. If the orchestration layer treats that output as trustworthy because it came from the model, the system has confused prediction with authorization. The same mistake appears when a tool's output is fed back into later model context without normalization: the tool becomes an instruction channel, not just a data source.

That asymmetry is what makes tool-using agents different from ordinary applications. An attacker does not always need classical code execution. If the platform lets a model-mediated request reach a privileged API, the attacker can obtain effects that look like legitimate operation from the system's perspective. Prompt injection, malicious documents, hostile web content, and poisoned retrieval corpora exploit that property by steering the model toward actions that are syntactically valid and semantically harmful.

### **Attacker positions and the paths they control**

A useful threat model distinguishes attacker position from attacker goal. An attacker who controls input to the model is not the same as an attacker who controls the execution environment, and each requires different containment assumptions.

*Input-side attackers* influence what the model sees. They may place adversarial text into retrieved pages, uploaded files, support tickets, code comments, package metadata, or API responses. Their main objective is to induce unsafe tool selection, data disclosure, policy confusion, or hidden instruction execution. The platform cannot assume that any content supplied to the model is benign just because it arrived through a normal business flow.

*Environment-side attackers* influence where the tool runs. They may compromise a dependency inside the sandbox, exploit a kernel or run-

time flaw, abuse a writable mount, or interfere with shared infrastructure in a multi-tenant platform. Their objective is usually broader: escape the sandbox, steal credentials, pivot to the host, or reach neighboring workloads.

*Control-plane attackers* target the orchestration and identity layers. They may compromise a policy service, intercept an overly broad token, abuse an operator account, or manipulate the broker that issues scopes and secrets. In an agent system, this class is especially dangerous because a control-plane compromise can convert otherwise well-contained executions into repeated, policy-sanctioned misuse.

### **Failure modes that matter operationally**

A threat model becomes useful when it names the damage categories the platform must bound. The important failure modes are rarely limited to code execution.

*Unauthorized data access* includes reading files, secrets, prompts, cached objects, or service responses outside the intended task. This happens when the sandbox sees too much of the workspace, when credentials are reused across tasks, or when a tool can enumerate metadata that was never meant to be exposed.

*Exfiltration* turns read access into outbound movement. Data may leave through network calls, uploaded artifacts, logs, side channels, or even model transcripts that get preserved in downstream systems. A boundary that prevents file reads but permits arbitrary egress still allows sensitive material to leave the environment.

*Lateral movement* occurs when the tool reaches adjacent services, shared storage, internal APIs, or neighboring tenants. In practice, this often follows credential theft or overbroad network reachability. The platform should assume that any principal with internal connectivity may be used to probe for additional trust relationships.

*Persistence* means the attack survives the current invocation. A malicious tool may write startup state, scheduled jobs, cache entries, or poisoned outputs that influence later runs. Persistence is especially important in agent systems because a single compromised run can become a standing influence on future reasoning.

*Integrity destruction* covers modification of source files, configuration, artifacts, or outputs in ways that corrupt later work. A tool run that silently edits a prompt template, model memory store, or dependency lockfile can cause damage long after the original request completes.

*Covert influence on downstream reasoning* is the special agent-specific failure mode. A tool can return structured text, logs, or generated artifacts that appear innocent but are crafted to alter later model behavior. That is not merely data corruption; it is an attack on the decision process itself.

*Resource abuse* includes denial of service, runaway cost, and cryptomining. Tool execution often has access to compute, network, or storage that can be consumed even when confidentiality is not the immediate target. A sound threat model counts those resources as assets too.

### **Containment questions that drive design**

Each failure mode should map to a small set of containment questions:

What assets were reachable from this run? What identities could the run use? What external destinations could it contact? What writable state could survive teardown? What evidence would remain after the action finished?

Those questions force specificity. If the answer to the first question is “the entire workspace,” the threat surface is already too large. If the answer to the second is “the agent’s normal service account,” the platform has probably conflated model identity with operator identity. If the answer to the fourth is “anything on disk,” then a transient ac-

tion can become a persistent compromise. If the answer to the fifth is “whatever the process happened to print,” then incident response will be guessing.

### **Security invariants for agent tool execution**

The following invariants are the architectural obligations that later controls must satisfy.

**The model output is not authorization.** The model can propose an action, but the platform must decide whether the action is permitted for the current principal, scope, and resource state. This follows the zero-trust principle that no request inherits trust merely because it came from inside the system.

**Tool inputs are adversarial by default.** Retrieved text, files, API responses, and logs can all contain attacker-controlled content. The platform should treat them as untrusted until they are validated, typed, or transformed into a safer form.

**The sandbox is assumed breachable at some layer.** If a threat model assumes perfect containment, it will overfit to one mechanism and miss the composition problem. You should expect that one boundary may fail and design the next boundary to limit blast radius.

**External services must not infer authority from network location alone.** Internal IP space, service mesh membership, or co-residency are not substitutes for explicit authorization. A tool should not gain permission just because it runs in the same cluster or subnet as a target.

**Every run needs a bounded blast radius.** A single invocation should have finite reach in time, privilege, data scope, and side effects. Even a prompt-induced misuse should be unable to accumulate open-ended authority.

## Mapping attackers to controls without collapsing the model

The point of the threat model is not to enumerate every exploit family. It is to assign control ownership. Input-side attacks push you toward prompt and content mediation, output gating, and task-specific scopes. Environment-side attacks push you toward stronger execution isolation, dependency control, and host hardening. Control-plane attacks push you toward explicit identity separation, short-lived credentials, and auditable policy decisions.

That separation matters because systems fail when designers collapse all risks into one generic “sandbox” label. A container may reduce filesystem reach but still hold a powerful token. A narrow IAM policy may prevent accidental cloud mutations but still permit arbitrary local parsing of hostile files. A robust audit trail may prove what happened after the fact, yet still leave the platform unable to stop the action in time. Those are different failures, and they demand different mitigations.

## A practical taxonomy of principals and authorities

At minimum, you should name the following actors:

- The *end user* requests work and may be honest, confused, or malicious.
- The *model runtime* interprets prompts and generates candidate actions.
- The *orchestration service* selects tools and enforces policy.
- The *tool runner* turns approved intent into executable work.
- The *sandbox substrate* isolates the process and mediates resource access.
- The *identity broker* issues credentials or scoped tokens.

- The *external service* consumes requests and returns data or side effects.
- The *platform operator* owns the infrastructure and can be either a defender or a high-value insider threat.

Once these roles are distinct, you can ask which one is trusted for what. The model is trusted to generate plausible plans, not permissions. The orchestration layer is trusted to enforce scope, not to be a hidden root account. The sandbox is trusted to limit reach, not to decide business policy. The identity broker is trusted to mint narrow credentials, not to infer the user's intent. The external service is trusted only to the extent that its responses are validated before reuse.

### **Attackers in multi-tenant and shared-infrastructure settings**

Shared infrastructure sharpens every mistake. If two tenants share a runtime, a host kernel, a storage backend, or a control-plane component, a failure in one tenant can become a cross-tenant incident. In that setting, the distinction between containment and trust is critical. You should not model tenant separation as a promise that the platform will never be probed; you should model it as a requirement that any probe hits a boundary before it reaches another tenant's data or credentials.

This is also where blast radius becomes measurable. A less-isolated execution boundary may be acceptable for low-risk, single-tenant, internally controlled tasks. The same boundary is harder to justify when the platform executes user-supplied code, retrieves untrusted content, or processes secrets on behalf of multiple customers. The boundary choice is therefore a security decision, not merely an infrastructure preference.

### **What the model should never be allowed to assume**

An agent platform becomes fragile when it lets one layer infer too much from another. The model should not assume that because a tool was

offered, it is safe. The runner should not assume that because a request was well-formed, it is harmless. The sandbox should not assume that because a process is isolated, its outputs are clean. The external service should not assume that because a request came from an internal IP range, the requester is entitled to all available data. The operator should not assume that because logs exist, the system was controlled.

Those assumptions are common because they make the system feel simpler than it is. The real system is a sequence of mediated accesses, each with its own attack surface and each with its own failure semantics. Treating the model as a security principal, treating prompt filtering as containment, or collapsing all tools into one trust zone hides those semantics and produces designs that look disciplined but fail under adversarial input.

The useful mental model is stricter: the agent is a coordinator of privileged operations, not a privileged identity itself. Every tool run is a fresh access request against concrete assets, with a scope, a boundary, and a trail of evidence.

## **1.2. Isolation, Authorization, and Auditing as Separate Controls**

A workload can fail securely in one dimension and still fail badly in another. You can isolate a tool from the host filesystem and still let it mutate production data through a valid token. You can authorize a request precisely and still run it in an environment that leaks neighboring memory. You can do both and still have no durable proof of what happened. The recurring mistake is to treat these guarantees as interchangeable because they often appear together in the same deployment diagram. They are not the same control, they do not fail in the same way, and they do not belong to the same enforcement point.

### A small example that separates the three guarantees

Consider a file-processing tool that needs to read one input directory, write a report, and upload the result to an object store. A minimal secure setup might look like this:

```
# Run with one read-only input and one writable scratch path.
docker run --rm \
  --read-only \
  -v "$PWD/in:/in:ro" \
  -v "$PWD/out:/out:rw" \
  --network=none \
  my-tool:latest \
  /bin/process --input /in --output /out/report.json
```

That command expresses *isolation* by constraining visible paths and network reachability. It does *not* answer whether the process is allowed to upload a specific report to a specific bucket; that is an *authorization* question. It also does not guarantee that you can later prove which input file produced which output, or whether the upload occurred at all; that is an *auditing* question. If you conflate those concerns, you end up debugging the wrong layer.

### Isolation answers reachability, not permission

Isolation asks a narrow question: what can the code technically reach or interfere with while it runs? The answer comes from execution boundaries, not business policy. Namespaces, mount layouts, cgroups, syscall filters, mandatory access controls, and VM boundaries all reduce the reachable surface of a process. They can stop a tool from opening host files, spawning arbitrary child processes, or touching kernel interfaces that were never intended for it.

That makes isolation a containment property. It limits damage from compromise and reduces the ambient attack surface of the runtime. It does not decide whether the task itself is legitimate. A process can be perfectly isolated and still misuse whatever authority it has inside the boundary. If the sandbox contains a credential, the sandbox has not

solved authorization; it has simply narrowed the place where abuse can occur.

The distinction matters because many operational teams describe a workload as “sandboxed” when they really mean only that it cannot see the host filesystem. That is a partial statement, not a security conclusion. A sandbox can still expose shared memory, network paths, mounted secrets, or writable caches. Isolation is therefore necessary, but it only answers *where* a process can act, not *whether* it should act.

### **Authorization answers who may do what, and under which identity**

Authorization asks a different question: should this principal or workload identity be allowed to perform this action on this resource now? The answer comes from policy, identity, and scope. A policy engine may allow one tool to read from one repository and deny another; an identity broker may mint a token that expires after one invocation; a service-side ACL may permit writes only to a single prefix. Those are authorization decisions because they bind an action to a named resource under a named identity.

This is why a container runtime cannot substitute for policy. A container may constrain kernel-level behavior, but it cannot know that `repo-A` is acceptable while `repo-B` is forbidden unless you encode that decision elsewhere. Likewise, an IAM token can grant access to an external API without reducing local kernel exposure. Authorization and isolation operate on different axes. One governs legitimacy of action, the other governs technical reach.

A useful rule is to treat authorization as a question of *named authority*. If you cannot name the resource, the principal, the operation, and the duration of the grant, you do not yet have an authorization design. You have only a vague promise that someone will be careful.

## **Auditing answers what can later be proven**

Auditing asks a third question: what evidence exists afterward? A useful audit trail records the decision, the action, the identity, the target, the timestamp, and enough run metadata to correlate events across layers. That evidence can support incident response, compliance review, abuse detection, and postmortem analysis. Without it, you may have containment and authorization, but you will not know whether the system followed them.

Auditing is not the same as logging everything. Raw logs can be incomplete, uncorrelated, or too sensitive to preserve safely. Good auditing produces attributable records, stable identifiers, and retention rules that support later reconstruction. A deny event is especially valuable because it tells you that policy was enforced rather than merely hoped for. If a decision is made in one service and execution happens in another, you need a correlation key so the two can be joined later. Otherwise the evidence fragments across components and becomes operationally weak.

## **Why the controls fail differently**

The value of separating these controls becomes obvious when one fails. Strong isolation with weak authorization gives you a well-contained way to execute the wrong task. Strong authorization with weak isolation still lets a process steal neighboring secrets or probe a vulnerable kernel. Rich auditing with broad privileges gives you a very detailed record of a breach that should not have been possible in the first place.

Those failure compositions are common because each control protects a different security objective:

|                      |                                                       |
|----------------------|-------------------------------------------------------|
| <b>Isolation</b>     | Constrains reachable surfaces and interference paths. |
| <b>Authorization</b> | Constrains permitted actions on named resources.      |
| <b>Auditing</b>      | Constrains what remains provable after the action.    |

A design that blurs those objectives tends to misplace trust. For ex-

ample, teams often try to use network segmentation as authorization, but segmentation only says whether traffic can flow. It does not say whether the request is allowed for this task, under this identity, at this time. Likewise, human approval is not a substitute for machine-enforced scope. A reviewer may understand the intended task, but the runtime still needs a policy decision it can enforce.

### **Where each control belongs in the system**

The cleanest way to avoid confusion is to assign each control to its enforcement locus.

*Isolation* belongs to execution boundaries. Containers, namespaces, mount layouts, cgroups, seccomp filters, MAC policies, and VM boundaries shape what the process can technically touch. These mechanisms are about reachability and interference. They are the subject of the mechanism chapters later in the book, so the only point here is placement: they are substrate controls.

*Authorization* belongs to policy engines and identity systems. Policy decision points, identity brokers, scoped tokens, service ACLs, export gates, and network mediators decide whether a requested operation should proceed. These controls answer the business question of who may do what. They do not create a sandbox by themselves.

*Auditing* belongs to event generation and evidence retention. Correlation IDs, immutable run metadata, explicit allow and deny records, provenance tags, and durable log pipelines preserve what happened in a way that can be reviewed later. Auditing can explain a decision, but only if the decision was actually recorded.

That mapping is operationally important because a control cannot be substituted without changing its meaning. An IAM token does not eliminate local attack surface. A container does not encode business intent. A log entry does not block an action. Once you separate the

responsibilities, the rest of the architecture becomes easier to reason about.

### **A practical failure mode: the right sandbox, the wrong scope**

A common failure pattern is to harden the execution boundary while leaving the tool over-authorized. Imagine a code-analysis agent that runs inside a minimal container with no host access, no root privileges, and a restricted filesystem. If that tool still receives a token that can read every repository in the organization, the system has only moved the blast radius from the host to the data plane. The sandbox did its job, but the authority model did not.

This is why permission shaping belongs in a different chapter and a different control plane. If the tool's identity is broad, the best sandbox in the world merely localizes the damage. The same logic applies in reverse: a narrowly scoped token does not protect you if the runtime can inspect unrelated memory, reach host paths, or escape into a neighboring namespace. You need both boundaries, and you need them for different reasons.

### **Auditing is only useful when identity and decision points correlate**

Auditability often breaks at the seams between services. A tool runner may emit a log line saying that a request was made, while an identity broker records that a token was issued, and an external service records that an object was modified. If those events cannot be tied together with a common invocation identifier, the record is mostly anecdotal.

Good audit design therefore starts with attribution. You want to know which run, which principal, which policy decision, and which resource are associated with each action. That requires more than a timestamp. It requires stable identifiers, consistent schemas, and retention that survives the life of the investigation. In regulated or high-assurance

settings, the ability to prove a deny is as important as the ability to prove an allow. A system that silently rejects actions without recording the reason is hard to debug and harder to defend.

At the same time, auditing can become a sensitive-data sink if you collect indiscriminately. Full prompts, credentials, or raw payloads may be too sensitive to preserve in plain logs. The right answer is selective evidence: enough structure to reconstruct the decision and action, but not so much raw material that the audit trail becomes a second exposure surface. That trade-off is one reason the controls must remain separate. Audit design should not be reduced to “log more.”

### **Mechanism-specific details vary, but the security semantics do not**

The concrete enforcement tools change across kernels, container runtimes, cloud platforms, and observability stacks. Seccomp can filter syscalls but cannot express “may read repository A but not repository B.” Landlock can narrow filesystem-style access from user space, but its effective coverage depends on the running ABI and on which object classes are currently restrictable. A VM boundary offers a stronger substrate isolation story than a shared-kernel container, but it still does not decide whether the workload is allowed to touch a particular external dataset. Each mechanism has a specific envelope, and its version-dependent behavior belongs in the mechanism chapters.

That variability does not weaken the conceptual model. The semantics remain stable: isolation limits technical reach, authorization limits permitted action, and auditing preserves evidence. The implementation details differ, but the control objective does not. That is the reason to keep the concepts separate before introducing any particular technology.

### **Decision criteria for layering the controls**

When you choose how much strength to put into each control, ask what guarantee you are buying.

If the main risk is accidental overreach into the host or neighboring workloads, increase isolation. If the main risk is a legitimate tool using the wrong data or service, tighten authorization. If the main risk is inability to investigate or prove what occurred, strengthen auditing. In most real systems, the answer is not one control alone but a layered combination with different tuning at each boundary.

The trade-offs are concrete. Stronger auditing raises storage, privacy, and operational overhead. Finer-grained authorization increases policy complexity and often adds latency or more failure modes in the control plane. Stronger isolation can reduce ergonomics, limit compatibility, or increase resource cost. You should accept those costs deliberately rather than smuggling them into a single vague claim that the system is “sandboxed.”

### **Anti-patterns that collapse the model**

Several design mistakes recur often enough to deserve explicit names.

- **Using network placement as authorization.** Internal origin does not imply legitimacy. A request from inside the cluster can still be malicious or over-broad.
- **Treating prompt filtering as containment.** Sanitizing input text does not create a boundary around the execution environment. The model can still be steered into legitimate-looking misuse.
- **Assuming logs are complete because they exist.** If events are not correlated, attributable, and retained appropriately, they are not reliable audit evidence.
- **Assuming a container is both a sandbox and a policy**

**engine.** It is neither a permission oracle nor a substitute for identity-aware access control.

- **Using human review as runtime enforcement.** Review can complement machine controls, but it cannot scale as the only guard on repeated tool execution.

These anti-patterns matter because they produce systems that appear layered while actually concentrating all trust in one place. That is the opposite of defense in depth.

### **The right mental model for the rest of the platform**

The practical payoff of separating the controls is simple. Once you know which question belongs to which layer, you can place the right guardrail in the right place. Isolation bounds the runtime. Authorization bounds the action. Auditing bounds the amount of uncertainty left afterward. Together they make the system understandable under attack, but only if you resist the temptation to let one substitute for the others.

A tool run is therefore not a single event with a single protection. It is a sequence of accesses, each with its own reachable surface, its own permission check, and its own evidence trail. When you design that sequence explicitly, the later choices about scopes, container hardening, and VM-grade containment become much easier to evaluate.

## **1.3. Least Privilege as a System Constraint (Default-Deny Everywhere)**

A secure tool platform does not begin by asking how much access a tool *needs*; it begins by asking how little authority the platform can survive with. That inversion changes the entire design. If you start permissive

and try to tighten later, the permissive path becomes the production contract, exceptions pile up, and the platform quietly normalizes ambient authority. If you start with no durable authority and add only the minimum required per invocation, least privilege becomes structural rather than aspirational.

### **A concrete baseline**

A minimal file-analysis tool can be launched with no inherited network, no write access to the host filesystem, and only one read-only input mount plus one writable scratch directory:

```
docker run --rm \
  --read-only \
  --network=none \
  -v "$PWD/input:/input:ro" \
  -v "$PWD/scratch:/scratch:rw" \
  --cap-drop=ALL \
  my-tool:latest \
  /bin/analyze \
  --input /input \
  --out /scratch/report.json
```

That command is not a complete security model, but it demonstrates the platform stance. Nothing is ambient. Nothing is assumed. Every privilege has to be named, bounded, and justified for the invocation at hand. The same posture should hold for identity, network egress, device access, and resource consumption, even when the mechanism changes later in the stack.

### **Filesystem scope starts with no inherited surface**

The filesystem is usually the first place least privilege fails, because it is easy to hand a tool an entire workspace and call that convenience. The safe baseline is narrower: no host paths by default, minimal read-only inputs, explicit writable scratch space, and controlled export points. If a tool only needs one repository subdirectory, do not mount the parent directory. If it needs to emit one artifact, do not give it a shared working

tree that survives across runs. If it needs temporary state, isolate that state so that teardown is predictable and residual data do not leak into the next invocation.

The important distinction is between *visibility* and *durability*. A tool may need to read a source tree but not modify it. It may need to write a report but not persist caches. It may need access to a generated artifact but not to the full workspace history. Treat each path as a grant with a purpose and a lifetime. That discipline makes it easier to reason about what can be exfiltrated, what can be corrupted, and what can survive after the process exits.

### **Network scope is not the same as connectivity**

Default-deny at the network layer means no unrestricted outbound access and no trust in local network position. A tool should not be able to assume that because it is running inside an internal cluster it may contact any internal service, resolve any hostname, or follow arbitrary redirects. DNS itself is a policy surface. So is destination selection, proxy configuration, and egress from the runtime to the broader internet.

The baseline therefore starts with no network unless a task explicitly requires it. If a tool must reach an API, narrow the destination list and the protocol family. If it must call out to more than one service, make those services explicit rather than inheriting ambient route access. If it must fetch packages or data, consider whether that fetch belongs in the same invocation at all. Network privilege is especially dangerous in agent systems because exfiltration does not need to look like obvious theft; it can be disguised as normal telemetry, package installation, or status reporting.

### **Identity must be invocation-bounded**

Least privilege fails quickly when a tool inherits static credentials or a

broad service account from the hosting environment. The safe baseline is no baked-in secrets, no shared long-lived agent token, and no authority that outlives the invocation unless the design explicitly requires it. The credential used by a tool should match the task, the resource, and the time window, not the host on which it happens to run.

This is where the distinction between ambient authority and requested authority becomes concrete. Ambient authority exists because the process was launched in a privileged context. Requested authority exists because the runtime obtained an explicit grant for a specific action. The latter is the only form that scales cleanly in agent systems. If a process can act simply because it shares a node with a credential cache, the boundary has already failed even if the application code appears careful.

#### **Device and kernel-adjacent access deserve the same treatment**

The same default-deny rule applies to devices, privileged interfaces, and kernel-adjacent surfaces. A tool should not run in privileged mode. It should not see arbitrary `/dev` nodes. It should not acquire capabilities that let it mutate host state indirectly through interfaces the tool author never modeled as security-relevant. Even non-root processes can retain significant privilege if capability sets remain broad, so “run as non-root” is necessary but incomplete.

This matters most when the execution boundary is shared-kernel. The runtime may be isolated enough for normal operations, yet a device file, a control socket, or a retained capability can reopen the host boundary. The platform should therefore treat kernel-facing privileges as explicit grants, not as a side effect of choosing a convenient container image.

#### **Resource limits are part of privilege**

Least privilege is not only about confidentiality and integrity. CPU,

memory, process count, file descriptors, and wall-clock time are resources that can be abused as surely as data can. A tool that can consume unbounded compute may not steal secrets, but it can still cause denial of service, runaway cost, or opportunistic cryptomining. In an agent system, that abuse is especially awkward because the work may appear to be legitimate until the bill or outage arrives.

The baseline should therefore bound resources by default. Every run should have a measured blast radius in time and capacity. If a task legitimately needs more resources, grant them for that invocation and nothing broader. That rule makes abuse observable and makes accidental overload less likely to propagate across tenants or across unrelated jobs.

### **Least privilege only works when the safe path is easy**

The design failure mode is rarely philosophical; it is operational. If a developer has to fight the platform to declare a narrow scope, broad grants will win by inertia. The secure path therefore needs to be the easiest path: reviewable scope templates, deterministic denials, stable defaults for common tool classes, and inheritance rules that narrow by default rather than widen. When the platform produces confusing rejections, people add exceptions. When it produces over-broad grants, people stop noticing.

That is why hard defaults matter. A good system makes it easier to specify “read this directory and write that scratch path” than to ask for “whatever is needed.” It makes it easier to request one outbound destination than to grant general egress. It makes it easier to use a short-lived task token than to reuse a service account. The platform should reward precision and make ambiguity expensive.

### **Domain by domain, the baseline remains the same**

The least-privilege rule is easiest to enforce when you treat each do-

main separately.

For *filesystem scope*, begin with no inherited host paths, a minimal read set, one writable scratch area, and controlled export points. For *network scope*, begin with no unrestricted outbound access, explicit destination allowlists, and no reliance on local network placement. For *identity*, begin with no static credentials in images, no broad agent service accounts, and no authority beyond the invocation unless the task requires it. For *devices and kernel-adjacent surfaces*, begin with no privileged mode, no arbitrary device exposure, and no hidden capability to alter host state. For *compute resources*, begin with bounded CPU, memory, process count, and time.

These are not implementation details; they are the shape of the system. Later chapters will show how to realize them with policy, container settings, and stronger isolation boundaries. The constraint itself does not change.

### **Version-aware mechanisms still serve a version-stable principle**

Many of the mechanisms that enforce least privilege are platform-specific. Docker seccomp profiles, Kubernetes security policies, Linux capabilities, Landlock, and runtime defaults all vary with version and environment. That variation matters when you implement the baseline, but it does not change the design rule. A `RuntimeDefault` seccomp profile may be a sensible starting point in one cluster, yet it still inherits the runtime's default assumptions rather than expressing workload-specific intent. For higher-risk tools, you often need a tighter profile or a stronger boundary altogether.

The same caution applies to cluster policy and kernel features. If you expect stable semantics over time, pin or validate them deliberately. Do not assume that a label, a default, or a convenience template will remain equivalent across runtimes. The principle is stable; the enforce-

ment surface is not.

### **Why defaults matter more than exceptions**

A platform that relies on exception handling to preserve least privilege usually loses to scale. Every exception creates a precedent, and every precedent becomes a candidate default. If a team can widen permissions for one urgent tool, another team will expect the same shortcut for another urgent tool, and soon the narrow baseline is only a policy document.

Default-deny avoids that drift. It makes the secure posture the normal posture. A new tool run begins with no durable authority, no ambient network reach, no writable shared state, and no implicit device access. Anything beyond that must be requested explicitly and can be revoked automatically when the invocation ends. That model is more honest about how agent systems operate: each run is a fresh access request, not a continuation of a permanent trust relationship.

### **The practical decision rule**

When you decide whether to grant a privilege, ask three questions: is the privilege necessary for this invocation, can you bound it tightly, and can you verify that it was actually used as intended? If the answer to any of those questions is no, the baseline should remain deny. That rule is conservative by design. It reduces the blast radius of a compromised tool, a misdirected model, or a confused operator, and it keeps the platform from smuggling convenience into production as if it were safety.

Least privilege becomes real only when the system makes overreach difficult by default. Once that holds, the later choices about scopes, runtime hardening, and stronger sandbox boundaries become refinements of a disciplined baseline rather than compensations for a permissive one.

## 1.4. Defining Assets and Data Boundaries for AI Tool Runs

The security boundary around a tool-using agent is only meaningful after you decide what the platform is protecting. If you do not identify the sensitive artifacts, the system will protect whatever happens to be convenient: a working directory, a log stream, a token cache, or an entire host. That is how implicit data sharing enters production. The correct starting point is an asset inventory and a set of explicit data boundaries that describe what may enter a tool run, what may be transformed inside it, what may leave it, and what must remain auditable after it ends.

**Start with named assets, not abstract risk** A useful inventory includes more than obvious secrets. You need to name user-provided files, retrieved corpora, prompts and system instructions, local caches, temporary outputs, network destinations, model transcripts, execution metadata, and audit evidence. Some of those assets are secret, some are integrity-critical, some are sensitive only in aggregate, and some become dangerous only when reflected back into later model context. Treating them all as one category leads to either overexposure or paralysis.

A practical way to ground the inventory is to express each asset in terms the runtime can actually enforce:

|                       |                                                                   |
|-----------------------|-------------------------------------------------------------------|
| <b>Source inputs</b>  | Mount points, pre-opened directories, stdin, or brokered fetches. |
| <b>Secrets</b>        | Scoped tokens, environment handles, or ephemeral credentials.     |
| <b>Network access</b> | Socket policies, proxy rules, and destination allowlists.         |
| <b>Outputs</b>        | Export paths, object-store prefixes, and artifact gates.          |
| <b>Evidence</b>       | Correlated logs, run IDs, and retention classes.                  |

That translation matters because you cannot defend an asset that the runtime does not know how to distinguish. A named directory is easier to bound than “the workspace.” A destination allowlist is easier to reason about than “the network.” A retention class is easier to audit than

“logs somewhere.” You want boundaries that map directly to enforcement points later in the stack.

**Follow the data through the run lifecycle** A tool run has a lifecycle, and each phase creates a boundary decision. Inputs arrive from the user, a retrieval system, an uploaded file, a service connector, or cached state. You materialize those inputs into the run as files, streams, environment variables, brokered fetches, or pre-staged objects. During execution, the tool may parse, enrich, compress, summarize, index, or transmit the data. On completion, it may emit a report, a side effect, a denial message, a diagnostic trace, or a structured observation for the planner.

That lifecycle exposes the control questions you actually need to answer:

- What may enter the sandbox at all?
- What may enter only after redaction or transformation?
- What may be writable, and what must remain read-only?
- What may be exported, and to which destination?
- What may survive teardown, and for how long?
- What may be visible to the model, and what should remain operator-only?

If you skip this analysis, the platform will smuggle data across boundaries by default. A raw tool stdout stream may become model context. A temporary file may become a persistent cache. A diagnostic log may capture a credential. A retrieved document may be copied into an output artifact without anyone deciding that disclosure was permitted. The lifecycle view makes those transfers explicit enough to control.

**Confidentiality boundaries need concrete handles** Confidentiality is usually the first boundary people imagine, but it is easy to implement it too broadly or too narrowly. The safe pattern is to segregate secrets and privileged tokens from untrusted inputs and from broad logs. If a tool needs to read confidential material, broker that access rather than materializing everything into the run. If a tool needs a secret only to call a downstream API, provide the narrowest credential that can satisfy that call and keep it out of the general workspace.

Concrete handles make that boundary operational. A mount point can hold a read-only corpus. An environment variable can carry a short-lived token. An object-store prefix can constrain output placement. A pre-opened directory can remove the need for path traversal. These are not just convenience mechanisms; they are the means by which you turn a data classification into an enforceable boundary.

You should be especially careful with logs. Operational logs often collect the same artifacts that are most dangerous to expose: prompts, headers, tokens, snippets of retrieved content, and error traces that include file paths or request bodies. If the platform needs both evidence and privacy, separate the evidence path from mutable workspace state and restrict who can read it. Audit evidence is itself an asset, and it should be tamper-evident, time-synchronized, access-controlled, and segregated from the data the tool can overwrite.

**Integrity boundaries protect later reasoning, not just storage** Confidentiality gets most of the attention, but integrity is often the more subtle boundary in agent systems. A tool output may be technically harmless as a file and still be dangerous as an input to later reasoning. Structured summaries, code patches, metadata blobs, and generated instructions can all become carrier channels for hidden influence if later stages trust them uncritically.

To keep integrity boundaries clear, you should type outputs and tag provenance. A downstream planner should be able to distinguish a validated artifact from a raw tool trace, a user-supplied file from a generated report, and a human-approved change from an automatically synthesized suggestion. If an artifact is destined for later model context, constrain it so the tool cannot smuggle arbitrary instructions or sensitive content into the next step. That means formatting discipline, provenance labels, and explicit export paths, not just content filters.

This is why data boundaries are not only about where bytes can travel. They are also about what semantics survive the journey. A file copied from one store to another may remain confidential. A log line copied into a prompt may become executable influence. A transformed output may be safe for humans but not safe for an untrusted downstream agent. Integrity boundaries make those distinctions visible.

**Retained evidence must not share state with mutable workspaces** Retention is a separate boundary because the data you need for audit and the data you want to discard often conflict. Ephemeral scratch data, caches, transcripts, and final evidence do not deserve the same lifetime. If you put them in the same writable workspace, the tool can erase proof, mutate history, or overwrite the record of a bad action. If you store them in the same bucket as user data, you risk preserving more than you intended.

A better design separates retention classes. Scratch state is short-lived and disposable. User artifacts are retained according to product policy. Audit evidence is immutable or append-only, tightly access-controlled, and retained for a defined period. Those choices let you answer a post-incident question that matters as much as prevention: what can you prove about the run after the fact?

That separation also helps with compliance and privacy. You may need

enough evidence to reconstruct a decision, but not enough raw data to create a second exposure surface. The platform should therefore preserve correlated metadata and decision records, not just voluminous transcripts. More data is not better if it cannot be defended.

**Filesystem boundaries have escape hatches** Filesystem scope is often described as if it were a clean tree of allowed paths. Real systems are messier. Host-linked paths collapse the boundary immediately. Namespace sharing can leak access through paths the policy never modeled. Landlock and similar mechanisms narrow filesystem-style access, but even those controls have edge cases: some resources reachable through `/proc/<pid>/fd` or other non-user-visible objects are not currently expressible as explicit restrictions. The practical lesson is simple: do not confuse a filesystem boundary with complete data containment.

The right response is not to abandon filesystem controls; it is to treat them as one boundary among several. Bind mounts, object-store prefixes, pre-opened file descriptors, and explicit export locations give you handles the runtime can enforce. HostPath-style mounts, host PID namespaces, and host network sharing are high-risk exceptions, not normal configuration. If you must use them, you should treat them as deliberate escalations of trust, not as implementation shortcuts.

**Network destinations are part of the data boundary** Data boundaries do not end at the filesystem. Once a tool can make outbound requests, every destination becomes a potential export point. That includes APIs, webhook endpoints, package mirrors, telemetry collectors, and any service the tool can reach through an internal network. If the tool can choose destinations freely, then any readable data can become outbound data.

You should therefore model network destinations as concrete handles.

Use destination allowlists rather than general egress. Mediate DNS if the destination name itself matters. Distinguish between a tool that may call one known API and a tool that may reach arbitrary internal services. The distinction is not cosmetic: a broad network boundary can turn a contained read-only analysis tool into an exfiltration channel with a single mistaken request.

**Identity and data boundaries are coupled** A data boundary without identity control is incomplete. The same asset may be safe for one tool and forbidden for another. The same export path may be valid under one scope and invalid under another. The same token may be appropriate for a single read operation and inappropriate for any write. You need identity to decide whose request is moving the data, and you need data boundaries to decide where that data may go.

That coupling is why later chapters treat identity brokers and scope design as first-class controls. For this section, the key point is more basic: do not mix credentials with untrusted inputs, and do not let broad service accounts stand in for task-specific access. If a tool can touch sensitive data, it should do so through an explicit grant that matches the task. That grant should expire when the invocation ends.

**Ask what the model is allowed to see** Not every asset in the system belongs in model context. Model-visible summaries, retrieved documents, and tool outputs can all be useful, but each one increases the chance of leakage or prompt injection. The model should receive only the data it needs to perform the current task, and only in a form that is appropriate for reasoning rather than direct replay.

That usually means transforming raw material before exposure. Redact secrets. Compress verbose traces into typed summaries. Strip irrelevant metadata. Convert file contents into structured excerpts. Keep raw artifacts in the tool plane if the model does not need

them. The more directly you feed untrusted or sensitive content into context, the harder it becomes to distinguish normal reasoning from adversarial influence.

**A working boundary model for one run** For a single invocation, the cleanest mental model is a layered chain of handles: read-only inputs at the edge, a bounded writable scratch area, task-specific credentials, a limited set of destinations, and a separate evidence channel. The tool can transform data within the scratch area, but only approved outputs can cross into the export path. The planner can see validated summaries, but not every raw intermediate. Logs record the run, but not in the same mutable location that the tool can rewrite.

That model gives you a practical decision rule: if you cannot describe an artifact as an input, a transform, an output, or evidence, then you have not yet decided what it is doing in the system. Once you can name it, you can decide who owns it, who may read it, how long it lives, and where it is allowed to go. Those are the decisions that later identity, filesystem, and audit mechanisms enforce with precision.



## Chapter 2

# Designing Permission Scopes for Agents

*The hard part of sandboxing agents is not only building an isolation boundary, but deciding what should be allowed inside that boundary in the first place. This chapter treats permission scope design as the hinge between threat modeling and enforcement, showing how small mistakes in scope shape can quietly dominate blast radius, operability, and auditability long before containers or microVMs enter the picture.*

### **2.1. Modeling Permission Scope Across Filesystem, Network, Compute, Identity, and Device Boundaries**

A tool run that is described only as “filesystem access” or “network access” is already too vague to review safely. That language hides which

tree may be read, which paths may be written, which destinations may be contacted, which identities may be borrowed, and which host resources may be consumed along the way. The result is familiar: a document processor reads the wrong project checkout because the working directory was treated as a safety boundary, a fetcher reaches an unapproved API because “external network” was left undefined, or a code-generation job stalls a host by exhausting memory or file descriptors while still staying inside a coarse policy. A useful permission scope must be written as a contract with dimensions, not as a single allow/-deny label.

That contract is easiest to reason about when you decompose it into five orthogonal boundary classes: filesystem, network, compute, identity, and device. Each class answers a different question. Filesystem scope asks what data the tool may observe or modify. Network scope asks where it may communicate and through what request paths. Compute scope asks how much host capacity it may consume. Identity scope asks whose authority the tool may exercise when it calls downstream systems. Device scope asks which kernel-visible capabilities and accelerators it may touch. These dimensions interact, but they are not substitutes for one another; a narrow file scope does not constrain outbound exfiltration, and a restricted network scope does not stop a job from consuming all memory or attaching to a GPU.

A practical way to pin down the contract is to write it in a structured form that separates subject, action, object, and envelope. For example:

```
tool: report-aggregator
subject: run-id: 83d2e9
filesystem:
  read:
    - /workspace/incoming/reports/**
  write:
    - /workspace/outgoing/**
    - /tmp/report-aggregator/**
  execute:
    - /usr/bin/python3
network:
```

```
egress:
  - https://api.partner.example
mode: proxy-only
compute:
  cpu: 2
  memory: 1GiB
  wall_clock: 15m
  fds: 256
identity:
  downstream:
    audience: partner-api
    token_type: access-token
device:
  gpu: none
  /dev:
    - deny-all
prohibited:
  - privilege_escalation
  - host_filesystem_mounts
```

The value of a template like this is not that it is complete; the value is that it forces you to state what is in scope and what is excluded. Reviewers can challenge a specific path prefix, a specific audience, or a specific resource ceiling. Engineers can later translate each field into container settings, kernel filters, broker policy, or sandbox controls without re-defining the intent.

## Filesystem scope

Filesystem scope is usually the first place where ambiguity becomes dangerous because tools often touch source trees, caches, secrets, and generated artifacts in the same execution. A precise policy distinguishes read, write, append, create, delete, metadata, and execute permissions. Those distinctions matter operationally. Reading a path is not the same as being able to alter it. Appending to a log is not the same as truncating it. Enumerating directory names is not the same as opening file contents. Executing a binary is not the same as inspecting it.

Path semantics need equal care. A scope that says “allow

`/workspace/project`” is not automatically safe if the implementation treats it as a string prefix and ignores symlinks, mount points, or path traversal. A path comparison must be evaluated after normalization, and the policy must define whether the authority applies to the canonical target or only to a textual pathname. If a tool can follow a symlink out of a permitted directory, the apparent boundary becomes a fiction. That is why policies should describe both the nominal path set and the resolution rules that enforce it. In practice, the safe unit is often a resolved mount or volume, not an arbitrary directory string.

The distinction between immutable inputs and writable scratch space also matters. You often want a tool to read a fixed corpus but write only to an isolated staging area. If you collapse those into a single “workspace” entitlement, you lose the ability to reason about accidental overwrites and prompt-injection-driven edits. A better contract states that source inputs are read-only, generated outputs must land in an export directory, and internal state may only live in a throwaway scratch tree. That way, a reviewer can ask whether the tool really needs to modify anything outside the scratch boundary.

It is also useful to separate internal state from exported artifacts. A build tool may create temporary objects, but only a specific subset should leave the sandbox. A file scope that permits “write anywhere under the project tree” is far too loose if the only required output is a single report file. The contract should identify approved export paths, and it should specify whether the tool may delete, replace, or only append to them. If a tool needs to redact or overwrite a prior output, that permission should be explicit rather than inferred.

### **Network scope**

Network scope is about destinations, request classes, and mediation mode, not just “internet access.” The most useful distinction is whether the tool may speak to a specific service, a class of partner services, an ar-

bitrary domain pattern, or only a proxy that performs additional policy checks. A policy that allows “network” says too little to review because it does not distinguish a package registry from a webhook endpoint, or a metrics collector from a data exfiltration channel.

The destination name itself should be treated carefully. DNS names, resolved addresses, and certificate identities are not interchangeable. If policy is written only against resolved IPs, it may break when the service moves. If it is written only against hostnames, it may be bypassed by resolving a different name to the same address or by talking to a direct IP when hostnames were intended. A strong scope describes the authoritative identity the tool may target, and it states whether enforcement occurs before DNS resolution, after resolution, or through an explicit proxy that re-evaluates the request. That distinction is especially important when the goal is partner API access rather than broad internet reachability.

You also need to say what kind of request the tool may make. A fetcher that may issue GET requests to a documentation site is not necessarily allowed to POST data to the same site. A deployment helper that may call a control API is not necessarily allowed to open arbitrary TCP sockets. A network scope should therefore include the call class: read-only retrieval, callback delivery, authenticated API invocation, or general socket use. The tighter the request class, the easier it is to distinguish expected automation from covert exfiltration.

Proxy-only mediation is often the cleanest policy surface because it gives you a single observation point for destination, protocol, headers, and rate. But even then, the scope should not rely on the proxy alone to imply trust. State the allowed endpoints and methods in the permission contract, then let the proxy enforce them. That separation makes reviews meaningful and avoids the common error of assuming that “it goes through a proxy” is itself a security boundary.

## Compute scope

Compute scope extends least privilege beyond data access. A tool that can only read the right files and call the right API can still become unsafe if it can monopolize the host. CPU quotas, memory ceilings, wall-clock limits, file descriptor budgets, process count limits, and storage quotas all belong in the permission model because they define how much damage a single run can cause before control returns to the orchestrator.

Wall-clock duration and CPU entitlement are different resources. A long-running job may be acceptable if it uses little CPU, while a fast loop may still be harmful if it spins too aggressively. Memory ceilings are equally important because out-of-memory termination often occurs after a tool has already produced partial side effects. If your scope says nothing about memory, a seemingly harmless parser can become a denial-of-service vector by loading a large archive or an adversarial input set into RAM.

File descriptors deserve explicit treatment because many tools leak them indirectly through libraries or subprocesses. A seemingly narrow job may open sockets, logs, pipes, and temporary files until the process hits system-wide limits. If you do not budget descriptors, you can end up with a tool that cannot complete even though CPU and memory remain available. Similar reasoning applies to process count: recursive spawn behavior, parallel workers, and unbounded retries can turn a modest workload into a fork bomb in miniature.

Compute scope should also identify whether the tool may expand its own resource envelope through child processes, threads, or background daemons. If a policy allows one Python interpreter but not uncontrolled subprocess trees, that condition must be written explicitly. Otherwise, the tool can offload work to helpers that inherit the same ambient authority and bypass the intent of the envelope.

These limits are not only about safety; they also shape predictability. A resource envelope that is part of the scope contract lets you reason about concurrency, queuing, and noisy-neighbor effects. It becomes possible to compare tool classes in review: a read-only analyzer might be allowed 512 MiB and 60 seconds, while a build step might require 4 GiB, more file descriptors, and a larger scratch volume. The numbers are not arbitrary decoration; they are part of the authorization decision.

### **Identity scope**

Identity scope answers a different question: whose authority is the tool exercising when it crosses system boundaries? A tool may run under one principal locally and act under another principal remotely. That distinction matters because the agent's authorizing identity, the run-time identity, and the downstream identity may all differ.

The contract should therefore distinguish the local subject from any borrowed credential. If a run uses a workload identity to obtain a cloud access token, the policy should identify the intended audience of that token, the downstream service it may be presented to, and whether the token is intended for delegation or impersonation. A token intended for one API should not be reused as a general bearer credential for unrelated systems. Likewise, an identity minted for a single run should not quietly persist into later runs just because the process is still alive.

Audience restriction is one of the most useful identity controls because it keeps credentials from becoming generalized proof of authorization. A tool that may consume a token for `partner-api` is not automatically allowed to present that token to a storage service, even if both live in the same cloud account. The scope should spell out the target audience, the credential type, and any claims or selectors that bind the credential to a specific execution purpose. That makes downstream use auditable and narrows the blast radius of leakage.

It is also helpful to separate identity used for control-plane actions from identity used for data-plane actions. A tool may need to register itself with a broker or request a lease, but that does not mean it should be able to invoke arbitrary administrative APIs afterward. When the identity model is collapsed, a tool can exceed its purpose simply because the same credential authenticates too many roles. A precise scope avoids that by naming the exact downstream calls the authority supports.

### **Device scope**

Device scope is often overlooked because many developers treat hardware access as a runtime detail, not an authorization boundary. That assumption fails quickly once a tool can touch GPUs, accelerators, special-purpose inference cards, raw block devices, or sensitive kernel interfaces under `/dev`. Device access can expose data, affect system stability, or open low-level attack surfaces that are much broader than a single file read.

The contract should state whether the tool may use no devices, only virtualized devices, or a specific hardware class. If a job needs a GPU for model inference, that does not imply access to every accelerator on the node. If it needs a serial interface or a virtual device, that should be listed explicitly. A default-deny stance for devices is usually appropriate because the reachable surface is often larger than it appears.

You should also separate device access from ordinary I/O. A tool that can write to a block device can bypass file permissions entirely. A tool that can access kernel-facing interfaces may obtain capabilities not visible in the application logic. Even when the device is virtualized, the policy should state whether the tool may mount, remap, or reconfigure it. The relevant question is not whether the device is physical; the relevant question is whether the device grants a capability beyond ordinary process memory and filesystem operations.

For GPU and accelerator workloads, it is useful to specify whether the

tool may allocate shared memory, persist model caches, or load vendor-specific runtime libraries that themselves reach into `/dev`. Those side effects can transform a narrow computational task into a broad hardware entitlement. If you need a clean boundary, define which device nodes are allowed, whether the tool may create new device-backed mappings, and whether the driver stack is part of the trusted base or part of the controlled surface.

### **Composing the dimensions**

The real contract is the intersection of all five dimensions, not the union. A scope that is safe on one axis may still be unsafe overall if another axis is unconstrained. For example, a tool may have read-only filesystem access but unrestricted outbound network access. That still permits exfiltration. Or it may have a strict network allowlist but unlimited compute and device access, which can still let it degrade the host or probe hardware in ways that were never intended. Least privilege only works when every dimension is bounded.

This intersection model also clarifies the difference between required authority and incidental authority. A report generator might need to read source data, write a single output file, call one external service, consume modest CPU, and use no devices. Anything beyond that is incidental and should be excluded. A deployment helper might need broader network authority but tighter filesystem access. A model-serving tool might need accelerator access but very limited write authority. The shape of the scope should follow the task, not the convenience of the implementation.

A compact way to review a tool scope is to ask five questions in order: What local data can it read or modify? Where can it communicate, and through what request types? How much host capacity can it consume? What credentials can it present downstream? Which kernel-visible devices can it touch? If any answer is vague, the scope is underspecified.

A useful review pattern is to challenge the most permissive dimension first. Teams often spend time narrowing path prefixes but leave identity or device access broad. That creates a false sense of safety because the most visible boundary gets polished while the least visible boundary remains expansive. The safest designs name each dimension independently, then make the reviewer confirm that no hidden path, network route, credential, or device capability remains outside the explicit contract.

### A reviewable scope template

A well-formed scope specification can be read almost like an access matrix. A minimal example for a single execution might look like this:

```
subject:
  type: tool-run
  id: run-83d2e9

filesystem:
  read:
    - /workspace/incoming/reports/**
  write:
    - /workspace/outgoing/final.csv
    - /tmp/report-aggregator/**
  metadata:
    - /workspace/incoming/reports/**
  deny:
    - /home/**
    - /etc/**
    - /var/lib/**

network:
  allow:
    - host: api.partner.example
      scheme: https
      methods: [GET,POST]
  deny:
    - all_other_egress

compute:
  cpu_cores: 2
  memory_max: 1GiB
  wall_clock_max: 15m
  process_max: 8
```

```
fd_max: 256
scratch_max: 2GiB

identity:
  downstream:
    audience: partner-api
    credential: short-lived-access-token
  deny:
    - shared_user_credentials

device:
  gpu: none
  deny:
    - /dev/kvm
    - /dev/mem
    - /dev/sd*
```

The details here are intentionally concrete enough to challenge. A reviewer can ask why POST is needed, why the scratch budget is 2 GiB, why metadata access is needed in addition to read access, or why no device access is required. If the answer is “because the tool always works that way,” the scope is too coarse. If the answer is task-specific and reproducible, the contract is doing its job.

The same template also exposes anti-patterns. “Limited access” is not a permission. “Internal use only” is not a boundary. “Safe directory” is not a path rule. “Standard compute” is not a resource envelope. Those labels may be useful in conversation, but they are not reviewable. A scope specification becomes meaningful only when it states exactly what is allowed, exactly what is denied, and exactly which dimensions those decisions apply to.

## **2.2. Designing Temporal Scope with TTLS, Just-in-Time Grants, and Revocation Boundaries**

A permission scope that never expires is not a scope; it is standing authority with a friendly label. The practical problem is that real tool

runs are not instantaneous. They queue, authenticate, fetch credentials, warm caches, make retries, and sometimes wait on humans. If you tighten duration too aggressively, you break legitimate work. If you leave duration open-ended, you create authority that outlives the task that justified it. Temporal scope resolves that tension by making time a first-class part of the contract: when authority begins, how long it remains valid, whether it can be extended, and what actually happens when you try to revoke it.

The useful mental model is to separate five clocks that are often conflated in design reviews. Maximum run lifetime limits how long the tool may execute. Credential lifetime limits how long a presented token remains usable. Approval lifetime limits how long a human or policy decision remains fresh. Cache lifetime limits how long a broker, proxy, or sidecar may reuse an earlier decision. Artifact retention lifetime limits how long outputs and intermediate state remain available after execution ends. These clocks are related, but they fail in different ways. A run can stay inside its wall-clock budget while carrying an expired approval. A token can remain valid while the job has already exceeded its intended duration. An artifact can persist long after the authority that produced it has disappeared.

A scope review becomes much sharper when you name those clocks explicitly. For a given tool, you should be able to say: the job may execute for 15 minutes, its downstream token may live for 5 minutes, that token may be renewed once, the authorization decision expires when the current sandbox dies, and any generated output persists only until export completes. That statement is more informative than “short-lived access” because it binds the authority to the actual operational lifecycle of the run.

**A concrete short-lived grant** The following pattern captures the essence of temporal narrowing: mint credentials only after the sand-

box exists, bind them to a narrow audience, and cap their lifetime with an explicit TTL.

```
vault write -field=token \  
  auth/token/create \  
  policies=report-read \  
  ttl=5m \  
  renewable=true \  
  num_uses=20 \  
  display_name=run-83d2e9
```

A token like this is not just “short-lived” in the abstract. It has a renewal policy, a bounded use count, and a specific identity label for the run that asked for it. That combination lets you encode both duration and purpose. In systems that support workload-bound credentials, you can add a stronger binding through audience restriction or object binding so that the credential only works for the intended downstream service and the intended run identity. Kubernetes projected service-account tokens are a common example of this shape: you can set an audience and an expiration so that the token is useful only for the intended API and only for a limited period. When a platform caps the maximum expiration, that cap becomes part of the scope contract, not a hidden implementation detail.

**Just-in-time grants** Just-in-time issuance matters because pre-created environments tend to accumulate stale authority. If you launch a sandbox with a long-lived token mounted from the start, the credential may sit idle while the job waits for input or retries on transient failures. During that idle time, the blast radius expands for no security benefit. Issuing the credential only when the sandbox is ready shortens the window in which authority exists but is not yet needed.

That pattern is especially useful when you split a task into stages. A tool might begin in read-only mode, perform validation, and only then request write authority or a downstream API credential. The earlier stage proves that the run is legitimate; the later stage unlocks the narrower

privilege needed for the next action. This staged escalation does two things at once. It reduces the default scope at launch, and it makes the authorization review easier because the risky capability appears only after a concrete precondition has been satisfied. If the validation fails, the broader privilege never exists.

The design rule is simple: do not mint authority before you can point to a concrete need. If a run needs access to one storage bucket only after a checksum passes, issue that access only after the checksum. If a workflow needs a write token only after generating a candidate artifact, keep the write token out of the initial launch. Temporal narrowing is often cheaper than compensating with broader static permissions, because it reduces both attack surface and the amount of authority that must be explained in review.

**Renewal versus reauthorization** Renewability is where many teams overestimate safety. A renewable credential is not safer by default than a nonrenewable one; it is safer only if renewal is itself a controlled decision. Renewal should be conditioned on continued need, continued sandbox integrity, and continued policy validity. If you let the process silently refresh its own authority without rechecking those conditions, you have only moved the standing privilege boundary a few minutes into the future.

A good renewal policy distinguishes three cases. First, the run may refresh automatically because its context has not changed and the policy allows a bounded extension. Second, the run may request renewal, but the broker or authorization server re-evaluates the request against current state. Third, the run may have to stop and resume under a new explicit grant if the task has crossed a meaningful phase boundary. Those are different operational choices, and they should not all be described as “refreshable.”

The most important question is whether renewal preserves the original trust assumptions. If the sandbox has been tainted, if the input set has changed, or if the downstream resource has become more sensitive, a renewal request should fail closed. Renewal is not a convenience feature; it is a controlled reassertion of scope.

**Revocation is bounded, not magical** Revocation is where temporal policy meets operational reality. You can invalidate a token, but you cannot retroactively erase data that the process already read. You can close a proxy, but you cannot always stop an already-established TCP session immediately. You can kill a sandbox, but queued side effects or buffered writes may already be in flight. That is why revocation must be modeled as a set of boundaries rather than as a single event.

The useful revocation boundaries are concrete:

- **Credential invalidation:** the token or key no longer passes authentication.
- **Proxy refusal:** a mediation layer stops forwarding new requests.
- **DNS withdrawal:** a name no longer resolves through the expected path.
- **Process teardown:** the run receives a signal or is terminated.
- **Mount unavailability:** input or output volumes disappear with the sandbox.
- **Job cancellation:** the control plane stops scheduling or tracking the run.

Each boundary cuts off a different class of action. Credential invalidation helps when the process must fetch more data or call another service. Proxy refusal helps when outbound communication is centrally

mediated. DNS withdrawal may stop some clients, but it is weak if the destination address is cached or already connected. Process tear-down helps only if the process respects signals and does not have side-car processes persisting elsewhere. Mount unavailability is powerful for filesystem containment, but it does not remove data already copied into memory. Job cancellation is useful for orchestration, but it depends on the control plane reaching the executor in time.

That distinction matters because teams often describe revocation as if every boundary fell simultaneously. It does not. You should design assuming that some effects are immediate, some are delayed, and some are impossible to reverse once work has started. The point of revocation is therefore to reduce further harm, not to erase all traces of the run.

The timeline makes a crucial point visible: authority is progressively narrowed and then progressively dismantled. Before minting, the run has no downstream capability. During execution, authority exists but is bounded by audience, TTL, and the sandbox envelope. After revocation begins, some paths close quickly while others linger until the process, proxy, cache, or control plane catches up. You should treat those gaps as part of the system design, not as edge cases that can be ignored.

**Operational revocation paths** If you want revocation to matter in practice, you need more than an invalidation endpoint. You need several coordinated paths. Token revocation handles bearer credentials that are still checked online. Introspection handles systems that ask the authorization server whether a token remains active. Control-plane cancellation handles jobs that are still queued or running. Process teardown handles stateful work inside the executor. Network mediation handles outbound calls after a permission change. Each path reduces a different kind of residual risk.

Token introspection is especially useful when you cannot rely on a self-contained token expiring soon enough. It lets a protected resource query the current status of a token and retrieve associated metadata. That gives you a stronger revocation story because the resource no longer trusts the token forever after issuance. The trade-off is extra latency and an ongoing dependency on the authorization server. That dependency is worth paying when revocation responsiveness matters more than offline autonomy.

Offline or self-contained tokens have the opposite profile. They are fast and resilient, but they are also harder to revoke quickly because the verifier may not consult a central source on every request. In those cases, TTL becomes the principal control. The shorter the lifetime, the smaller the residual window after a compromise. That is why TTL is not merely an administrative convenience; it is the design lever that compensates for delayed revocation.

**What can still happen after revocation** A common design mistake is to assume that once a grant is revoked, the associated task becomes harmless. That is only true if the task has no cached inputs, no buffered outputs, no pending retries, and no alternate paths. In reality, a run may already hold data in memory, may have staged writes that have not yet flushed, may have queued retries in a worker, or may have copied credentials into a child process. Revocation stops future authority checks, but it does not erase prior state.

That is why you should distinguish between *authority to act* and *capacity to continue*. A revoked token may stop further API calls, but a process may still render a report from data already fetched. A canceled job may stop new output from being committed, but it may not stop a sidecar from finishing an in-flight write. A killed process may release a mount, but the generated artifact may still be sitting in a shared cache. Temporal scope therefore requires you to think not only about when

permission ends, but also about what useful work remains possible after it ends.

This is also where approval lifetime matters. If human approval expires while the run is still active, renewal should force a fresh decision rather than assume continuity. Otherwise, the run continues on the strength of a stale judgment. A good policy records whether the grant expired naturally, was renewed, or was forcibly terminated, because those outcomes imply different residual risk and different operational follow-up.

**Designing for fail-closed behavior** A temporal policy is only useful if it fails closed under uncertainty. If the broker cannot confirm renewal, the grant should expire. If the authorization server is unreachable for an online-introspected token, the verifier should stop treating the token as active unless your system explicitly defines a safe degraded mode. If the teardown signal cannot reach the executor, the control plane should still withdraw mounts and stop accepting new work. Delay is sometimes unavoidable; indefinite continued authority is not.

Fail-closed behavior also means not hiding authority inside ambient process state. Long-lived bearer tokens in environment variables are a classic anti-pattern because they are easy to copy, easy to leak, and hard to retract from a running process tree. Storing authority in a shared cache with no expiry is equally poor. A scoped run should receive only the minimum authority it needs, with an expiry that aligns with the current phase of work and a renewal path that rechecks the same conditions that justified issuance.

The strongest temporal designs make expiry visible to the operator and to the program itself. The program should be able to see how much lifetime remains, whether renewal is possible, and which actions will fail once the grant ages out. That visibility helps you avoid surprise failures

in long-running tasks and reduces the temptation to compensate with oversized lifetimes “just to be safe.”

**Choosing TTLS** There is no universal TTL that fits every tool. You choose duration by balancing legitimate work duration, sensitivity of the downstream resource, confidence in revocation points, and the cost of renewal friction. A short TTL reduces blast radius but increases the number of renewal decisions. A longer TTL improves throughput but increases exposure if the run is hijacked. The right answer depends on whether the run is interactive, batch-oriented, retry-heavy, or likely to stall on external dependencies.

A good heuristic is to set the TTL to cover one meaningful phase of work, not an entire open-ended session. If the tool fetches data, processes it, and then publishes a result, those are distinct phases. Give the fetch phase one grant, the processing phase no external authority, and the publish phase a separate grant. If the tool must run for hours, break that span into bounded renewals tied to checkpoints. That way, stale authority cannot survive simply because the overall task is long.

The same logic applies to revocation latency. If your revocation path is slow, make the TTL shorter. If your TTL is necessarily longer, invest more in online checks, proxy mediation, or control-plane cancellation so that compromise is not a free ride until natural expiry. Temporal scope works best when issuance, renewal, and teardown all point in the same direction: authority exists only while the task legitimately needs it, and it disappears as close as possible to the moment that need ends.

### 2.3. Choosing Between Per-Invocation, Per-Agent, and Per-Workflow Permission Models

A scope can be perfectly defined on paper and still be poorly assigned in practice. The assignment model determines who holds authority, for how long, and across how many actions the authority can accumulate. That choice often matters more than the exact contents of an allowlist, because a narrow permission attached to the wrong unit of work can still create a large blast radius. The central question is not only *what* a tool may do, but *when* and *across which boundary of work* that authority persists.

The design choice usually appears early in agent architecture. A document conversion tool may need file access for one request, a web fetcher may need network access for several related calls, and a deployment helper may need a credential only long enough to publish one artifact. If you bind all three to the same long-lived principal, you simplify orchestration at the cost of persistence and reuse of privilege. If you reauthorize every step independently, you constrain the blast radius but increase broker traffic and operator friction. The right answer depends on the shape of the task.

#### A concrete token exchange pattern

When you want to narrow authority for a single call, token exchange gives you a precise mechanism for step-down delegation. A broker can accept a broader upstream grant and mint a narrower downstream credential for one tool invocation.

```
curl -sS https://auth.example.com/token \  
-d "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \  
-d "subject_token=$UPSTREAM_TOKEN" \  
-d "subject_token_type=urn:ietf:params:oauth:token-type:access_token" \  
-d "audience=partner-api" \  

```

```
-d "scope=reports:read" \  
-d "requested_token_type=urn:ietf:params:oauth:token-type:  
    access_token"
```

That shape is useful because it makes the downstream token specific to the immediate call. You can narrow audience, scope, and sometimes token type without reusing the original credential directly. The important architectural point is that the downstream capability is derived for a particular unit of work, not inherited indefinitely.

### **Per-invocation scope**

Per-invocation scope ties authority to a single tool call or action. Each invocation receives a fresh authorization envelope, and that envelope dies as soon as the call completes or times out. This is the strongest model for reducing blast radius because it prevents one action from silently seeding the next. If a prompt-injected instruction reaches the agent, it can only influence the next call that is individually authorized; it does not gain ambient authority over the rest of the session.

That containment has a cost. Per-invocation flows tend to increase broker latency, credential churn, and user-visible approval prompts. They can also complicate stateful protocols that expect continuity, such as multi-step authentication handshakes or long polling sessions. You pay for safety with orchestration overhead. The cost is acceptable when the action is sensitive, the target resource is valuable, or the task decomposition is uncertain.

Per-invocation scope is the natural fit when any one of the following is true:

- the call can independently create, modify, or exfiltrate sensitive data;
- the call boundary aligns with a clear semantic unit, such as “fetch one URL” or “write one artifact”;

- the downstream service is valuable enough that reuse of credentials would be risky;
- the agent's context is expected to contain untrusted or changing instructions.

The model works best when the broker can mint narrow, short-lived credentials on demand and revoke them aggressively after use. That does not eliminate all risk, but it prevents stale authority from piling up across unrelated steps. It also produces cleaner audit trails because each action maps to one grant and one decision.

### **Per-agent scope**

Per-agent scope binds authority to the session or agent instance. The agent receives a credential once, then reuses it across multiple actions until the session ends. This model reduces friction. You do not pay repeated setup cost, you avoid reauthenticating every step, and stateful interactions become simpler. For a conversational agent that repeatedly calls the same API, that convenience is attractive.

The risk is accumulation. A per-agent principal tends to become the default home for every privilege the system needs, and that principal often outlives the specific subtask that justified it. As the interaction continues, the agent context becomes a shared container for user intent, tool outputs, and potentially untrusted instructions. If one tool call succeeds under broad authority, later tool calls inherit that authority even when they are unrelated. Audit becomes coarse because many actions share one principal, and forensic reconstruction has to infer which operation actually consumed which permission.

Per-agent scope is therefore appropriate when the session is narrow, the tasks are homogeneous, the external systems are stable, and the cost of repeated authorization is high relative to the sensitivity of the work. It can also be acceptable when the agent is constrained to low-

risk read operations or when downstream systems are already mediating access with their own fine-grained checks. But once the agent can write, publish, deploy, or impersonate, per-agent authority becomes hard to defend unless you add strong compensating controls.

One useful way to think about the model is that it trades repeated decisions for continuous trust. That trade is viable only when the session itself is the unit of trust. If the session is long, interactive, or likely to absorb instructions from multiple sources, the model starts to look like standing privilege with a conversational front end.

### **Per-workflow scope**

Per-workflow scope sits between the two extremes. Authority persists across a bounded sequence of related steps, but not across arbitrary session activity. A workflow may fetch source material, transform it, validate it, and then publish the result. Those steps share a purpose, so continuity has value. Unlike per-agent authority, workflow scope lets you keep the grant attached to the operational pipeline rather than the conversational container.

This model is strongest when the workflow has explicit stage boundaries and typed transitions. You want to know when read-only access is enough, when write authority becomes necessary, and when a downstream credential should narrow further before the final publish step. That structure makes workflow scope easier to review than agent scope because the purpose of the grant is traceable to a finite job graph. It is also easier to revoke because you can cancel the workflow without worrying about unrelated conversational state.

Per-workflow scope becomes dangerous when the workflow itself is vague. If the boundaries are not explicit, a workflow grant can quietly turn into long-lived ambient authority under a more disciplined name. That failure mode is common in automation systems where a single orchestration object spans many minutes or hours and touches several

unrelated resources. In that case, the workflow has become the de facto principal, and the scope is only as good as the workflow definition.

The best use of workflow scope is as a structured compromise: broad enough to avoid reauthorizing every microscopic step, narrow enough that the grant expires when the job graph ends. If you can define the workflow as a stable sequence of stages with known resources, you can often get most of the security benefit of per-invocation design without the operational noise.

### **How the three models differ**

The three models are best compared along four axes: blast radius, user friction, orchestration complexity, and audit clarity.

Blast radius is smallest in per-invocation scope because each action has to justify itself independently. It grows in per-workflow scope because a chain of related steps can share authority. It is largest in per-agent scope because a long-lived session can reuse whatever was granted early on. If you expect prompt injection, tool confusion, or untrusted intermediate outputs, blast radius should dominate the decision.

User friction moves in the opposite direction. Per-invocation scope can require repeated approvals or repeated broker calls, which is annoying for routine tasks. Per-agent scope minimizes interruption and feels smoother to operators. Per-workflow scope sits in the middle and often provides the best balance when a task is naturally decomposed into stages.

Orchestration complexity is also asymmetric. Per-invocation models require a broker that can mint and validate grants rapidly, but the logic is conceptually simple. Per-agent models are simpler to wire up initially, yet they hide complexity in reuse, revocation, and state cleanup. Per-workflow models demand a clear workflow engine or control plane that knows where one stage ends and another begins. They are often

the most demanding to implement correctly because the control plane must understand the job graph, not just individual calls.

Audit clarity usually favors per-invocation design. A single grant maps to a single action, which makes event logs easier to interpret. Per-agent audit trails are broader but fuzzier: many distinct operations share one identity. Per-workflow logs can remain intelligible if the stages are named and the transitions are explicit, but they degrade quickly if the workflow is only a long sequence of generic steps.

Per-invocation: one action, one grant, one audit record.

Per-agent: one session, many actions, shared authority.

Per-workflow: bounded chain, stage-specific authority.

### **Task shape determines the right model**

The right assignment model depends more on the task structure than on the tool itself. A web fetcher that only retrieves one document per request fits per-invocation scope well. The same fetcher in a crawler that follows a bounded series of related links may fit per-workflow scope better. A deployment helper that must authenticate once to a release system and then perform several related checks may justify per-workflow authority, but only if each stage is explicit and the final publish step is isolated from earlier exploratory actions.

Per-agent scope is most defensible when the work is conversational and repetitive, the underlying resources are low risk, and the benefit of continuity is high. Examples include a read-only assistant that repeatedly queries the same internal knowledge base, or a helper that performs local transformations without external side effects. Even there, you should be careful not to let convenience make the agent principal the default bearer of broad rights.

A good heuristic is to ask whether the task can be partitioned into independent checkpoints. If every checkpoint can be authorized separately without breaking the job, per-invocation scope is usually preferable. If

the task needs continuity across a small number of explicit stages, per-workflow scope is usually the right compromise. If the task is truly session-centric and the risk is low, per-agent scope may be acceptable, but only with tight lifetime limits and strong revocation paths.

### **Delegation and narrowing across steps**

When a workflow chains tools, the strongest pattern is step-down delegation. A broad upstream grant should not flow unchanged into every downstream step. Instead, each stage should request the narrowest possible credential for its own purpose. Token exchange, session policies, and permission boundaries all serve this same architectural goal: the downstream principal may act only within the cap defined by the upstream one, and sometimes only on a subset of that cap.

That intersection model is particularly useful when one stage prepares data for another. A retrieval step may receive read access to a data source, a transformation step may receive only local filesystem access, and a publish step may receive a temporary write token scoped to one destination. If you reuse the retrieval credential for the publish step, you collapse those stages into a single authority domain and lose the benefit of separation. Narrowing authority at each step preserves the workflow while preventing privilege from accumulating just because the pipeline is convenient.

Permissions boundaries are useful here because they cap what a delegated principal can ever do without themselves granting access. That distinction matters in agent systems. A boundary can prevent a generated worker identity from exceeding the envelope of the workflow, even if the worker is misconfigured later. The same logic applies to session policies: they narrow temporary authority relative to a baseline principal, but they do not replace the baseline itself. The more explicitly you model those intersections, the less likely you are to mistake a convenience credential for a complete authorization story.

## Choosing under operational constraints

The assignment model is not selected in isolation. You choose it under constraints of latency, approval burden, revocation speed, and observability. If every invocation must wait on human approval, users will bypass the system or ask for overbroad grants. If every agent session caches a powerful credential, operators may accept the friction but end up with poorly bounded authority. If every workflow transition triggers new policy evaluation, the orchestration layer becomes more complex, but the resulting system is easier to reason about when things fail.

The practical decision criteria are straightforward:

- **Blast radius:** how much harm can one compromised step cause?
- **Continuity:** how much shared state does the task truly need?
- **Friction:** how often can you afford to reauthorize?
- **Revocation:** how quickly must you be able to stop authority?
- **Auditability:** can you reconstruct why each action was allowed?

If blast radius dominates, choose per-invocation or a tightly bounded workflow. If continuity dominates and the task is low risk, per-agent scope may be acceptable. If the job has a known series of stages, per-workflow scope usually gives the best balance. The model should follow the structure of the work, not the convenience of a single identity.

What matters most is consistency. A team that uses per-invocation grants for some tools and per-agent grants for others without a clear rule creates invisible policy drift. The result is not just administrative confusion; it is a change in the system's trust boundary. When you pick the granularity model deliberately, you are deciding where authority is allowed to pool, where it must be reissued, and where it must die with the action that created it.

## 2.4. Specifying, Reviewing, and Testing Scope Policy as an Enforceable Artifact

A permission scope stops being useful the moment it lives only in prose, slide decks, or incident lore. Teams can debate least privilege all day, but unless the scope is machine-readable, versioned, and tested, the operational system will drift away from the design under pressure. Ad hoc exceptions creep in through copied YAML, flags, or one-off broker rules, and the effective policy becomes impossible to audit. The point of treating scope as an artifact is to make the intended boundary explicit enough that you can diff it, test it, review it, and deploy it with the same discipline you apply to code.

That discipline matters because policy changes are not neutral. Adding one wildcard to unblock a release can silently widen access for every future run that inherits the same policy path. Tightening a rule can break a previously working workflow in ways that only appear under edge inputs. A policy artifact must therefore express both allowed behavior and expected denials. If you cannot describe what should fail, you have not really specified the boundary; you have only described the happy path.

### A concrete policy-and-test workflow

A minimal policy-as-code workflow makes the lifecycle visible. You author a policy, write positive and negative tests, run the test suite in CI, and examine the decision log before promotion.

```
package agent.scope

default allow = false

allow {
  input.tool == "report-aggregator"
  input.action == "read"
  startswith(input.path, "/workspace/incoming/")
}
```

```
deny_reason["path_not_allowed"] {  
  input.action == "write"  
  not startswith(input.path, "/workspace/outgoing/")  
}
```

```
package agent.scope_test  
  
test_allow_read_incoming {  
  allow with input as {  
    "tool": "report-aggregator",  
    "action": "read",  
    "path": "/workspace/incoming/a.csv"  
  }  
}  
  
test_deny_write_home {  
  not allow with input as {  
    "tool": "report-aggregator",  
    "action": "write",  
    "path": "/home/alice/secrets.txt"  
  }  
}
```

```
$ opa test policy.rego policy_test.rego  
PASS: 2/2
```

This pattern is useful because it turns intent into executable checks. The policy file says what is allowed and what is denied. The test file proves that the boundary behaves as expected. The CI result becomes an artifact of record, not just a convenience during development.

### What a good scope language must expose

A scope specification language does not need to be universal, but it must be explicit about the semantics that matter for review. You need to see the subject, the resource class, the action, the conditions, the identity claims, the time bounds, and the reason for any exception. If the language hides any of those behind implicit inheritance or ambiguous defaults, reviewers cannot answer the basic question: what changed?

Diffability is the first requirement. If a reviewer cannot see that a policy expansion added write access to one path, an external API audience, or a new device class, the artifact is not reviewable. The second requirement is explicit deny behavior. Deny-by-default policies are much easier to reason about than allow-with-many-exceptions policies because the surface of the policy is smaller. The third requirement is composability without hidden merge semantics. If one rule silently overrides another through ordering or inheritance, you will eventually ship a policy that nobody can mentally reconstruct.

A useful review format is to keep the scope object close to the deployment context but separate from the runtime plumbing. The policy should say what may happen. The runtime chapter can later explain how that policy is enforced through containers, microVMs, brokers, or sidecars. Keeping those layers separate prevents the common error of confusing policy intent with one particular enforcement mechanism.

### **Testing for permission boundaries, not just success cases**

Most teams test the permissive path and stop there. That is not enough. Authorization drift often appears in the denied cases, especially where path normalization, hostname resolution, protocol variants, or inherited defaults interact. A correct policy must reject what it should reject even when the input is shaped to look similar to a permitted case.

A good regression suite includes three classes of tests:

- Golden tests for operations that must be allowed.
- Negative tests for operations that must be denied.
- Bypass tests for previously discovered edge cases.

Golden tests prove that legitimate work still passes. Negative tests prove that the policy rejects obvious violations. Bypass tests preserve institutional memory. If a symlink traversal, hostname mismatch,

or wildcard overreach once bypassed the policy, the regression suite should encode that failure forever. Otherwise the same class of bug returns during the next refactor.

You should also test denial quality, not only denial existence. A denial should be deterministic, attributable, and safe to expose to the caller. If the policy fails with an opaque internal error, the caller cannot distinguish a legitimate refusal from a broken control plane. If the policy leaks sensitive matching details, the error message can itself become an information channel. A good negative test therefore checks the denial code, the message class, and the fact that no side effect occurred.

### **Versioning and semantic diffs**

Policy without version control is just configuration debt. Put the scope artifact under the same change management discipline as application code. Each change should have a diff, an owner, and a rationale. The review should answer three concrete questions: what new authority appears, what old authority disappears, and what assumptions make the change safe?

Semantic diffing matters more than textual diffing. A single line change from `/workspace/incoming/**` to `/workspace/**` is not a formatting tweak; it is a boundary expansion. Likewise, changing a token audience from one API to another is not a minor rewrite. Reviewers should read policy diffs as security diffs, not as syntax diffs. If your tooling can annotate the diff with the affected resource class or action set, use it. The more clearly the change surfaces privilege expansion or contraction, the less likely a reviewer will miss it.

You should also separate policy versions from runtime versions. A newer runtime might interpret a matcher, claim, or selector differently. If the policy format or enforcement backend changes behavior across releases, record that dependency explicitly. Otherwise a policy that was correct in one release can become broader or narrower in the next with-

out any text change at all.

### **Governance checks that reviewers can actually apply**

A good review process does not ask a human to simulate the whole policy engine. It asks a human to check a small number of concrete governance questions. NIST least-privilege and separation-of-duties controls are useful here as a checklist: does the scope limit the authority to the stated purpose, and does it keep incompatible duties from collapsing into one principal? That framing is useful even if the policy language itself is custom, because the review question remains stable across implementations.

For each proposed scope, the reviewer should be able to identify:

- the exact resources and selectors covered by the policy;
- the actions allowed on each resource;
- the conditions under which the policy applies;
- the identities or workloads that can exercise the scope;
- the exceptions and why they exist;
- the failure mode when the policy does not match.

If any one of those answers is vague, the policy is not ready. “Internal use” is not a resource class. “Limited access” is not an action set. “Temporary exception” is not a condition. Reviewers should reject adjective-driven language until it becomes measurable. A policy artifact must be precise enough that another engineer can challenge it without guessing what the author intended.

### **Decision logs as audit material**

When policy is enforced at runtime, decision logs become part of the evidence trail. A useful decision log captures the input facts that were

evaluated, the policy bundle or version that made the decision, the result, and a stable decision identifier. That record lets you reconstruct why a request was allowed or denied without reverse-engineering the live system state.

Decision logs also support offline debugging. If a request that should have been denied was allowed, the log helps identify whether the issue came from a bad rule, a stale bundle, an unexpected claim, or a matcher bug. If a request that should have been allowed was denied, the log helps distinguish a policy defect from a missing input field or a runtime serialization problem. That is operationally valuable, but it comes with a privacy cost: inputs may contain secrets, identifiers, or sensitive path names. Redact aggressively, and store only the data needed to explain the decision.

A mature logging policy distinguishes between audit needs and forensic overcollection. You do not need to log every raw header or every secret to prove that a policy made a particular choice. You need the minimal facts that support accountability. Overlogging makes the policy system itself a new data-hygiene problem.

### **Why benchmarking matters**

Fine-grained authorization is not free. Every contextual check adds work: parsing inputs, matching selectors, consulting claims, evaluating conditions, and emitting logs. In low-volume systems, that cost may be trivial. In agent systems that make many short tool calls, policy latency can become the bottleneck. A policy that is elegant but slow will tempt operators to bypass it or broaden it.

Benchmark the policy path before you commit to a design. Measure the hot path for both allowed and denied requests. Denials are not necessarily cheaper; they can be more expensive if they require deeper rule evaluation or richer logging. If a rule tree is complex, profile it and look for expensive string operations, repeated lookups, or unnecessary

external calls. A policy that runs only in happy-path demos is not ready for production.

The performance goal is not raw speed at any cost. It is predictable latency under realistic load. A policy that takes a few extra milliseconds but scales linearly and remains understandable is better than one that is faster in the common case but brittle under edge conditions. Where the policy cost is material, consider caching only if the cache key preserves the same security semantics as the underlying evaluation. A stale cache entry can be indistinguishable from a permission bug.

### **Testing the edges, not only the center**

The most useful tests are the ones that look like innocent variations of valid traffic. A path with `..` segments, a hostname that resolves to a different audience, a token with the right shape but the wrong claim, or a resource selector that differs by one character can reveal boundary mistakes that the center-path tests never exercise. If the policy uses prefixes, test both the intended prefix and the next directory over. If the policy uses audience claims, test both the correct audience and a near-miss. If the policy allows one device class, test the sibling class that should remain denied.

Edge-case tests are also where you validate deny specificity. If two different violations produce the same refusal, the system may be hiding important diagnostics. If two different violations produce different codes, the system may be leaking too much about the internal rule structure. The policy should expose enough detail to support operators without turning denial into an oracle for attackers. That balance is best checked with tests, not by convention.

### **A deployment flow that keeps least privilege durable**

A practical release process for scope policy can be simple and strict:

- author the policy change as code;

- write or update positive, negative, and bypass tests;
- review the semantic diff;
- run the test suite in CI;
- profile the hot path if the rule set changed materially;
- deploy the policy bundle to a narrow slice;
- inspect decision logs for unexpected allows or denials;
- promote only after the policy behaves as intended.

That flow prevents the usual decay pattern where a policy begins as a careful contract and ends as a pile of exceptions. The tests keep the intended denials alive. The logs show how the policy behaved under real traffic. The review keeps the policy aligned with the stated purpose. And the versioned artifact gives you a rollback point if a change widens authority or breaks legitimate work.

The larger point is that least privilege survives operational change only when you treat policy as an enforceable object with its own lifecycle. A scope that can be reviewed but not tested will drift. A scope that can be tested but not versioned will be hard to govern. A scope that can be versioned but not explained will be hard to defend. The durable form is the one that exposes intent, proves behavior, and leaves a clear trail from design to enforcement.



## Chapter 3

# Linux Isolation Primitives as the Building Blocks

*A sandbox for AI tools is not created by a single product flag; it is assembled from low-level kernel mechanisms that each constrain a different failure mode. This chapter shows how namespaces, cgroups v2, and mount patterns work together to create a usable low-privilege execution cell, then closes by confronting the uncomfortable truth that careful composition still leaves shared-kernel risk that later chapters must address.*

### 3.1. Namespaces and the Anatomy of Process Containment

A tool launched by an agent should look, from the tool's point of view, like it is running on a small machine that belongs to it alone. On the host, though, it is one process among many, sharing a kernel with the agent runtime, background services, and other workloads. Namespaces are the kernel's first major answer to that tension. They do not create a new machine, and they do not by themselves stop abuse, but they let you reshape what a process can see, name, and influence.

The practical value of namespaces comes from composition. Each namespace trims a different part of ambient reality: process trees, mount graphs, network interfaces, identity mappings, IPC objects, hostnames, cgroup paths, and time views. You use them together to turn one host into many execution cells. The reader who treats namespaces as a single feature will miss the real design problem: deciding which parts of the host must remain shared, which parts must be hidden, and which parts must be remapped so that the tool can function without inheriting the operator's trust.

A compact launch sequence makes the shape of the problem concrete:

```
unshare \  
  --user --mount --pid --net \  
  --ipc --uts --cgroup \  
  --fork --mount-proc \  
sh -c '  
  mount -t proc proc /proc  
  hostname toolcell  
  id  
  ps -ef  
'
```

That command is only a sketch, but it captures the important idea. You are not simply starting a process; you are choosing a set of views. The process will see a new PID space, a private mount tree, a private net-

work stack, isolated IPC objects, a separate UTS name, and its own user identity mapping. If you leave out the user namespace, many operations still require host-level privilege. If you leave out the mount namespace, the filesystem view remains ambient and hard to reason about. If you leave out PID isolation, the tool can observe and sometimes target host processes. The security value is in the combination, not the individual flag.

#### **PID namespaces: process visibility and init semantics**

PID namespaces answer a simple question: which processes can you name? Inside a PID namespace, a process sees only processes in its own namespace and descendant namespaces. It does not see ancestors. That one-way visibility matters more than it first appears, because many tool-sandbox failures begin with unexpected process reachability rather than a direct escape.

A process inside a new PID namespace also gets a special PID 1 role. That process becomes the namespace's init-like ancestor, which means it inherits signal-handling and reaping responsibilities that ordinary application processes usually ignore. If the init-like process exits, the namespace loses its controlling root and the kernel tears down the namespace's process tree. If it fails to reap children, zombies accumulate. If it ignores termination semantics, shutdown becomes unreliable. When you launch an untrusted tool, this is not an academic detail: the process wrapper needs to act as a correct subreaper or init surrogate so that child processes do not linger or outlive the intended execution window.

PID namespaces also change how you think about monitoring and control. Host-side supervisors can still see the real host PIDs, but code inside the namespace works with names that are local to the namespace. This is useful for operational simplicity, yet it also means that logs, signals, and debug output can become misleading if you do not

record both the namespace-local PID and the host PID. In practice, tool runners should treat PID namespaces as a containment layer and not as an access-control feature. They reduce confusion and limit process reachability, but they do not decide which syscalls are allowed, and they do not stop a process from exhausting CPU or memory.

### **Mount namespaces: filesystem topology as policy**

Mount namespaces are the most security-critical namespace for many tool runners because the filesystem is where secrets, credentials, source trees, caches, and outputs meet. A new mount namespace starts as a copy of the parent's mount list, so isolation does not begin with emptiness. It begins with inheritance. That inherited graph is the first trap: if you do not change mount propagation and then rebuild the view deliberately, a process can continue to receive mount events from the host or leak its own changes back out.

The operational rule is straightforward: after entering a new mount namespace, make the mount tree recursively private or slave before doing any further mount work. Without that step, you can create a filesystem layout that looks isolated but still participates in shared propagation. That means a mount in the sandbox may appear on the host, a host mount may appear in the sandbox, or a recursive bind may drag in more state than intended. The failure mode is subtle because the filesystem looks correct at launch time and drifts later when unrelated mount activity occurs elsewhere on the system.

Mount namespaces also separate view construction from path resolution. Even if the visible tree is private, a tool can still traverse paths inside that tree in unsafe ways unless the launcher constrains resolution semantics. That is why a mount namespace is necessary but not sufficient. It limits the topology; it does not guarantee that a path operation stays within the intended subtree. For that, later sections use path-resolution controls such as `openat2`, but the mount namespace is

the prerequisite that lets those checks operate on a small, explicit tree rather than on the host's entire mount graph.

#### **User namespaces: identity remapping and scoped authority**

User namespaces matter because they let you separate host identity from in-namespace authority. A process can have one set of user and group IDs outside the namespace and a different set inside it. Capabilities granted inside the user namespace are scoped to resources governed by that namespace hierarchy, not to the whole host. That distinction is the heart of rootless sandboxing: the tool can behave as privileged within a narrow scope while remaining unprivileged outside it.

That model is powerful, but it deserves careful handling. Root in a user namespace is not host root. It can create mounts, change IDs, and perform namespace-local privileged operations only where the kernel recognizes the namespace's authority. At the same time, user namespaces deliberately expose privileged kernel code paths to less-trusted callers, which expands the attack surface. A design that depends on user namespaces should therefore treat them as a de-privileging mechanism with a cost, not as a universal safety blanket.

The compatibility details matter. Some distributions enable unprivileged user namespaces broadly; others restrict them or apply additional policy. Some container runtimes rely on user-namespace behavior that is not identical to direct kernel use. If your sandbox launcher needs to run on multiple distributions, you should verify the kernel and sysctl settings rather than assume that rootless containment is available everywhere. The conceptual model is stable, but the operational envelope is not.

#### **The other namespaces: network, IPC, UTS, and cgroup**

Network namespaces isolate network devices, routing tables, inter-

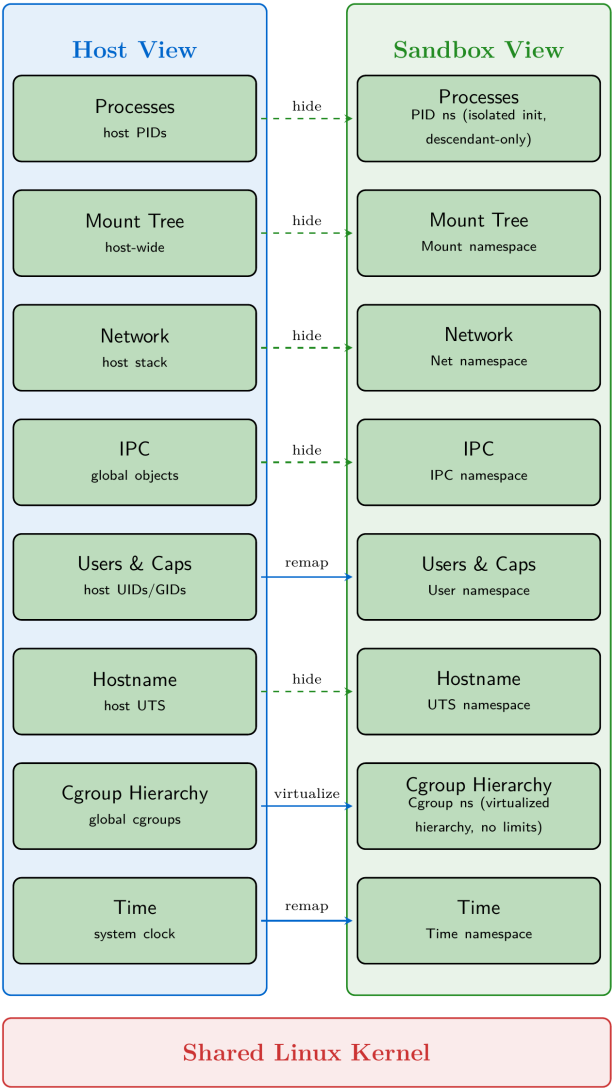
faces, and related state. They are the prerequisite for per-tool egress mediation, but they are not egress control by themselves. A private network namespace simply gives you a separate place to enforce policy. Without that boundary, network filtering later in the stack has to reason about traffic mixed with the host's own interfaces and sockets.

IPC namespaces isolate System V IPC objects and POSIX message queue namespaces. They prevent a tool from accidentally or maliciously reaching shared IPC state on the host. For tool runners this mostly matters as cross-process hygiene: a sandboxed process should not be able to discover, attach to, or interfere with IPC objects that belong to unrelated workloads.

UTS namespaces isolate hostname and domain-name identity. That is often treated as cosmetic, but it still reduces ambient cross-talk and makes logs and diagnostics more consistent when multiple tool cells run in parallel. It is also a useful signal that the process is running in a deliberately constructed environment rather than on the host.

Cgroup namespaces virtualize the view of the cgroup hierarchy. They do not enforce resource limits; they change how the process sees its place in the hierarchy. That distinction is easy to blur, but it matters operationally. Resource enforcement belongs to cgroups themselves, which are discussed separately. The namespace only makes the hierarchy look local so that the process does not learn about host-level structure it should not care about.

The point of listing these namespaces together is not symmetry. Their security weight differs. PID, mount, and user namespaces usually do the most work in tool sandboxing. Network, IPC, and UTS namespaces reduce ambient reach and improve hygiene. Cgroup namespaces make the resource view less revealing. All of them contribute to the overall shape of containment, but they are not interchangeable.



The diagram above is useful because it shows the real boundary condition: namespaces do not all do the same job, and they do not all sever

the same relationship to the host. Some namespaces hide state, some remap authority, and some simply localize a naming scheme. The process still runs on the same kernel, and many host resources remain shared unless later layers close them off. That is why namespace design should be read as a map of what the process can perceive, not as a proof of safety.

### **How the namespace set composes**

The usual mistake is to ask whether one namespace is enough. The right question is which namespace relationships your workload needs to retain and which ones must be cut. A tool that only parses local files may need a private mount namespace and a user namespace, but not network isolation if it fetches dependencies from a controlled internal service. A build tool may need IPC and UTS separation for cleanliness, while an analysis tool handling untrusted code might also need a separate PID namespace so that it cannot enumerate sibling tasks.

Composition also changes the semantics of the namespaces themselves. A mount namespace owned by a different user namespace is treated as less privileged in several kernel operations, which means the order in which you create namespaces affects what you can later mount or remount. That is one reason launchers typically establish identity remapping early, then construct the filesystem view, then drop privileges further before `execve`. The sequence is not aesthetic; it determines what the kernel will allow.

Another subtle interaction appears with PID and user namespaces together. A process inside a user namespace can gain the capabilities needed to create or configure a PID namespace, but the namespace-local privileges do not escape to the host. This is useful for rootless runtimes and risky at the same time: you gain operational flexibility, but you also expose more kernel logic to untrusted callers. The practical response is not to avoid user namespaces categorically, but to treat

them as part of a defense-in-depth layout that still needs cgroup limits, file-system hygiene, and later syscall filtering.

#### **Why user and mount namespaces carry special weight**

If one namespace pair dominates sandbox design, it is user and mount. The user namespace gives the sandbox a local authority model; the mount namespace gives that authority a filesystem topology to act on. Together they let you create a minimal, mostly self-contained environment without relying on host-root privileges.

The user namespace is especially important when you want to run tools as if they were privileged inside their own cell but harmless outside it. Rootless containment depends on this. Without it, the sandbox launcher either runs as host root or gives up on many mount and device operations. With it, the launcher can map an unprivileged host identity to a privileged namespace identity and then use that identity only within the namespace hierarchy. The subtlety is that privilege is not erased; it is scoped.

The mount namespace is the counterpart because authority is only useful if the visible filesystem is small enough to control. A process with local privilege but a host-wide mount tree is still dangerous. It can touch host state far beyond what a sandboxed tool should see, and it can make assumptions about path stability that break as soon as the host evolves. A well-built mount namespace turns the filesystem into an explicit contract: inputs here, scratch space here, exports there, everything else absent. That is the right mental model for untrusted agent-invoked tools.

#### **A practical launch pattern**

A sane launch pattern is to create identity isolation first, then build the visible filesystem, then add optional network and IPC isolation, and only then start the payload. In pseudocode, the flow looks like this:

1. create a user namespace and map a non-host identity to local root
2. enter a new mount namespace
3. make mount propagation private
4. mount a minimal root and /proc
5. attach scratch and output directories
6. optionally enter net, IPC, UTS, and cgroup namespaces
7. drop any remaining ambient privilege
8. exec the tool

That order reflects dependency, not preference. The mount namespace needs a trustworthy authority context. The user namespace supplies that context without giving host-wide privilege. The mount tree then becomes the concrete filesystem view that the tool will inhabit. Additional namespaces tighten other channels of interaction after the core filesystem and identity boundaries are in place.

The operational discipline here is to derive namespace choices from declared tool scope rather than from a one-size-fits-all runtime profile. A tool that only formats text does not need network access, and a tool that only reads a repository should not see the host's live IPC objects. Conversely, over-isolating a workload can break legitimate functionality and create brittle workarounds. Namespaces are not a badge of security; they are a design language for expressing exactly what kind of machine the tool should think it is using.

That framing stays useful as the chapter moves into resource scoping and filesystem layout. Namespaces define the shape of the cell. The remaining layers decide how much the cell can consume and what data it can touch.

### 3.2. cgroups v2 for Resource Scoping and Abuse Resistance

A sandbox that limits what a tool can *touch* but not what it can *consume* is incomplete. An agent may hand a tool a legitimate task, yet the tool can still stall the host by forking aggressively, allocating memory until the kernel reclaims aggressively, or burning CPU in a tight loop because the workload boundary only described access, not appetite. cgroups v2 closes that gap. It turns abstract resource permissions into enforceable runtime ceilings, so you can bound the blast radius of a run even when the payload is careless, adversarial, or simply larger than expected.

The central idea is simple: namespaces shape visibility, while cgroups shape contention. Namespaces say which processes, filesystems, network objects, and identities a task can perceive. cgroups say how much CPU time, memory, I/O bandwidth, and process fan-out the task may consume while it runs. For untrusted tool execution, both are necessary. A tool that cannot see the host is still dangerous if it can monopolize the machine. A tool that is carefully rate-limited but can read ambient secrets is still unsafe. The sandbox contract has to cover both.

A practical launch pattern makes the mechanics concrete:

```
mkdir -p /sys/fs/cgroup/tool-17

cat > /sys/fs/cgroup/tool-17/cgroup.subtree_control <<'EOF'
+cpu +memory +pids +io
EOF

cat > /sys/fs/cgroup/tool-17/cpu.max <<'EOF'
50000 100000
EOF

cat > /sys/fs/cgroup/tool-17/memory.high <<'EOF'
512M
EOF

cat > /sys/fs/cgroup/tool-17/memory.max <<'EOF'
768M
```

```
EOF

cat > /sys/fs/cgroup/tool-17/pids.max <<'EOF'
64
EOF

echo $$ > /sys/fs/cgroup/tool-17/cgroup.procs
```

That example is intentionally plain. It creates a cgroup, enables controllers in the parent, sets a CPU ceiling, adds a soft and hard memory boundary, limits process fan-out, and attaches a process. The details vary by runtime and privilege model, but the design pattern does not: translate the tool's declared budget into a cgroup before you start execution, not after pressure has already spread.

### **Why cgroups v2 is the right mental model**

The most useful shift from cgroups v1 to v2 is conceptual: one unified hierarchy replaces the fragmented controller-specific trees of v1. That matters because a sandbox should not have to reason about one hierarchy for memory, another for CPU, and a third for I/O with incompatible rules and overlapping placement constraints. In v2, you manage a single tree and enable the controllers you need. The operational result is less ambiguity about where a workload lives and which resource policies apply to it.

The hierarchy is not merely a directory structure. It is the enforcement topology. Parent cgroups define policy envelopes; child cgroups inherit placement and can tighten the bounds further. That makes cgroups naturally suited to multi-tenant tool runners, where you may want platform-level defaults at one layer and tool-specific budgets at the next. It also encourages a clean separation between supervisory orchestration and workload placement. You create the cgroup, configure the relevant knobs, then attach the process before 'execve' so that the first instruction already runs under the intended budget.

v2 also makes the distinction between *limits*, *protections*, and *allocations* explicit. These words are not synonyms. Limits cap consumption and can often be overcommitted across peers. Protections bias reclaim and throttling behavior to preserve some working set, but they can still be exceeded under pressure. Allocations are reserved shares that cannot be overcommitted in the same way, so invalid configurations may be rejected. When you design a sandbox, you should know which behavior you want: a ceiling that blocks abuse, a soft buffer that improves survivability, or a reserved guarantee that protects a critical workload.

### **CPU control: fairness before hard denial**

CPU control is usually the first knob you reach for, but it has two distinct modes with different trade-offs. Weight-based scheduling gives you proportional fairness. Quota-based scheduling gives you a hard upper bound. For a mixed tool runner, those are not interchangeable.

Use weights when you want tools to share idle capacity without imposing an abrupt ceiling. This is a good fit for bursty but legitimate workloads, especially when you care about latency and want to avoid artificial cliffs. A tool that parses a repository, compiles a small target, or processes a file set often benefits from proportional fairness because it can finish quickly when the machine is quiet and politely yield when siblings become busy.

Use quotas when you need a strict containment boundary against opportunistic CPU abuse. Cryptomining, infinite loops, and adversarial search workloads can be slowed into harmlessness if their CPU window is short enough. Quota is a stronger control, but it can introduce tail-latency spikes and confusing slowdowns for benign tasks. If the window is too tight, a bursty tool may appear to stall even though it is simply waiting for the next period. That creates poor ergonomics for developers and can make an otherwise correct sandbox feel unstable.

A practical rule is to reserve quota for hard abuse resistance and use

weight for fairness among otherwise trusted or predictable jobs. In a multi-tenant runner, you can combine them: assign a modest weight so the tool participates fairly in contention, then enforce a quota if the host cannot tolerate worst-case CPU consumption. The right threshold depends on workload shape, not just on security policy. A short-lived static analysis job and a long-running model post-processing job need different CPU envelopes even if they share the same trust level.

### **Memory control: soft protection, throttling, and failure**

Memory is the most subtle resource to scope because the failure modes are both operational and user-visible. `cgroups v2` gives you three distinct levels of control that map well to sandbox design. `memory.low` is a best-effort protection. `memory.high` is a throttling boundary. `memory.max` is a hard ceiling.

`memory.low` tells the kernel that a workload's working set deserves some protection during reclamation. This is useful when you have a critical sibling service or an interactive task that should keep a responsive footprint under pressure. It is not a reservation in the strict sense, and the system may still reclaim below it if pressure is severe enough. Think of it as a preference that improves survivability.

`memory.high` is where abuse resistance begins to bite. When a cgroup crosses `memory.high`, the kernel slows allocation and forces the workload to pay for its own growth through direct reclaim and throttling behavior. That is often preferable to a hard kill because it gives benign tasks a chance to settle while still preventing unbounded expansion. For bursty tools, `memory.high` is a good place to set the line between normal operating range and obvious overuse.

`memory.max` is the final boundary. Exceed it, and the kernel will invoke the OOM path for that cgroup rather than letting memory pressure spill outward. This is the knob you use when the platform must fail closed. If a tool can consume more than its declared envelope, you want that

failure to stay local. The downside is obvious: the job can die abruptly, and the agent may see a partial result or no result at all. That is why the section distinguishes *memory safety* from *memory survivability*. The first protects the host. The second determines whether the workload still completes.

For agent-invoked tools, memory limits should match expected behavior rather than ideal behavior. A parser that usually uses 200 MiB but occasionally spikes to 600 MiB may need a `memory.high` near the spike and a `memory.max` above it if you want graceful degradation. If the task is untrusted or the input is adversarial, a tighter hard cap is justified. When the payload is likely to leak memory or load large native dependencies, you should assume the declared footprint is a lower bound, not a ceiling.

### **Process fan-out and fork-bomb resistance**

`pids.max` is one of the most important abuse-resistance controls because process count is a cheap weapon. A workload that forks repeatedly can exhaust scheduler attention, PID space, and memory for task structs long before it consumes much CPU. That makes fork bombs an especially awkward failure mode: they are fast, noisy, and disruptive even when each child does almost nothing.

Setting `pids.max` constrains how many tasks can exist in the cgroup at once. For untrusted tools, this is usually a small integer, not a generous default. The exact value depends on the runtime model. A simple command-line tool may need only a handful of helper processes. A build or language runtime may need significantly more. What matters is that you choose a ceiling based on observed concurrency, not on the worst case of a general-purpose container image. Blindly inheriting large defaults leaves the system vulnerable to process storms that are easy to trigger and hard to recover from cleanly.

The control is also operationally useful for detecting misuse. If a work-

load repeatedly hits `pids.max`, that is a signal that the tool is either spawning too much work or being driven into pathological behavior by its input. In both cases, the failure is local and attributable. Without the limit, the host may simply become sluggish in ways that are difficult to trace back to one run.

### **I/O control and storage pressure**

I/O abuse is less dramatic than CPU exhaustion but often more expensive to the rest of the platform. A tool that streams large outputs, scans huge trees, or repeatedly writes small files can dominate device queues and slow unrelated workloads. `cgroups v2` exposes I/O throttling so you can limit throughput or control the amount of pressure a sandbox can impose on shared storage.

The key operational distinction is that I/O control is about fairness and containment, not filesystem permissions. A read-only mount can still issue heavy reads. A writable scratch directory can still generate enough writes to degrade the machine. If your tool runners share disks with latency-sensitive services, I/O ceilings are part of the sandbox contract. They prevent one run from turning into a storage-wide slowdown.

I/O limits tend to be workload-specific and device-specific. You do not set them because they are fashionable; you set them when the host has shared storage characteristics that can be harmed by one runaway job. Build systems, archive unpackers, and content indexers are common examples. The tuning signal is straightforward: if the platform sees high device utilization or increased latency during tool execution, you need either tighter I/O control or stronger queue isolation.

### **The launch-and-observe loop**

The most reliable way to size cgroup policies is to start from a declared tool shape and then validate it with measurement. A good runner keeps

a compact budget model for each tool class: expected CPU intensity, memory footprint, process concurrency, I/O pattern, and tolerated latency. That model is then translated into cgroup settings before execution.

A simple operational loop looks like this:

- classify the tool by expected resource shape
- create a dedicated leaf cgroup
- enable the needed controllers
- set CPU, memory, pids, and I/O budgets
- attach the process before exec
- observe pressure and OOM activity
- adjust budgets from measured behavior

You want to attach the workload to a leaf cgroup, not a busy interior node, because v2 expects clean process placement when domain controllers are active. That structural rule is easy to violate if you treat the hierarchy like a generic directory tree. In practice, each workload gets its own leaf, and parent cgroups express policy only for their descendants.

Observation matters as much as configuration. A workload that frequently touches `memory.high` may be underprovisioned or may have an input distribution you did not anticipate. A job that never touches its CPU quota may be overprotected relative to its peers. A repeated `pids.max` breach may indicate an accidental recursion or a malicious fork storm. Cgroups are effective only when you close the loop between policy and measured pressure.

**Trade-offs: strictness, fairness, and ergonomics**

Tight cgroup limits improve abuse resistance but can hurt usability. Loose limits preserve ergonomics but weaken tenant isolation under load. You have to choose where to sit on that spectrum.

Strict enforcement is appropriate when the cost of overload is high or the workload is untrusted enough that denial of service is the main concern. In those cases, it is better to fail the task quickly than to let it compete with the rest of the platform. This is common for public-facing tool execution, multi-tenant automation, and workloads that process adversarial input.

Fairness is better when workloads are mostly benign but bursty. In that regime, proportional sharing and soft memory protection keep the system responsive without imposing arbitrary cliffs. This reduces the number of false denials and makes the platform feel less brittle to the people using it.

Developer ergonomics are not a luxury. If the limit model is too opaque, people will overfit to it, work around it, or stop trusting it. That is why a runner should report which cgroup policy was violated and which resource crossed the line. “Killed by OOM” is less useful than “memory.max exceeded at 768 MiB after 14 seconds.” Good observability turns a hard boundary into actionable feedback rather than a mystery failure.

The common anti-patterns are easy to spot. Copying generic container defaults without measuring your own workload shape is usually wrong. Setting CPU quotas without testing burst behavior creates latency cliffs that look like bugs. Omitting `pids.max` leaves a cheap denial vector open. Treating `memory.max` as a harmless guardrail ignores the fact that it changes failure mode, not just capacity. Each of those mistakes comes from assuming that resource limits are plumbing rather than policy.

### **What cgroups v2 does not do**

cgroups v2 is strong at containment, but it is not a complete sandbox boundary. It does not decide which syscalls are allowed, which files are readable, or which kernel attack surfaces a workload can reach. It does not replace namespaces, filesystem design, or later syscall filtering. It also does not eliminate the shared-kernel risk that comes from running hostile code on the same kernel as everything else.

That limitation is not a flaw in cgroups. It is the right boundary for the tool. Cgroups answer a specific question: how much can this workload consume before it harms the rest of the system? For agent-invoked tools, that question is just as important as visibility and privilege. If you leave it unanswered, a compromised or simply misbehaving tool can still wreck availability even if it cannot escape its filesystem or network view.

A well-shaped sandbox therefore uses cgroups v2 as the resource layer beneath the namespace layer. Namespaces tell the tool what it can see. Cgroups tell it how much of the machine it can spend while doing so. When you align those budgets with the declared scope of the tool, the platform stops treating resource use as an accident and starts treating it as part of the permission model.

### **3.3. Designing Filesystem Views for Untrusted Tool Workloads**

A tool that only needs “one directory” can still become a liability if you hand it the host’s ambient filesystem. The directory may sit next to credentials, caches, socket files, or mutable state the tool should never learn about. Worse, once the process can traverse that ambient tree, a nominally local task can start behaving like a general-purpose host process with partial access to everything. The practical job here is to make the mount layout itself express least privilege: visible inputs, bounded

scratch space, a controlled export path, and nothing else.

The simplest useful pattern is a filesystem contract, not a permission afterthought. You decide which paths the tool may read, which paths it may write, and which results you are willing to accept back. Then you construct a mount topology that makes those decisions true at runtime. A minimal launch often looks like this:

```
mkdir -p /run/tool/root
mkdir -p /run/tool/scratch
mkdir -p /run/tool/out

mount --bind /opt/tool-base /run/tool/root
mount -o remount,ro,bind /run/tool/root

mount -t tmpfs tmpfs /run/tool/scratch
mount --bind /data/input /run/tool/root/input
mount --bind /run/tool/out /run/tool/root/out
mount -o remount,rw,bind /run/tool/root/out
```

That sketch is not complete sandboxing, but it shows the shape of the problem. You mount a base view, keep it read-only, attach a scratch area for transient writes, and expose a single output path. The tool sees a small filesystem whose topology matches its declared scope. Everything else is absent, not merely discouraged.

### **Start from a minimal view, not a large one with extra rules**

The most reliable filesystem designs begin with absence. If the process does not need a path, do not mount it. If it does not need a writable tree, make the root read-only and give it an explicit writable subtree instead. That structure is easier to reason about than a mostly writable root with ad hoc exclusions because the visible surface area is small and enumerable.

Read-only roots are useful even when the tool needs writable state elsewhere. Many workloads only need one of three writable patterns: ephemeral scratch, an isolated workspace for user inputs, or a narrow export directory for results. Ephemeral scratch belongs on ‘tmpfs’

when the data is transient and should vanish with the process. That choice keeps intermediate state off persistent storage and simplifies cleanup. An isolated workspace suits tools that transform supplied input into derived artifacts but should not mutate the inputs in place. A dedicated export directory is appropriate when the tool must emit results back to the agent or to a downstream pipeline, but you want to restrict what gets written and how it is collected.

The design mistake is to let writable state accrete around the workload. Once a tool sees a mutable root, it can start caching in surprising places, creating lockfiles in shared directories, or leaving behind state that affects later runs. That undermines both reproducibility and containment. A better rule is that each writable area should have a narrow purpose and a clearly owned lifecycle.

#### **Mount namespaces give you a private topology, not a finished policy**

A mount namespace is the control boundary that makes filesystem layout expressive. When you enter a new mount namespace, you inherit the parent's mount tree. That inheritance is where many unsafe layouts begin: the tree is not empty, and inherited propagation modes can still link host events to the sandbox. If you mount into that tree without changing propagation semantics, the sandbox can receive unrelated mount changes from the host, or its own mount changes can leak back out.

The standard first step is to make the mount tree recursively private or slave before any further manipulation. That cuts propagation links and turns the inherited tree into something you can reshape safely. Only then should you start building the visible filesystem the tool will use. This order matters because mount operations are stateful. If you remount or bind in the wrong order, you may preserve reachability to paths you intended to hide, or create views that become inconsistent

under later host activity.

Mount namespaces solve topology, not path correctness. They decide which mount points are visible, but they do not stop a process from resolving a path in an unsafe way inside those visible points. A private mount tree with careless path handling is still vulnerable to symlink tricks, traversal bugs, and time-of-check/time-of-use races. The filesystem view and the path-resolution policy are separate problems.

### **Bind mounts let you expose exactly the subtrees you mean**

Bind mounts are the practical workhorse for tool sandboxes because they let you expose a host directory at a controlled location without copying data. Used carefully, they are ideal for inputs, work directories, and export paths. Used carelessly, they become a fast path to ambient host visibility.

You should treat bind mounts as a mapping operation with a clear trust boundary. A read-only bind from an input directory is a common way to make source material available without giving the tool write access. A read-write bind should be reserved for a workspace that belongs to the sandbox instance, not to the host generally. If you need a deep subtree, use a recursive bind only when you understand what other mounts live beneath it. Recursive binds can drag in state that was not obvious from the top-level path. A subtree that looks harmless may contain device nodes, sockets, or mounts from other services.

Read-only remounting is also easy to misunderstand. A read-only bind helps, but it is not a magical freeze on all possible mutation. The mount flags apply to the mounted view, not to every underlying representation of the data. If other parts of the system still reference the source path with different permissions, you have not made the data immutable globally. For a sandbox, the safe rule is to expose only the specific path you mean to expose and to avoid assuming that a single read-only remount protects every route to the same bytes.

#### **Scratch space should be explicit and disposable**

Transient computation needs somewhere to write, but that somewhere should not be the host's persistent filesystem by default. 'tmpfs' is the cleanest choice for short-lived scratch because it gives the process a writable area that is easy to discard and hard to confuse with durable state. If a tool unpacks archives, stages temporary data, or emits intermediate results that are not worth keeping, 'tmpfs' keeps that traffic local to the run.

The capacity of scratch space should reflect expected use, not convenience. Too little space produces mysterious failures deep in the workload. Too much space invites the process to spill large temporary files that increase memory pressure or compete with other runs. A good scratch policy is sized from workload shape: file count, temporary working set, and peak intermediate artifact size. If the workload's temporary usage is unpredictable, separate scratch into one area for the tool and another for the runtime's own bookkeeping so that cleanup remains deterministic.

Scratch space is also part of the threat model. If a tool can write into a shared host directory, it can plant files that affect later jobs or leak data through naming patterns and timestamps. A dedicated scratch mount makes those artifacts local to the sandbox and easier to discard as part of teardown.

#### **Overlay and layered views solve convenience, not privilege**

Overlay filesystems are useful when you want a writable façade over an immutable base. That is common for tool runners that need a standard runtime image plus a writable layer for generated files or package caches. The merged view can make a tool think it is operating on a normal filesystem while the runtime preserves a clean base beneath it.

That convenience comes with edge cases. Copy-up behavior can sur-

prise applications that expect writes to update a lower layer in place. Whiteouts and redirect semantics can affect how deleted files appear. Option compatibility varies across kernel versions and backing filesystems. If you use overlay, you need to test the exact behavior of the combinations you plan to deploy, not just the nominal feature. Some overlays fail when prerequisites such as xattrs or export properties are missing; others behave differently depending on whether the upper and lower layers live on the same backing store.

Overlay is not a substitute for a proper writable policy. It gives you a merged filesystem view, but it does not decide which paths should be mutable in the first place. The safe design pattern is to make the base layer immutable, place only narrowly scoped writable content in the upper layer, and keep the export path separate from the workspace so that outputs do not silently mix with inputs.

### **Path safety and mount safety are different problems**

The filesystem view can be perfect and the tool can still escape its intended directory by resolving paths badly. This is why trusted code that handles untrusted paths needs explicit resolution rules. Mount namespaces shape the visible tree; path-resolution controls constrain how a particular lookup traverses that tree.

The distinction matters most when user input names a file, directory, or output path. A malicious path can use ‘..’ components, symlinks, or mount-point tricks to reach outside the intended root unless the resolver blocks that behavior. That is why ‘openat2’ is so valuable for sandbox brokers and file mediators. It lets you ask the kernel to resolve paths with explicit constraints rather than reproducing pathname policy in user space.

A simple example shows the intent:

```
int fd = openat2(dirfd, "inputs/report.txt",
                &(struct open_how){
                    .flags = O_RDONLY,
```

```
.resolve = RESOLVE_BENEATH |  
          RESOLVE_NO_SYMLINKS  
}, sizeof(struct open_how));
```

That is not a full broker, but it illustrates the model. You anchor the lookup to a directory file descriptor and ask the kernel to reject escapes below that anchor and reject symlink traversal. When a tool is allowed to name a file inside a sandboxed workspace, those constraints are far safer than validating a string path in application code and then hoping the filesystem does not change before the open call completes.

‘RESOLVE\_IN\_ROOT’ is useful when you want a lookup to behave as if a directory were the process root without actually changing global process state. That can simplify brokers that must serve many sandboxes at once. The key point is that you should treat pathname policy as a kernel-enforced resolution rule, not as a convention layered on top of a permissive open call.

### **Construct export paths as a controlled data channel**

Output handling deserves the same care as input handling. If you let a tool write directly into a shared host tree, you create a covert channel for leakage and a direct path for accidental corruption. A better pattern is to collect outputs into a single sandbox-owned export directory, then validate and copy them out after the run.

That export directory should be the only path the agent can see when it asks for results. The sandbox can write there, but it should not have access to the broader host workspace where the agent might keep credentials, transcripts, or cached state. After execution, a separate trusted component can inspect the export directory, filter filenames, verify size and type constraints, and move only approved artifacts into the agent-visible store.

This separation matters because output can hide more than bytes. File names can encode state, symlink targets can redirect future readers,

and directory structures can reveal internal processing details. A controlled copy-out step gives you a place to normalize names, reject unexpected object types, and apply size limits. It also keeps the tool from using the export path as an ambient write target to probe the host filesystem.

### **Avoid the common mount-pattern failures**

A few filesystem mistakes recur in sandbox design because they are convenient and subtly wrong. The first is exporting a whole workspace “for convenience” when only one directory is actually needed. That expands the attack surface and makes it much harder to tell which host data the tool can observe. The second is trusting user-controlled symlinks inside an input tree. If the tool or broker follows those links without restriction, the visible tree stops being meaningful. The third is assuming that a read-only flag at one mount layer makes the underlying data impossible to mutate. It does not. The correct guarantee comes from the entire mount graph and from who else can reach the source path.

Another common failure is to build the filesystem view before the mount namespace has been made private. That leaves propagation links intact and can leak mount events into or out of the sandbox. A similar issue appears when a sandbox attaches scratch or output paths directly under a host directory instead of under a dedicated workspace root. That choice is easy to implement and hard to audit because it hides the sandbox’s own writes among unrelated files.

These problems are not theoretical. They show up in tool runners that start from a general-purpose container image and only later try to “tighten” the filesystem with a few remounts. The safer design is to define the mount tree from the start as if the tool were the only tenant of that namespace.

### **A practical construction flow**

The most robust filesystem setup follows a small number of steps:

- enter a new mount namespace
- make mount propagation private
- mount an immutable base root
- bind only the required input trees
- create tmpfs for transient writes
- attach a single export directory
- remove paths the tool does not need
- launch the payload

That order gives you a mount topology that is both legible and enforceable. The immutable base establishes the environment. The bind mounts add the few host paths the task needs. The scratch area absorbs temporary writes. The export directory gives you a controlled egress point. Everything else stays absent, which is the strongest possible statement of least privilege in a filesystem.

If the tool needs more complex layering, add it deliberately rather than by inheritance. Use overlay when you need a writable façade over a stable base. Use separate workspaces when multiple paths need different trust or mutability properties. Use path-resolution constraints whenever user input names a path. The combination is what keeps the layout honest.

A well-constructed filesystem view does more than prevent obvious leaks. It also makes the tool's behavior easier to reason about, because every directory the process sees has a purpose. That clarity pays off later when you need to debug an unexpected write, explain why a path was rejected, or prove that a result could not have come from anywhere except the sandbox-owned export tree.

### **3.4. Shared-Kernel Risk Boundaries and What These Primitives Cannot Solve**

A carefully built sandbox can still fail in the only way that matters most: the tool runs on the same kernel as the rest of the platform. Namespaces, cgroups, and filesystem shaping reduce visibility, authority, and resource abuse, but they do not change that shared-kernel fact. That is the boundary to keep in view. If the workload's threat model includes kernel escape, cross-tenant interference, or exploitation of kernel subsystems reachable from untrusted code, Linux primitives are a strong first line and not a final boundary.

A useful launch stack therefore begins with a simple acknowledgement: the isolation layers are real, but they are not independent security domains. You can reduce the reachable surface, constrain the resource blast radius, and remove ambient filesystem and network access, yet the process still exercises the host kernel through syscalls, device interfaces, protocol parsers, and shared implementation paths. A tool that is correctly sandboxed can still crash the kernel if it reaches a vulnerability in a reachable path. That is not a failure of configuration; it is a property of shared-kernel isolation.

#### **Residual risk starts at the syscall boundary**

The first remaining exposure is the syscall surface. Even after you isolate namespaces and cap resources, the process can still invoke a broad set of kernel APIs unless something else narrows them. That API surface includes file operations, memory mapping, signal handling, process control, network access, and a long tail of less common interfaces. The sandbox may ensure that those calls operate on a narrow view of the host, but the calls themselves still execute in the host kernel.

That distinction matters because many kernel bugs are reachable through otherwise legitimate interfaces. A tool that merely reads

files, maps memory, or manipulates sockets can still trigger bugs in parser code, filesystem code, or protocol handlers. The sandbox can stop the tool from seeing the whole machine, but it does not stop the kernel from parsing its inputs. This is why syscall filtering becomes a separate defensive layer later in the book. Namespace isolation reduces what the calls can affect; syscall policy reduces which calls can happen at all.

When you reason about residual risk, ask which kernel subsystems the workload can still reach. A Python interpreter loading native extensions, a media parser handling attacker-crafted data, and a build tool invoking many filesystem and memory syscalls all present different reachable surfaces even if they run in identical namespaces. The security question is not whether the sandbox is present, but which code paths in the kernel remain reachable from the workload.

### **User namespaces improve usability and expand attack surface**

User namespaces are often the enabling mechanism for rootless tool execution, but they do not come for free. They scope privileges so that a process can act with local authority inside the namespace hierarchy while remaining unprivileged on the host. That makes rootless sandboxing practical, especially when you need to create mounts or adjust ownership semantics without running the launcher as host root.

The trade-off is that user namespaces expose privileged kernel code paths to less-trusted callers. That is the part operators sometimes underweight when they treat user namespaces as pure hardening. They are not. They are an attack-surface trade-off that buys operational flexibility and defense-in-depth at the cost of making more kernel logic reachable from unprivileged processes. If the host kernel has a flaw in a code path that user namespaces can reach, the namespace does not protect you from it.

That does not make user namespaces a bad choice. It makes them a deliberate one. For least-privilege execution cells, the gain is substantial: you can run many sandboxed tasks without host-wide privilege, and you can make mount and file-handling semantics precise. For hostile code with a kernel-exploitation objective, however, user namespaces should be treated as a powerful convenience layer, not a complete safety boundary. Their value is real, but it lives inside the shared-kernel trust problem rather than outside it.

### **Resource limits reduce blast radius, not exploitability**

Cgroups v2 closes off a different kind of harm: exhaustion. It stops a tool from taking all of the CPU, memory, I/O, or process table. That is essential in multi-tenant tool runners, but it does not change exploitability. A bounded workload can still target an unbounded kernel vulnerability. In fact, many exploit attempts are entirely compatible with low resource footprints because they need only a small number of carefully constructed syscalls.

That is why resource governance and code-execution safety must be kept separate in your mental model. A workload that respects its `memory.max` can still exploit a filesystem bug. A workload that stays within `pids.max` can still trigger a kernel panic. A CPU quota can slow abuse, but it cannot make a bug disappear. Limits are for containment of consumption and denial of service; they are not a substitute for narrowing the set of kernel behaviors the workload can invoke.

This distinction also affects incident response. If a cgroup hits memory or pids ceilings, you can usually attribute the failure to workload shape. If the host kernel crashes or a sibling workload is affected despite strict cgroup limits, the likely explanation is not resource abuse but shared-kernel reachability. The containment model must explain both.

### **Filesystem isolation does not eliminate parser risk**

A minimal mount layout is a major security improvement, but it still leaves the filesystem stack as a live attack surface. The process can read, write, and traverse only what you expose, yet every exposed path is still parsed by kernel code. Overlay behavior, bind mount behavior, symlink resolution, inode metadata handling, and path lookup itself remain active implementation paths. A neatly confined tree is not a dead tree.

That matters because filesystem bugs are often reachable from ordinary operations. Opening a file is ordinary. Listing a directory is ordinary. Renaming, unlinking, and mapping files are ordinary. If the kernel has a flaw in the code that makes those operations work, a well-contained workload can still reach it. The mount namespace and path-resolution constraints from the previous sections reduce the accidental surface, but they do not transform the filesystem into an inert object.

The same logic applies to exported artifacts. A controlled export path is safer than a shared host directory, but the output still travels through the same filesystem machinery as any other write. If the workload can generate malicious file structures or pathological metadata, the kernel still has to process them. That is one reason to validate and normalize outputs in a trusted copy-out stage rather than handing the export tree directly to a broader system.

#### **Shared-kernel features can cross sandbox boundaries**

Research on shared-kernel features in containerized environments has shown that some mechanisms can observe or influence workloads outside a namespace boundary. The exact feature set evolves, but the lesson is stable: the kernel contains components whose effects are not perfectly partitioned by namespaces alone. If a feature is global, privileged, or designed for host-wide observability, then it may remain visible or influential across sandboxes even when the sandbox itself is otherwise well built.

That observation is especially relevant for high-leverage kernel features that were not designed as per-workload isolation primitives. Features in this class can create side channels, interference channels, or direct management channels that cut across nominal containment boundaries. The operational response is not to assume that all namespaces and cgroups fully sever the system. The response is to audit which kernel facilities remain enabled, which are reachable from untrusted code, and which must be denied or moved into a stronger boundary.

The broader systems framing is straightforward: a monolithic kernel is one trust domain unless you add stronger fault isolation around it. Hardening can reduce exploitability. It can reduce the number of reachable interfaces. It can make compromise harder. It cannot, by itself, create a separate kernel.

### **When Linux primitives are enough**

There are workloads for which namespaces, cgroups, and filesystem controls are the right answer. Low-risk automation, internal build tasks, trusted data transformation, and short-lived single-tenant jobs often fall into this category. In those cases, the residual risk is acceptable because the workload is not expected to face determined adversaries, and the cost of a stronger boundary would outweigh the benefit.

The decision signals are practical. If the workload has narrow inputs, low tenant asymmetry, modest incident impact, and no reason to load untrusted native code, the shared-kernel model is often good enough. You still want the isolation layers, because they reduce accidental exposure and limit blast radius. You may not need to pay the latency, memory, and operational overhead of a stronger boundary.

This is where the earlier layers earn their place. They turn a host into an execution cell that is good for most controlled workloads. They keep the filesystem small, the process tree local, the network mediated, and

the resource budgets bounded. For a large class of automation, that is exactly the right answer.

#### **When you should escalate to stronger boundaries**

The justification changes when the workload is hostile, multi-tenant, or likely to process adversarial inputs. If the cost of a kernel escape is high, the shared-kernel model becomes difficult to defend on risk grounds alone. That is especially true when the tool parses attacker-controlled archives, executes code from third parties, loads native extensions, or handles data that can be shaped to target kernel edge cases.

The decision criteria are concrete. You should escalate when tenant trust is asymmetric, when the reachable kernel surface is large, when the expected payoff of an escape is high, or when the operational consequence of a mistake is unacceptable. In those cases, namespaces and cgroups are still useful, but they become one layer inside a larger boundary rather than the boundary itself.

Stronger boundaries buy two things that Linux primitives cannot. First, they reduce the kernel attack surface that the guest workload can reach. Second, they limit the blast radius if a kernel bug is triggered anyway. The trade-off is cost: more boot latency, more memory overhead, a more complex management plane, and a different operational model for artifacts, networking, and observability. The right answer depends on how much those costs matter relative to the security consequence of compromise.

#### **MicroVMs and hardware-backed isolation as the next layer**

A microVM is the most common next step when shared-kernel risk is too high. It gives the workload a separate guest kernel and a smaller host-facing device model while keeping startup and density better than a full general-purpose VM in many deployments. The important property is not branding or implementation detail; it is kernel separation.

The guest no longer shares the host kernel's implementation faults directly.

That change alters the threat model substantially. A guest exploit now has to cross a hypervisor boundary or break out of the virtualized device model instead of simply reaching a host kernel code path. The attack surface does not vanish, but it changes shape in a way that is often materially safer for untrusted tool execution. For workloads that handle adversarial inputs at scale, that reduction is often worth the added complexity.

The cost remains real. MicroVMs introduce extra orchestration, guest image management, and usually more expensive cold starts than pure Linux isolation. They may also constrain some forms of filesystem sharing and live introspection. You pay for the stronger boundary with operational friction. The reason to pay is simple: if the workload is dangerous enough, shared-kernel efficiency is the wrong optimization target.

### **Decision framework for tool runners**

A sane policy separates workload classes by threat and consequence, not by technical fashion. One practical framework is to ask five questions before choosing the boundary:

- Does the workload process attacker-controlled or untrusted input?
- Would a kernel escape meaningfully affect other tenants or the host?
- Can the workload tolerate the latency and memory overhead of a stronger boundary?
- Does the workload need privileged kernel features that are hard to confine?

- Is the operational team able to maintain the more complex run-time path?

If the answer to the first two questions is yes, the case for stronger isolation grows quickly. If the answer to the last three questions is no, then Linux primitives may still be the only workable option, but you should acknowledge that you are choosing a lower boundary with a known residual risk. That honesty matters, because security reviews usually fail when the architecture claims more than the platform can actually deliver.

A useful way to document the decision is to tie each workload class to a boundary and a reason. For example, a repository formatter may run in namespaces plus cgroups because the input is trusted and the blast radius is small. A public code execution service may require a microVM because the input is hostile and the consequences of compromise are severe. The policy then becomes repeatable rather than ad hoc.

#### **What “defense in depth” means here**

Defense in depth is not a slogan; it is the recognition that each layer removes a different failure mode. Namespaces hide and remap visibility. Cgroups contain resource abuse. Filesystem layout removes ambient access. Syscall policy narrows kernel reachability. MicroVMs change the trust boundary altogether. No single layer solves the whole problem, and no layer should be credited with doing work it cannot do.

That layering is especially important for tool runners that execute untrusted code on behalf of users or agents. The right question is not whether Linux primitives are “secure” in the abstract. The right question is what residual risk remains after you use them correctly, and whether that residual risk is acceptable for the workload’s value and exposure. For many workloads, the answer is yes. For others, especially those designed to face hostile inputs, the answer is that shared-kernel

containment is a foundation, not a finish line.

## Chapter 4

# Containers as a Sandbox Boundary (and How to Harden Them)

*Containers are attractive precisely because they are easy to adopt, fast to start, and operationally familiar-but those same properties often tempt teams into mistaking packaging for isolation. This chapter shows how to use containers as a serious sandbox boundary for agent-invoked tools by defining their real security envelope, hardening defaults into policy, placing runtime components inside the trusted computing base, and designing denial paths that remain safe when the system is under attack or simply mis-specified.*

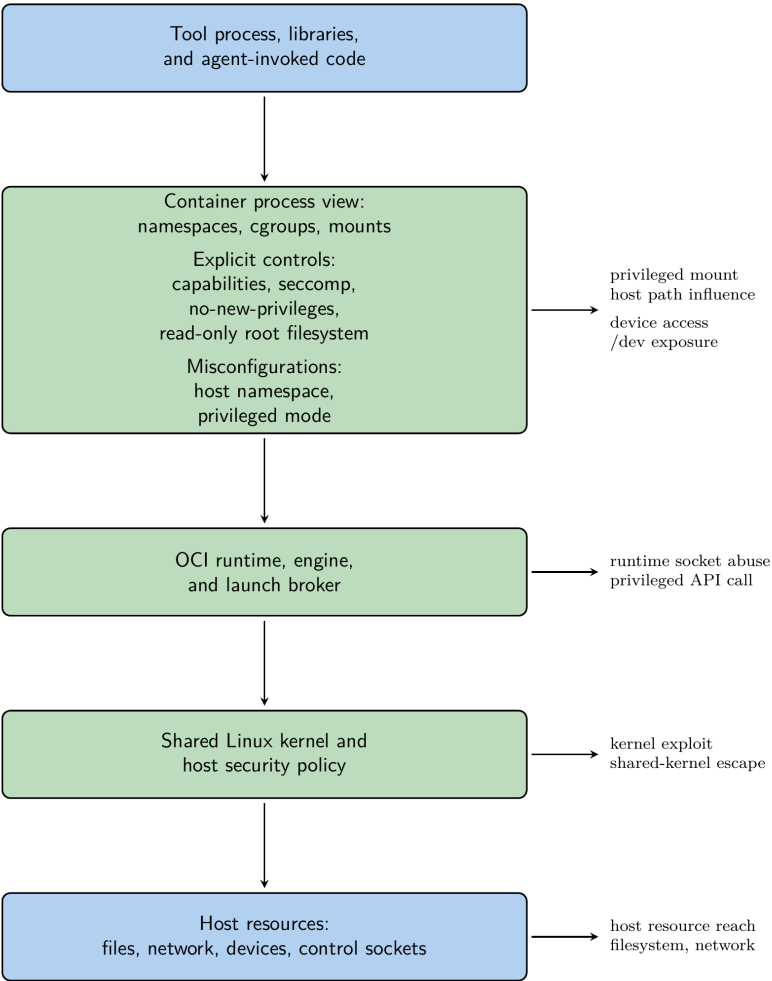
## 4.1. OCI Container Boundary Properties and Common Escape Classes

When you run an agent tool inside a container, the security question is not whether the process is “inside” something smaller than the host, but what the container boundary actually constrains. A container gives you a process view with selected namespace isolation, resource controls, and a curated set of host interfaces. It does *not* give you a separate kernel, nor does it automatically prevent every path from the sandbox back into the host. That distinction matters in agent systems because untrusted prompts, hostile files, and dependency chains all steer the tool toward code paths you did not explicitly write, and those paths often end at the container’s narrowest seams.

A concrete hardened launch makes the boundary visible:

```
docker run --rm -it \  
  --read-only \  
  --cap-drop=ALL \  
  --security-opt no-new-privileges \  
  --security-opt seccomp=default.json \  
  --tmpfs /tmp:rw,nosuid,nodev,noexec,size=64m \  
  --mount type=bind,src="$PWD/in",dst=/work/in,ro \  
  --mount type=bind,src="$PWD/out",dst=/work/out \  
  my-tool:latest \  
  analyze /work/in /work/out
```

That invocation is useful not because it is fashionable, but because it exposes the actual security envelope. The root filesystem is read-only, capabilities are stripped, privilege escalation is blocked, seccomp constrains the syscall surface, and only two narrow writable paths exist. If the tool tries to rewrite itself, load a helper into the image root, create new devices, or reach beyond the declared mounts, the platform should fail closed. The security boundary is therefore the composition of runtime policy, kernel enforcement, and host configuration, not the image layer alone.



The diagram is intentionally layered because the attack classes are layered. Some failures break out of the container view itself; others do not “escape” in the classic sense, but erode the boundary by making host resources reachable through a permitted interface. For agent workloads,

that distinction is operationally important. A prompt-induced misuse may never exploit a kernel bug, yet it can still coerce the tool into reading sensitive bind-mounted files, writing to an unexpected mount, or calling a privileged daemon socket on the host's behalf. The sandbox failed, even though the process remained caged in a technical sense.

### **What the container boundary actually guarantees**

An OCI container is defined by a runtime specification that turns an execution request into a constrained process tree. The core protection properties come from Linux namespaces, cgroups, mount settings, capabilities, seccomp, and related host security hooks. Namespaces narrow what the process can see; cgroups limit how much CPU, memory, and I/O it can consume; mount policy determines which paths are accessible and writable; capabilities control which privileged kernel operations remain available; seccomp filters the syscall interface; and no-new-privileges prevents later privilege gain through setuid-style transitions. In practical terms, a container gives you a process with a smaller view of the machine and fewer direct powers over it.

That smaller view is real and useful. A tool that only needs to transform files can often be isolated from the rest of the host by a read-only root filesystem, a couple of scratch mounts, a strict capability set, and a deny-by-default seccomp profile. If the tool is compromised, the compromise is then bounded by the mounts, syscalls, and identities you intentionally exposed. For agent tool execution, this is the right unit of isolation because the agent will often execute many short-lived tasks, each with different trust assumptions, and you need a boundary that is cheap enough to instantiate but strict enough to express least privilege.

The container boundary does *not* imply host independence. The runtime shares the host kernel, and the runtime stack itself must interpret the policy correctly. If the kernel is compromised, every container on that node is at risk. If the runtime misconfigures mounts or credentials,

the sandbox begins with more authority than intended. If a host interface remains reachable, the tool can ask a more privileged component to act for it. You should treat the container as a constrained execution environment, not as a miniature virtual machine.

### **Shared-kernel risk is the irreducible escape class**

The first and most fundamental escape class is kernel exploitation. Because containers share the host kernel, a vulnerability in the kernel's namespace, filesystem, network, or memory-management paths can break isolation for every container on that host. This is not a theoretical edge case; it is the reason container hardening must be paired with host patching, kernel feature selection, and node-level policy. No amount of image minimization changes the fact that the kernel mediates all container isolation.

In agent systems, kernel risk becomes more pressing because tools are frequently asked to handle malformed, adversarial, or simply huge inputs. A document parser, archive extractor, image analyzer, or language runtime embedded in a tool may exercise unusual kernel paths through file metadata, filesystem operations, or memory pressure. A hostile repository or attachment can therefore become an exploit delivery vector even when the agent never intended to run untrusted binaries. The relevant failure is not merely “the tool crashed”; it is that the shared kernel is the last line of defense between a malformed input and host compromise.

### **Capabilities widen the action vocabulary**

Linux capabilities are often treated as fine-grained privilege, but in a container they are just as dangerous as they are on the host if you leave too many in place. Capabilities such as `CAP_SYS_ADMIN`, `CAP_NET_ADMIN`, or `CAP_MKNOD` expand what the process may ask the kernel to do. They can enable mount operations, network reconfiguration, device node creation, namespace manipulation, and other host-

adjacent actions that collapse the sandbox from within.

For agent workloads, capability bloat is a common failure mode because the tool maintainer discovers one broken operation, adds a capability to fix it, and never revisits the broader privilege surface. That pattern is especially risky when the tool is driven by prompts or external content. A code assistant that can write to its own environment, adjust networking, or create device nodes has enough power to turn a narrow mistake into persistent host influence. The correct question is not “can the tool still run?” but “which privileged kernel operations must this tool be able to justify, and can any of them be mediated outside the container instead?”

### **Mounts define reachability, not just storage**

A bind mount is a policy decision with filesystem syntax. If you expose a host directory, you are not merely giving the container a place to write; you are exposing that path tree to whatever code the agent triggers. Read-only mounts still reveal content, filenames, timestamps, and sometimes application secrets. Writable mounts can become persistence locations, staging areas for helper binaries, or bridges into adjacent host-managed state. A broad mount such as the developer’s home directory or workspace root often carries more authority than its operator realizes because it aggregates source code, credentials, build artifacts, caches, and configuration files.

This is where agent-specific abuse becomes concrete. A prompt-injected tool may be convinced that it needs to “inspect nearby files” or “repair the workspace,” and the broad mount makes that instruction operationally real. A hostile repository can place symlinks, special filenames, or nested metadata in the mounted tree to steer the tool into touching paths outside the intended scope. A dependency installer can populate writable directories with executables or hooks that later runs will pick up. Once you treat the mount set as part of

the security policy, the choice becomes clear: mount the smallest path that satisfies the task, prefer read-only unless writes are essential, and separate input, output, and scratch spaces so that one path cannot masquerade as another.

### **Devices and special files break the abstraction quickly**

Device exposure is a particularly sharp boundary failure because devices often bypass the nice filesystem abstraction entirely. A container with access to `/dev` objects, GPU devices, or other special files may gain capabilities that are difficult to reason about from the application layer. Even when a device is not directly exploitable, it can widen the I/O surface, introduce driver-specific vulnerabilities, or enable side effects that persist beyond the container's lifetime.

Agent tool platforms frequently encounter this pressure through convenience requests: a model asks for GPU acceleration, access to local serial hardware, or a host-mounted socket that happens to appear as a file. Each of those asks should be treated as a change in the trust boundary, not as a performance toggle. If the workload genuinely needs device access, the resulting policy should be explicit, reviewed, and isolated to the smallest possible device set. If the workload does not need devices, the default device set should be empty or tightly curated. “Just let the container see the hardware” is not a sandboxing strategy.

### **Runtime sockets and control planes are privilege multipliers**

The container runtime stack is itself part of the trusted computing base. If a tool can talk to a daemon socket, orchestration API, or helper service that runs with elevated host privileges, the tool may not need a kernel escape at all. It can simply ask the privileged component to do something on its behalf: start a sibling container, mount a path, widen a namespace, or inspect a host resource. That is boundary erosion through delegation rather than escape through exploitation.

This class of failure is easy to underestimate because it often looks like ordinary automation. A build helper with access to a container engine socket may appear to be “managing containers” while actually obtaining host-level authority over mounts, images, or process launches. For agent systems, this is a classic overreach path: the model proposes a helpful fix, the tool is wired to comply through a socket or API, and the platform quietly converts a narrow request into an administrative action. The defense is to keep privileged sockets out of the sandbox unless the tool is explicitly a control-plane component, and even then to scope the operations to the minimal reviewed set.

### **Privileged mode and host namespaces erase the boundary**

Privileged containers and host namespace sharing are not subtle misconfigurations; they are boundary contractions. Privileged mode broadens the process’s operating system authority enough that many of the usual container constraints lose force. Sharing the host PID, network, IPC, or mount namespaces reduces the isolation that namespaces were meant to provide. In effect, the process is no longer operating in a sharply bounded container view; it is interacting with the host much more directly.

The agent-specific hazard is convenience pressure. If a tool needs to debug a service, access host networking, or inspect processes, someone may be tempted to attach it to host namespaces “just for this run.” That temptation is dangerous because agent workloads are repetitive and hard to audit manually. A temporary exception becomes a pattern, and the pattern becomes policy by habit. The right default is to avoid privileged mode categorically and to expose host namespaces only when you are deliberately building a host-integrated diagnostic tool, not a sandboxed agent worker.

### **How agent inputs turn into escape opportunities**

Agent systems amplify all of these escape classes because they create

many routes from untrusted text to powerful actions. Prompt injection can coerce a tool into reading or writing files it should ignore, invoking subcommands with broader effects, or requesting extra mounts. Hostile input files can trigger parser bugs, path traversal inside archive handlers, or unexpected behavior in third-party libraries. Dependency-triggered privilege expansion occurs when a tool installs or loads packages that execute code during setup, build, or import, and that code inherits the container's host reach.

The crucial point is that the agent does not need to be malicious for the risk to materialize. The tool can be following a plausible instruction from a user while the input data itself is adversarial. A code review assistant asked to analyze a repository may encounter a crafted build script; a document-processing agent may open a file that embeds an exploit chain; a data-extraction tool may be urged to "try a different path" and end up traversing an overbroad mount. Container boundaries must therefore be designed for confused-deputy behavior, not only for obviously hostile operators.

### **Isolation properties and security objectives are different questions**

Filesystem reach, device reach, network reach, and privilege escalation are policy questions layered on top of the container boundary. If a tool can reach a path, that does not mean it should; if it can open a socket, that does not mean it should contact arbitrary hosts; if it can access a device, that does not mean it should control hardware. The container boundary gives you the mechanism to constrain these dimensions, but the authorization model has to define the actual scope.

That separation is easy to lose when teams treat containerization as a one-step answer to sandboxing. A better mental model is to map each security objective to an explicit runtime control. Narrow filesystem scope becomes narrow mounts and a read-only root. Narrow execu-

tion scope becomes capability dropping and seccomp. Narrow escalation scope becomes no-new-privileges. Narrow device scope becomes a curated device set or none at all. The container is the enforcement envelope; the permission scope is the policy that decides what should be placed inside it.

### **Operational anti-patterns repeatedly weaken the boundary**

Several mistakes recur often enough that you should treat them as design smells. The first is assuming that “running as non-root inside the container” makes the workload low risk on the host. It does not, because the container may still have dangerous mounts, capabilities, sockets, or kernel attack surface. The second is using a minimized image as a substitute for sandboxing. Image size says little about the authority of the runtime configuration. The third is bind-mounting large host paths for convenience, which usually turns the container into a lightly disguised host process with extra ceremony. The fourth is allowing broad engine or runtime socket access to simplify developer workflows, then discovering that the tool can indirectly administer the host.

A fifth anti-pattern is letting every exception accrete in the launch script until no one can explain what the sandbox actually protects. For agent platforms, this is especially costly because the input surface is already variable and the invocation rate is high. If the boundary is not simple enough to inspect, it is not simple enough to trust. The launch specification should be readable, reviewable, and intentionally sparse.

### **What an acceptable container sandbox looks like**

A useful container sandbox for agent tool execution is narrow, predictable, and boring. It has a read-only root filesystem, a small set of writable scratch paths, no unnecessary Linux capabilities, no privilege escalation, no access to host control sockets, no host namespace sharing, and no device exposure unless the tool cannot function without

it. Its seccomp profile denies high-risk syscalls by default. Its runtime stack is treated as part of the trusted computing base and is patched, audited, and minimized accordingly. Its mounts and credentials are declared up front rather than improvised during launch.

That design does not eliminate risk; it makes the residual risk legible. You still share a kernel, so kernel bugs remain relevant. You still depend on the runtime, so parser and configuration bugs matter. You still need to reason about the tool's behavior under adversarial prompts and hostile inputs. But the container boundary now has a precise meaning: it constrains the process's view, limits its direct actions, and forces any stronger access to cross an explicit policy or runtime boundary rather than hiding inside convenience defaults.

## 4.2. Configuring Hardened Defaults for Container Tool Runs

A permissive container policy often starts as a convenience choice and ends as an implicit security architecture. One team adds a writable mount for debugging, another grants a capability to satisfy one tool, a third exposes a host socket because it is “only for automation,” and the platform gradually normalizes broad host reach. The safest way to prevent that drift is to make the hardened configuration the default and require explicit, reviewed exceptions for anything broader. That turns the permission-scope model into a launch-time contract: if the tool does not need a power, the container should not expose it.

A practical baseline looks like this:

```
docker run --rm -it \  
  --read-only \  
  --cap-drop=ALL \  
  --security-opt no-new-privileges \  
  --security-opt seccomp=default.json \  
  --security-opt apparmor=tool-runner \  
  --
```

```
--tmpfs /tmp:rw,nosuid,nodev,noexec,size=64m \  
--tmpfs /run:rw,nosuid,nodev,noexec,size=16m \  
--mount type=bind,src="$PWD/in",dst=/work/in,ro \  
--mount type=bind,src="$PWD/out",dst=/work/out \  
my-tool:latest \  
process /work/in /work/out
```

That command illustrates the policy shape you want to standardize. The container starts with a read-only root filesystem, no Linux capabilities, no privilege escalation, a syscall filter, optional AppArmor confinement, and only two bind mounts with narrow purposes. Scratch space exists only as ephemeral tmpfs. If the tool needs more than this, the exception must be justified in policy, not improvised during an incident.

### Start from deny-by-default

The most reliable hardened default is a deny-by-default posture. You want the runtime to refuse dangerous behavior unless the platform has explicitly allowed it for a particular tool class. In practice, that means four things: keep seccomp enabled, drop all capabilities unless you can defend a specific one, enable `no-new-privileges`, and make the root filesystem read-only. Those controls compose well because they fail closed in different ways. Seccomp blocks unexpected syscalls. Capability dropping removes privileged kernel operations from the action set. `no-new-privileges` prevents the process from regaining power through `setuid` or similar transitions. A read-only root prevents the tool from rewriting its own environment or planting persistent changes in the image layer.

This posture is especially valuable for agent workloads because the tool often executes under partial trust. A prompt can ask for behavior the operator never intended. A hostile input file can trigger an unexpected code path. A dependency can execute setup logic that inherits the tool's privileges. If the default sandbox already assumes that any writable path or privileged operation could be abused, the platform becomes

far less brittle when those inputs arrive.

### **Capability reduction is a policy decision, not a tuning knob**

Container capabilities are not a decorative hardening option; they define which privileged kernel operations remain available to the process. The correct baseline for most tool runners is `CAP_DROP=ALL`, with narrowly justified additions only where the tool cannot function otherwise. In Kubernetes terms, the Restricted security profile expresses the same principle: drop all capabilities and add back only a very small set when required, typically `NET_BIND_SERVICE` for low-numbered ports. That model is portable even outside Kubernetes because it captures the right operational instinct: start from zero privilege and build up only after the need is proven.

You should treat each capability request as a reviewable security event. A tool that needs to bind port 80 is different from a tool that needs to create devices, modify host networking, or mount filesystems. Some requests are simply incompatible with a safe sandbox and should be handled outside the container boundary, perhaps by a mediation layer or a dedicated host service. Others can be replaced with a narrower alternative. For example, a tool that only needs to listen on a local socket does not need broad network administration rights. Capability inflation is hard to reverse once it becomes a pattern, so the safe default is to make any capability addition visibly exceptional.

### **Read-only roots reduce both persistence and self-modification**

A read-only root filesystem is one of the highest-value controls in a tool runner because it narrows what compromised code can do after execution begins. It prevents the tool from mutating its own binaries, altering configuration files in place, or dropping helper executables into standard paths. That matters not only for malware-style persistence, but also for prompt-induced abuse. A model that is told to “fix the

environment” may otherwise reach for writes that silently change later runs. A read-only root forces that behavior to fail, which is exactly the signal you want.

The read-only root is not a complete answer by itself. Most tools still need temporary write locations for caches, lockfiles, sockets, and intermediate artifacts. The right pattern is to create those locations explicitly and keep them small. Use `tmpfs`s for transient state when possible, and mount output staging directories separately from input data. Avoid using the image root as scratch space and avoid making broad host directories writable just because the application expects a working directory. If a tool needs a large writable tree, consider whether that tree really belongs inside the sandbox or whether the task should be split so that only the minimum write path crosses the boundary.

### **Mounts should reflect declared need, not operator convenience**

Mount design is the point where a container becomes either a constrained workspace or a lightly disguised host process. Narrow, purposeful mounts preserve the boundary; broad bind mounts dilute it. You want to separate at least three roles: immutable input, mutable output, and ephemeral scratch. That separation makes access reviewable and simplifies incident analysis because you can later distinguish what the tool read from what it created.

A bind mount should usually be read-only unless the tool must produce results there. Even then, the writable location should be the narrowest possible subpath. Avoid mounting the workspace root when the tool only needs a handful of files. Avoid mounting home directories, parent directories, or control-plane data. A broad mount often contains credentials, local caches, build scripts, and hidden files that the tool did not need to know existed. In agent environments, where the invocation may be influenced by untrusted prompts or repository content,

overbroad mounts become an easy path from ordinary file access to host influence.

OCI runtime settings also include underused but useful path controls such as masked paths and readonly paths. These are worth adopting when the process should not even observe selected `procfs` or `sysfs` locations. Hiding or freezing those paths reduces accidental dependency on host-specific state and shrinks the set of host internals that tool code can inspect or misuse. When a path exists only because the host exposes it, ask whether the sandbox really needs to see it at all.

### **No-new-privileges closes a common escalation path**

`no-new-privileges` is one of the simplest and most effective default settings you can apply. It stops the process and its descendants from gaining additional privilege through mechanisms such as `setuid` binaries or file capabilities that would otherwise change effective privilege mid-run. In a tool runner, that matters because the container image may contain utilities you did not audit in detail, and the runtime may invoke helper binaries that were not part of the original threat model. Without `no-new-privileges`, a low-privilege process can sometimes cross into a more privileged execution context if a permitted binary is present.

This flag is not a substitute for capability reduction. It complements it. If you leave broad capabilities in place, the process already has too much power; `no-new-privileges` only prevents certain escalation forms. But when used together with a minimal capability set, a read-only root, and restricted mounts, it removes one of the most common paths by which a compromised tool broadens its own authority after launch.

### **Seccomp gives you deterministic denial behavior**

A hardened default should keep `seccomp` on, not as an optional opti-

mization but as a normal part of the boundary. Docker’s default profile is allowlist-based and uses a deny-by-default action for blocked syscalls, which gives deterministic permission-denied behavior rather than ambiguous failure. That matters operationally because the platform can distinguish “this tool tried something outside policy” from “the tool crashed for unrelated reasons.” For an agent system, predictable denial is preferable to a partial success that silently widens behavior.

Seccomp does not replace capability or mount policy; it constrains what the process can ask the kernel to do. A good default profile removes especially risky syscalls and leaves only the operations a typical tool truly needs. You do not need to write a profile from scratch for every tool. Instead, start from a conservative baseline and add back only after you have a concrete, reproducible need. This approach aligns with platform-wide policy because it gives you a stable starting point and a reviewable exception path.

### **AppArmor and seccomp solve different problems**

Where the host supports it, AppArmor is a useful complement to seccomp. Seccomp filters syscalls. AppArmor constrains resource access patterns such as which files, capabilities, or paths a process may interact with under the node’s policy. The two controls do not overlap perfectly, and that is their strength. A syscall filter can prevent dangerous kernel entry points from being used at all, while an LSM profile can narrow what the process can touch even when the syscall itself is legitimate.

For a tool runner, this means you can use AppArmor to tighten path access and seccomp to reduce the syscall vocabulary. If one control is bypassed or too coarse for a particular workload, the other still narrows the blast radius. The operational caveat is compatibility: AppArmor must be available and enabled on the node, and the profile must exist

where the runtime expects it. If the profile is missing, treat that as a deployment error, not as a reason to continue in a weaker mode.

### **Translate scope into concrete launch settings**

The cleanest way to avoid policy drift is to map each permission dimension directly into a runtime default. If filesystem scope is narrow, make the root read-only and expose only declared writable mounts. If device scope is empty, do not pass through host devices. If network scope is limited, avoid attaching unnecessary host network features. If privilege escalation is not legitimate for the workload, enable `no-new-privileges`. If the tool never needs a capability, do not add it. This translation is more reliable than post hoc hardening because it makes the intended scope visible before execution starts.

You can capture that mapping in a policy artifact that is diff-friendly and reviewed like code. A small example helps make the shape concrete:

```
tool_class: code-analyzer
filesystem:
  root_readonly: true
  write_mounts:
    - /work/out
    - /tmp
capabilities:
  drop: ["ALL"]
  add: []
security:
  no_new_privileges: true
  seccomp_profile: default.json
  apparmor_profile: tool-runner
devices:
  allow: []
mounts:
  masked_paths:
    - /proc/kcore
    - /proc/keys
  readonly_paths:
    - /proc/sys
    - /sys
```

That kind of artifact is useful because it keeps the policy readable. Operators can review the declared scope before the runtime compiles it into flags and mount settings. The same artifact can also feed automated validation, which reduces the chance that a supposedly hardened launch is silently degraded by a packaging mistake.

### **Platform baseline versus tool-specific exceptions**

A platform-wide default should be narrow enough that most tools inherit it without discussion. Exceptions should exist, but they must be explicit, traceable, and hard to justify casually. The decision criteria are practical: how often is the exception needed, how large is the blast radius if it is abused, how easy is the failure to detect, and whether a narrower mediation path exists outside the container. If a tool only needs an extra path or a single low-numbered port, prefer a targeted exception. If it needs privileged mounts, device access, or host namespace sharing, reevaluate whether it belongs in the same sandbox class at all.

This separation is important for agent platforms because exception pressure is continuous. A tool that fails under a hardened default will often prompt an operator to “just allow the thing this time.” If the baseline already has a formal exception path, that request becomes a reviewable change instead of an ad hoc workaround. The result is not zero exceptions; the result is that exceptions remain visible and bounded.

### **Use tmpfs and narrow overlays for writable state**

Many tool runners need only small, transient write areas. For those cases, tmpfs is often the right answer because it avoids durable host writes and naturally disappears at container exit. Typical candidates are /tmp, /run, lock directories, and short-lived caches. If the workload needs more space than a small tmpfs can provide, you can still use an overlay or a separate writable mount, but keep the writable surface area as small as possible and do not generalize that mount into a broad

workspace.

This design reduces the chance that prompt-induced or dependency-triggered behavior can stash persistent state where later runs will reuse it. It also makes cleanup deterministic. If the writable state is ephemeral, a failed or suspicious run leaves less residue on the host, which simplifies both incident response and routine operations.

### **Validate the baseline as part of startup, not after failure**

A hardened default is only useful if the runtime can actually enforce it. Missing kernel features, absent LSM profiles, or unsupported storage behavior should fail the launch, not silently weaken it. That means your launcher should verify the presence of the seccomp profile, the AppArmor profile if you rely on one, the necessary mount capabilities, and the expected filesystem behavior before admitting work. If the node cannot satisfy the baseline, reject the run and surface a precise operational error.

That startup validation is particularly important in mixed fleets where distributions, kernel versions, and container engines differ. Rootless support, overlay filesystem behavior, and even capability handling can vary enough to affect the real sandbox strength. When the platform treats the baseline as a requirement rather than a preference, you avoid the worst outcome: a workload that appears hardened in policy but executes with a weaker boundary on a subset of nodes.

### **Design the default to be intentionally annoying to bypass**

A hardened container profile should make unsafe shortcuts inconvenient. A developer should not be able to add a broad bind mount, a capability, or a privileged socket without creating a visible policy diff. A tool should not silently gain write access because someone changed the working directory. A runtime should not fall back from a missing security profile to an open configuration. The friction is deliberate. If

bypassing the sandbox is easy, it will happen under delivery pressure.

That friction is not anti-usability; it is how you keep the baseline honest. A platform that enforces a narrow default and forces exceptions through review can still support real workloads, but it does so on top of an explicit contract. For agent tool execution, that contract is the difference between a constrained process environment and a container that merely looks safe while quietly inheriting the host's privileges.

### 4.3. Choosing Rootless Containers for Agent Tool Runners

A rootless setup changes the trust question at the point where the sandbox is created, not just where it runs. If the launcher or daemon itself holds host root, then a compromise in the control path can widen into host-level authority even when the container payload looks constrained. Rootless execution reduces that concentration of privilege by moving the daemon and the workload into an unprivileged user context. For agent tool runners, that can materially shrink the blast radius of a control-plane bug, an API misuse, or a support script that was never meant to handle hostile input.

A minimal rootless launch shows the shape of the setup:

```
dockerd-rootless-setup tool.sh install
export DOCKER_HOST=unix:///run/user/1000/docker.sock
docker run --rm -it \
  --users=host \
  --read-only \
  --cap-drop=ALL \
  --security-opt no-new-privileges \
  --mount type=bind,src="$PWD/in",dst=/work/in,ro \
  --mount type=bind,src="$PWD/out",dst=/work/out \
  my-tool:latest
```

The command is intentionally plain. It shows that rootless is not a mag-

ical sandbox mode; it is an execution model in which the container stack itself no longer relies on host root for its normal launch path. That matters because the control path is often the highest-value target in an agent platform. If the tool runner, broker, or daemon can be tricked into doing something unintended, you want the damage to be bounded by an unprivileged user context rather than by a root-owned service.

### **What rootless actually changes**

Rootless execution uses user namespaces to map container identities onto unprivileged host identities. Docker rootless mode and Podman rootless mode both depend on subordinate UID and GID ranges, along with kernel support for user namespaces. In practice, that means the process inside the container can appear to run as root while the host sees an unmapped or subordinate identity. The important distinction is that root inside the container is no longer host root by default.

That makes rootless stronger than the older `usersns-remap` pattern in an important failure scenario. With `remap`, a rootful daemon still sits on the host and mediates container lifecycle operations with elevated privilege. With rootless, both the daemon and the container avoid host root in the ordinary launch path. If the control-plane or daemon is compromised, the attacker inherits a much narrower set of host powers. For multi-user platforms and CI-style agent runners, that reduction is often the main reason to adopt rootless at all.

The benefit is real, but it is bounded. Rootless does not change the shared-kernel model, so a kernel escape remains a host escape. It does not remove the need for capability reduction, read-only roots, or narrow mounts. It does not make every workload safer by default. It changes where privilege lives and how much trust you must place in the launch machinery.

### **Why agent tool platforms care**

Agent tool runners execute frequently, often ephemerally, and often under weakly trusted instructions. A model can be prompted to inspect a workspace, build a dependency, or transform files without any malicious intent from the user and still end up traversing adversarial content. If the platform’s control plane or daemon is rootful, then even a relatively ordinary bug in launch logic can become a host-level incident. Rootless narrows that failure mode.

This matters most where many users, jobs, or tenants share a host. A rootless launcher reduces the consequences of accidental socket exposure, daemon compromise, or helper-process misuse because the execution stack no longer depends on a privileged root-owned service. That is an especially useful property for developer-facing agent platforms, CI workers, and shared tool farms where operators would rather accept some compatibility friction than keep a rootful administrative choke point in the middle of every run.

### **The operational constraints are not incidental**

Rootless execution changes the filesystem, networking, and resource-control behavior of the container stack. These are not small implementation details; they determine whether rootless is actually viable for the workload you plan to run.

First, storage behavior changes because rootless engines cannot rely on the same kernel-level mount privileges as rootful engines. Podman documents `fuse-overlayfs` as a common storage mechanism in rootless mode, and it also notes that some storage backends are not supported on NFS and similar distributed filesystems. That matters if your tool runner depends on a network filesystem for shared workspaces or artifacts. A configuration that works cleanly on a local `ext4` host may fail or degrade on a shared volume.

Second, networking may be more constrained. Rootless containers can still provide useful isolation, but they do not always support every host-

network or low-port behavior that a rootful setup can expose. If a tool genuinely needs low-numbered ports, raw socket behavior, or unusual network mediation, you may need additional plumbing or a different design altogether. For agent workloads, that often argues for keeping networking simple: prefer outbound-only access, avoid host networking, and treat host-port exposure as an exception rather than a default.

Third, cgroup behavior can be weaker or more irregular if the host prerequisites are not present. Some resource-control flags are ignored or less effective in rootless configurations when the host is not set up correctly. That is not merely a performance issue; it is a policy risk. If you think you have memory and CPU enforcement but the node silently ignores the flags, your sandbox is weaker than the policy artifact claims. Rootless adoption therefore has to include startup validation of the cgroup and storage prerequisites, not just a successful container launch.

### **Choose the user namespace mode deliberately**

User namespace mode is one of the few rootless knobs that can change both security posture and workflow friction. Podman exposes modes such as `keep-id`, `auto`, and `nomap`, and each one produces a different host-identity story.

`keep-id` maps the calling user to the same UID inside the container. That is convenient when bind-mounted workspace files should remain usable without ownership surprises. It is also the least surprising mode for interactive development because files created in the container look like they came from the same user on the host. The trade-off is that you preserve a direct identity relationship between host and container, which can be undesirable if you want the container to behave as a more abstracted execution domain.

`auto` and `nomap` reduce that direct mapping. They are often more attractive for agent tool runners because they avoid projecting the caller's

host UID into the container as a first-class identity. That can help when you want stricter separation between the user invoking the tool and the file ownership semantics inside the sandbox. The cost is extra operational complexity around workspace permissions and artifact handling. If the tool needs to write into a host-mounted directory, you need to think carefully about which identity owns those files after the run.

The right choice depends on whether you value compatibility or separation more strongly. For a general-purpose agent platform, the default should usually favor separation, with `keep-id` reserved for interactive or compatibility-heavy tools that truly need host-aligned ownership semantics.

### **Rootless reduces daemon privilege, not all privilege**

Rootless often improves the security story for the launch path, but you should not mistake that for complete sandbox strength. The shared-kernel attack surface remains. A kernel bug can still cross the boundary. A privileged mount or device exposure can still undermine the sandbox if you add it later. A host socket mounted into the container can still become a delegation channel to more powerful software. Rootless narrows one category of failure; it does not erase the rest.

That distinction is important because teams sometimes adopt rootless and then become less disciplined about container hardening. They stop asking whether capabilities were dropped, whether mounts are narrow, whether the root filesystem is read-only, or whether the runtime still has an unnecessary control socket. Rootless is best understood as one layer in a defense-in-depth stack. It reduces the consequences of control-plane compromise, but it does not replace the other layers.

### **Compatibility pressure is where rootless often loses**

Rootless is compelling when the workload shape is compatible with

it. It becomes less compelling when the platform keeps adding exceptions until the rootless properties no longer matter. The usual pressure points are easy to predict: low ports, device access, exotic networking, storage drivers that do not behave well under user namespaces, and workloads that expect broad host file ownership semantics.

This is where you need to be skeptical of convenience arguments. If a tool requires a privileged device, a host namespace, or a broad writable mount, rootless may still launch the container but deliver little practical security value. In those cases, rootless can become an operational shim that hides the real risk behind a friendlier launch command. The decision question is not whether rootless can be made to start; it is whether the workload can still run with acceptable safety and supportability once the privilege assumptions are removed.

Agent platforms are especially prone to this drift because the workload catalog is diverse. A code formatter, a repository scanner, a document parser, and a build tool do not want the same host interfaces. If you force all of them into one rootless policy, you may end up reintroducing privileged exceptions so often that the original benefit disappears. The right response is to segment the workload classes instead of overgeneralizing the sandbox.

#### **A decision framework that survives real operations**

Rootless is most valuable when you want to reduce daemon privilege, support multi-user hosts, or limit the impact of a runtime-path compromise. It is a strong default candidate for agent tool runners that mainly process files, run compilers, or perform analysis without special hardware or broad host integration. It is also attractive when the platform has many semi-trusted users or when the control-plane surface is large enough that you do not want it operating as host root.

Rootless is less compelling when the platform depends on host devices, tight kernel integration, or filesystem semantics that are fragile under

user namespaces. It can also lose value when the host environment is heterogeneous enough that rootless support becomes inconsistent across nodes. If the operational burden of exceptions starts to rival the security benefit, you should treat rootless as a selective deployment choice rather than a blanket default.

The practical decision signals are straightforward. Prefer rootless when the workload is file-centric, the host is shared, the launch path is exposed to frequent automation, and the storage/network prerequisites are stable. Prefer rootful or a different sandbox boundary when you need devices, low ports, or mount semantics that rootless cannot support cleanly. Do not use rootless as a substitute for mount hygiene or capability minimization, and do not accept “it works on my machine” as evidence that the mode is robust enough for the fleet.

### **Validate prerequisites before you schedule work**

A rootless deployment should fail early when the host cannot support it. That means you should verify subordinate UID and GID ranges, user namespace support, cgroup behavior, and storage-driver compatibility during host enrollment or node readiness checks. If the platform will rely on `fuse-overlayfs`, confirm that it is installed and behaving as expected before you admit workloads. If a node cannot satisfy the required rootless assumptions, route the job elsewhere or reject it explicitly.

This validation is not a bureaucratic extra. It prevents a subtler failure mode in which the platform believes it has deployed a hardened rootless sandbox while the actual node has fallen back to weaker behavior. In agent systems, silent fallback is dangerous because the tool can be launched many times before anyone notices that a resource flag was ignored or a storage path behaved differently than expected. If the host cannot enforce the policy you intended, the node should not be considered eligible for that workload class.

### **Treat ownership semantics as part of the design**

Rootless changes file ownership in ways that affect both usability and security. A workspace mounted into a user namespace can appear with different UID and GID mappings than it would under a rootful runtime. That is not a bug; it is the cost of avoiding host-root privileges. But it means you need a deliberate policy for artifact ownership, workspace cleanup, and cross-run reuse.

For tools that write outputs back to a shared workspace, ownership surprises can become a major source of friction. For tools that process ephemeral input and emit disposable results, the problem is much smaller. This is another reason to separate tool classes. A rootless sandbox is easy to justify for short-lived analysis jobs and harder to justify for workflows that need stable, host-aligned filesystem semantics. The more a workload depends on direct ownership mapping, the more carefully you should inspect whether `keep-id` is acceptable or whether rootless is the wrong boundary altogether.

### **Rootless should reduce trust, not increase complexity for its own sake**

The best reason to adopt rootless is simple: it removes host-root from the ordinary launch path. That lowers the consequence of daemon compromise, narrows the authority of control-plane bugs, and improves the security story for shared environments. The worst reason to adopt it is fashion. If you enable rootless but then rely on broad writable mounts, host devices, ignored cgroup flags, or compatibility exceptions that undermine the original goal, you have only moved complexity around.

Used well, rootless is a strong fit for agent tool runners that handle frequent, partially trusted, file-oriented work. Used poorly, it becomes another layer of operational vocabulary without a meaningful reduction in risk. The useful test is whether the workload can still run under

a user namespace while keeping the launch path unprivileged and the host interfaces narrow. If the answer is yes, rootless earns its place. If the answer is no, the platform should admit that a different sandbox model, or a more specialized exception path, is the honest choice.

#### 4.4. Modeling the Container Runtime Stack as Part of the TCB

A hardened container payload does not help much if the software that launches it is itself overpowered, opaque, or easy to trick. In an agent tool platform, the launch path is part of the security boundary because it translates a policy decision into mounts, namespaces, identities, and runtime flags. If that translation is wrong, the container starts with the wrong authority before the tool process even runs. The practical question is not whether the container image is trusted; it is whether the broker, engine, shim, runtime, and node policy loaders that create the container deserve the trust you have implicitly granted them.

A minimal, policy-driven launch makes that trust chain visible:

```
tool-broker run \  
  --policy policy/tool.yaml \  
  --image my-tool:latest \  
  --rootfs-ro \  
  --drop-all-caps \  
  --no-new-privs \  
  --seccomp seccomp/default.json \  
  --mount in:/work/in:ro \  
  --mount out:/work/out:rw \  
  --runtime runc
```

That command says nothing about the tool itself and everything about the systems that interpret the request. The broker reads policy, compiles it into a launch specification, and hands it to a runtime stack that must honor the requested boundary. If you do not model those intermediate components as security-relevant, you miss the most danger-

ous place for privilege drift: the code that decides what container you actually get.

### **The TCB begins above the container process**

The trusted computing base for containerized tool execution is larger than the process tree inside the sandbox. It includes the control-plane broker or orchestrator that decides whether a run is allowed, the engine or API layer that receives and stores the request, any containerd-class management service that mediates lifecycle operations, the OCI runtime binary that turns spec into process state, the shims or helper processes that keep the container alive, and the host kernel facilities that enforce the isolation primitive. Storage and networking helpers belong in that picture too, because they assemble the actual filesystem and packet paths that the process will inherit.

That inventory matters because each layer can widen the sandbox in a different way. A control-plane bug can grant the wrong mounts or credentials. An engine bug can ignore a policy field or apply the wrong default. A runtime parser bug can misread a mount or capability setting. A shim or helper process can retain privileges you did not intend to expose. A kernel bug can defeat the whole boundary after launch. You want each of these components to be part of the security review, not an invisible implementation detail.

### **Policy belongs in a reviewable control plane**

High-level authorization should live in a control-plane layer that knows the tool identity, invocation context, requested scope, and audit correlation. That layer should decide whether the run is allowed and what its maximum authority will be. The runtime should not invent policy; it should instantiate policy. If you spread those decisions across shell scripts, ad hoc hooks, and container defaults, you lose the ability to reason about where privilege came from.

The cleanest design uses a narrow translation boundary. The control plane accepts a request, evaluates the requested scope, and emits an immutable launch spec. The execution layer receives that spec and compiles it into the exact runtime settings for mounts, credentials, namespaces, capabilities, and seccomp. A post-launch inspection step then verifies that the live container matches the approved spec. This separation is especially useful for agent platforms because the request may be triggered by untrusted text or data. You do not want the tool itself to negotiate for more power than the policy layer intended.

If you are already using a policy model from the previous hardening work, keep that model upstream of the runtime. The runtime should not decide whether a capability is justified; it should enforce the set it is given. That rule sounds obvious, but many platforms violate it by allowing launch-time flags, helper defaults, or image metadata to override policy after review.

### **containerd, shims, and the runtime are all in scope**

containerd-class architectures make the execution path explicit, which is useful because they show where control ends and mediation begins. Runtime v2-style designs separate lifecycle management from the actual OCI runtime invocation by placing a shim between the higher-level manager and the container process. That separation improves robustness and observability, but it also makes the shim part of the TCB. If the shim can mediate I/O, keep lifecycle state, or restart a workload, it can also become a point where privilege assumptions are enforced incorrectly.

The OCI runtime binary, such as `runc` or an equivalent implementation, is especially sensitive because it parses the container spec and creates the namespaces, mounts, and credential state. A bug in that parser or in the runtime's interpretation of the spec can invalidate otherwise sound policy. You can write an excellent `config.json`, but if the run-

time mishandles a field, the live container may not match the intended security envelope. That is why runtime selection matters even when the caller-facing interface stays OCI-compatible.

Switching to an alternative runtime, such as gVisor or Kata, changes the TCB and the boundary without changing the fact that the caller still submits an OCI-like request. That distinction is useful operationally: the API may stay familiar while the enforcement model changes substantially. A platform may use the same broker and the same policy language but rely on a different execution boundary underneath. In other words, “container” is the interface; the runtime is part of the security architecture.

### **Minimize privilege in the launch path itself**

The launch path should not contain broad host authority unless it absolutely must. The usual danger points are daemon sockets, root-owned helpers, mutable admission hooks, and orchestration scripts that can mutate the launch spec after policy evaluation. If a tool runner can talk directly to a privileged socket, it may be able to ask for sibling containers, host mounts, or other administrative actions that were never part of the approved scope.

A safer pattern is to reduce the number of actors that can directly request privileged runtime operations. Prefer a broker that compiles a declarative spec and submits it to the runtime through a constrained interface. Avoid a design where the tool or its helper scripts can reach the container engine API directly. Keep image selection, mount assembly, namespace choices, and capability settings attributable to a reviewed request rather than to operator convenience.

That same discipline applies to host services that look harmless because they are “just helpers.” A storage driver, network plugin, or policy loader may run with more privilege than the workload itself. If one of those components is compromised, the attacker may not need to

break out of the container at all; they may simply abuse the trusted component that created it. In agent systems, where automation frequently chains through support services, that risk is not abstract.

### **Drift detection is part of the boundary, not an afterthought**

A launch spec only matters if the live container matches it. Configuration drift is one of the easiest ways to widen tool access without a visible policy change. A runtime may fall back to a default mount, inherit an unexpected device, ignore a resource flag, or accept a debug option that was never meant for production. If you do not inspect the resulting container state, you can end up trusting the paper policy rather than the actual process state.

You should verify the launched environment against the approved spec as a normal step in the control flow. That means checking the effective mount set, namespace membership, capability set, seccomp profile, and user identity mapping. It also means ensuring that the runtime did not substitute an unsafe default because a profile was missing or a feature was unavailable. The validation step is not a nice-to-have in a high-assurance environment; it is the only way to know whether the spec was honored.

A simple operational rule helps: if a required security artifact is missing, fail closed. Do not continue with a weaker container because the launch would otherwise succeed. A missing seccomp profile, an absent AppArmor profile, or an unexpected runtime fallback should stop the run. Silent downgrade is the enemy here because it creates a false sense of compliance.

### **Treat policy compilation as a security-critical function**

The translation from abstract policy to concrete runtime settings is a security-critical function in its own right. You are not just formatting JSON or emitting CLI flags; you are deciding which kernel and host

interfaces the tool will receive. That translation should be deterministic, reviewable, and testable. If a policy says the root filesystem is read-only, the compiled spec should not accidentally add a writable host mount. If the policy says capabilities are dropped, the runtime should not inherit a default add-back list from an image or profile.

This is where diff-friendly launch specs pay off. A signed or immutable spec lets you compare the approved settings with the actual runtime request and quickly spot unwanted changes. It also makes configuration review practical. Security reviewers can reason about a single artifact instead of reconstructing the launch state from a chain of scripts and environment variables. That matters because many container failures in real systems are not dramatic exploit paths; they are mundane policy mismatches that quietly expand access.

If you need a concrete shape for that artifact, keep it compact and explicit:

```
image: my-tool:latest
runtime: runc
rootfs_readonly: true
drop_caps: ["ALL"]
no_new_privileges: true
seccomp: seccomp/default.json
mounts:
  - src: in
    dst: /work/in
    readonly: true
  - src: out
    dst: /work/out
    readonly: false
```

That format is useful not because YAML is special, but because it keeps the policy legible. A security review should be able to answer, at a glance, what the container can read, what it can write, and which privileged kernel operations remain available.

### **The runtime stack is more than the happy path**

A robust TCB analysis includes failure modes, not just nominal launches. A bug in the launcher may bypass policy before the runtime is even called. A parser bug in the OCI runtime may mis-handle a field and create a broader mount than intended. A shim may keep state across runs in a way that surprises the policy engine. A node-level policy loader may fail to load an AppArmor or seccomp artifact and continue with a weaker configuration. Each of these failures produces a different kind of drift, and each needs a different detection strategy.

The critical point is that pre-runtime bugs can be just as dangerous as runtime bugs. If the orchestration layer mis-applies the policy, the sandbox may be wrong even though the runtime is perfectly correct. That is why TCB analysis must start above the container process itself. The launch request, the policy decision, and the runtime execution form one chain, and the chain is only as strong as its weakest trusted link.

### **Privilege minimization also means reducing standing access**

You can narrow the launch path by reducing both the number of privileged components and the time they remain privileged. Favor short-lived brokers over long-lived administrative daemons when that is operationally feasible. Avoid letting a general-purpose engine socket sit exposed to every component that needs to start a tool. Prefer separate service identities for policy evaluation, launch execution, and artifact handling. The fewer actors that can directly mutate runtime state, the smaller the attack surface for prompt-induced abuse or support-tool compromise.

This is a useful place to connect the control plane back to the scope model. A policy engine can approve a narrow filesystem scope, but the runtime stack still has to enforce that scope without adding undocumented exceptions. If the launch path is already overprivileged, the policy model becomes aspirational rather than enforceable. The trust

boundary is therefore not only about where the policy lives, but also about where the power to violate it resides.

### **When to change the runtime boundary itself**

Sometimes the right answer is not to trust the standard runtime stack more carefully, but to change the boundary. If your workload or threat model makes shared-kernel risk too high, a different runtime such as gVisor or Kata may be a better fit because it inserts additional isolation between the tool and the host kernel. That does not remove the need to trust a runtime stack; it changes the stack and therefore the TCB. The caller still launches a container-like unit, but the enforcement path is no longer the same as the default OCI stack.

That choice should be driven by workload shape and residual risk, not by marketing labels. If your agent tool runner handles high-frequency, file-centric jobs with moderate sensitivity, a hardened OCI stack may be sufficient. If it handles more sensitive code, untrusted binaries, or data with a high consequence of escape, a stronger boundary may justify the operational cost. The important part is to recognize that changing runtimes changes both assurance and complexity. It is not a drop-in detail from a security perspective.

### **Operationally, trust the spec less than the live state**

The final habit to cultivate is verification. A spec is what you asked for; the live container is what you actually got. For agent systems, that distinction is non-negotiable because the workload is often triggered by content you do not fully control. A single unexpected fallback can turn an otherwise good policy into an overbroad sandbox. Inspect the running container, correlate it with the launch request, and treat discrepancies as security-relevant events.

When you do that consistently, the runtime stack stops looking like plumbing and starts looking like what it really is: the enforcement path

for the platform's container policy. If you trust it blindly, you inherit its bugs, defaults, and hidden privilege. If you model it as part of the TCB, you can place policy where it belongs, minimize the privilege required to launch tools, and detect drift before it widens access in ways the agent cannot see.

## 4.5. Operating Deterministic Denies and Safe Failure Paths

A sandbox that only works when everything is configured perfectly is not a sandbox you can operate at scale. Agent tool runners fail all the time: a path is outside the declared mount set, a write target is read-only, a capability is missing, a seccomp rule blocks a syscall, a node lacks a required profile, or a rootless host quietly ignores a resource flag it cannot enforce. The question is not whether those failures happen, but whether they happen in a predictable, inspectable, fail-closed way. If denial is deterministic, the platform can classify the failure, preserve safety, and let you refine policy without normalizing privilege expansion.

A concrete deny contract makes the behavior legible:

```
{
  "run_id": "a7f3c2",
  "tool": "repo-scan",
  "status": "denied",
  "reason": "filesystem_scope_violation",
  "detail": "write target /etc is not allowed",
  "retryable": false,
  "support_id": "deny-20260501-1842"
}
```

That shape matters because it separates operator-facing diagnostics from agent-facing signal. The agent learns that the request failed because it exceeded the declared scope, not because the platform became unstable or because the operator might be persuaded to widen access.

The platform, meanwhile, retains enough structure to decide whether the denial was ordinary under-scoping or evidence of suspicious behavior.

### **Fail closed by default, not by aspiration**

A safe failure path begins with the assumption that the sandbox should refuse to continue whenever it cannot enforce the intended policy. That includes missing security artifacts, unsupported runtime features, and rootless prerequisites that are absent on the node. If a seccomp profile is unavailable, do not fall back to an unfiltered syscall set. If an AppArmor profile is missing on a node that is supposed to enforce one, reject the run. If a rootless container cannot obtain the cgroup support or storage behavior the policy assumes, treat that as a launch error instead of an implementation detail.

This is the operational opposite of “best effort” security. Best effort sounds tolerant, but in a sandbox it usually means silent weakening. A tool that starts with a weaker boundary than the policy describes is more dangerous than one that fails immediately, because the former produces false assurance and the latter produces an explicit incident. In agent systems, the cost of a failed run is usually lower than the cost of an unnoticed privilege downgrade.

### **Determinism is what makes denial useful**

A denial is operationally valuable when you can classify it reliably. Read-only filesystem violations should fail as writes to a protected path, not as generic I/O noise. Capability-dependent actions should return a predictable permission failure, not a confusing partial success. Mount rejections should identify the blocked path or mount class. Namespace restrictions should surface as a constrained launch, not as a container that half-started and then behaved unpredictably.

Determinism matters because agent software tends to retry. If the fail-

ure is ambiguous, the tool or orchestrator may keep trying different paths, different flags, or broader scopes. That is exactly the wrong feedback loop. A deterministic error allows the agent to stop, log the denial, and ask for a policy review rather than improvising a workaround. For the human operator, it also means you can distinguish a normal under-scoped request from a tool that is probing the boundary in a way that deserves scrutiny.

### **Shape the error contract around safe and unsafe audiences**

The error you expose to the agent should be deliberately sparse. It needs to tell the model enough to avoid blind retries, but not enough to reveal sensitive host structure, prompt internals, or policy implementation details. The richer context belongs in an audit stream, not in the tool-visible response. That audit stream can carry correlation IDs, the declared scope, the denied resource, the runtime component that issued the failure, and any non-sensitive facts needed for debugging.

A useful mental model is to split denial into three audiences. The agent gets a compact, non-secret classification. The operator gets a redacted but actionable diagnostic. The security record keeps the full provenance needed to review the request later. This separation prevents the platform from using denial handling as an accidental side channel. It also keeps prompt-injected tools from learning exactly how close they were to a broader permission set.

### **Read-only roots and scratch mounts make failure legible**

A read-only root filesystem is especially helpful when you want failures to be obvious. If the tool tries to write outside its declared scratch area, the write should fail immediately. That failure is easier to interpret than a permissive environment where the tool silently modifies its own image, caches state in hidden directories, or accumulates artifacts in places the platform never meant to expose. The same logic applies to explicit scratch mounts: a tool should have a small number of writable

paths, and everything else should be read-only by default.

This is not just a persistence control. It also makes policy mistakes visible. If the tool unexpectedly needs a write location under `/etc`, `/usr`, or another protected tree, the denial tells you something concrete about the workload shape. You can then decide whether the workload was mis-scoped or whether the exception should be handled through a more narrowly mediated channel. The point is to force the mismatch into the open rather than letting the container quietly absorb it.

### **Treat startup validation as part of the sandbox**

Safe failure paths require validation before scheduling or launch. You should verify the kernel features, seccomp profile, AppArmor support, subordinate UID/GID ranges, storage driver behavior, and any other prerequisite that the policy assumes. If the node cannot meet the baseline, reject the workload. Do not let the runtime discover the mismatch halfway through execution.

This is particularly important in mixed fleets where rootless support varies. A node may appear capable of running a rootless workload, but if the cgroup prerequisites are missing, some resource-control flags may be ignored. That is a policy risk, not a mere compatibility quirk. If the platform thinks memory or CPU limits are enforced but the host does not honor them, the sandbox is weaker than it looks. Startup validation avoids that false confidence by turning unsupported assumptions into explicit admission failures.

### **Retry discipline prevents denial from becoming escalation**

A denied run should not automatically trigger a broader retry. If the container could not write to a protected path, retrying with more privilege is not a neutral recovery step; it is a policy change. Likewise, if the workload failed because a capability was blocked, the right response is not to keep reissuing the run with progressively looser flags until

something works. That pattern normalizes privilege expansion under operational pressure.

The platform should distinguish between bounded retries that preserve the same policy and retries that would cross into a different risk class. A transient network timeout may justify another attempt. A mount rejection does not. A missing node profile may justify rescheduling on a compliant node. A request for a new capability requires a review, not an automatic retry. The criterion is simple: if the next attempt would change the authority of the tool, it is no longer a retry; it is a new security decision.

### **Quarantine the uncertain cases**

Not every denial is the same. Some are ordinary under-scoping, and some indicate hostile behavior or malformed input. A secure platform should route uncertain cases into quarantine rather than forcing an immediate yes-or-no expansion. Quarantine can mean retaining the input artifact for review, pausing the tool class pending investigation, or marking the run as suspicious so subsequent requests receive more scrutiny.

This is particularly useful for agent workloads because the same tool may process benign and adversarial content on adjacent runs. A repository scan that fails because a path was outside the mount set might simply need a narrower workspace. The same failure repeated across a crafted input set may indicate that the content is trying to drive the tool into broader host reach. Quarantine preserves evidence and prevents the platform from responding reflexively with broader privileges.

### **Policy refinement should narrow uncertainty, not widen default power**

Good sandbox operations produce feedback loops, but those loops should move through review, not through automatic elevation. When

a denial reveals that a declared scope was too narrow for a legitimate workload, you refine the policy artifact and retest. You do not grant a standing exception in the launch script just to make the current request succeed. The distinction seems procedural, but it is central to keeping the baseline trustworthy.

A healthy refinement loop asks three questions. First, was the workload mis-specified? Second, can the requirement be mediated outside the container through a narrower host service or a dedicated wrapper? Third, if the exception is unavoidable, can you make it specific to a tool class, a data class, or a node pool rather than to the entire platform? That sequence keeps the platform from converting every failure into a larger sandbox.

### **Operator context should be precise, not verbose**

A denial message needs enough context to support action, but not enough to leak secrets or normalize experimentation. A good message names the failure class, the relevant resource category, and the stable run identifier. It should not dump the full mount table, credentials, prompt history, or kernel details unless an operator is already in a restricted diagnostic channel. The tighter the interface, the less likely the denial path becomes an information leak.

There is a useful operational trade-off here. A more verbose message can reduce debugging time in the short term, but it also increases the chance that a compromised tool learns about host structure or policy boundaries. A redacted message takes a little more effort to interpret, yet it keeps the sandbox honest. For platform-wide defaults, the safer choice is usually to optimize for redaction and attach richer details only to authenticated operator logs.

### **Make the failure mode consistent across components**

The launch broker, runtime, and node policy loaders should agree on

how a denial looks. If the broker reports a clean policy violation but the runtime emits a low-level parser error, operators will struggle to tell whether the request was blocked intentionally or failed due to infrastructure drift. Consistent classification lets you separate policy refusal from platform malfunction, which is critical when you are debugging a fleet.

That consistency also makes automation safer. If every component emits a different style of error, agent logic will overfit to incidental phrasing and may start treating unrelated failures as opportunities to retry with different parameters. A common taxonomy gives you a stable basis for automation: this is a scope violation, this is a missing prerequisite, this is a node capability mismatch, this is a suspicious input pattern. The fewer edge cases you invent in the error path, the less likely the platform is to drift into permissive behavior just to simplify its own control flow.

### **Use denial as a signal for workload segmentation**

Repeated denials are often a sign that the platform has put too many different jobs into one sandbox class. A file-analysis task, a build task, and a hardware-integrated task do not want the same resource envelope. If one class keeps asking for mounts, devices, or privileges that another class never needs, the right response may be segmentation rather than repeated policy exceptions.

This is especially important for agent systems because the tool catalog tends to grow faster than the policy model. If you see the same denial pattern over and over, ask whether the tool should be split into a different runner with a different scope. That is usually safer than broadening the default sandbox for all tools just to accommodate one outlier. A denial, in that sense, is not only a failure; it is data about where the platform boundary is too coarse.

### **Preserve evidence without preserving danger**

A blocked run may carry artifacts worth retaining for review: a request payload, a denied mount path, a suspicious input file, or a hash of the tool image. Quarantine those artifacts carefully. Do not re-execute the input in a less constrained environment just to “see what happens.” Do not restore the denied tool into a privileged context unless you are intentionally doing incident analysis under a separate control regime.

The evidence chain should be enough to explain what failed without recreating the risk. That means immutable identifiers, redacted logs, and controlled artifact storage. For agent workloads, this is especially important because the same input that triggered a denial may be able to trigger a more serious bug if you replay it casually. Retain the evidence, but keep the execution path constrained.

### **The safest sandbox is the one that makes bypass expensive**

A container sandbox is useful only when its denials are reliable enough that operators do not get tempted to route around them. If the platform can tell a clear story about why a run was blocked, how to review the request, and how to refine policy without generalizing privilege, then the boundary stays meaningful. If denials are noisy, ambiguous, or hard to investigate, the platform will eventually weaken the sandbox to keep delivery moving.

That is why deterministic deny behavior is not a usability detail. It is the operational mechanism that keeps hardened defaults in place after the first real workload collides with them. When the platform fails closed, reports the failure cleanly, and channels exceptions through review instead of reflexive expansion, the sandbox remains a boundary rather than a suggestion.



## Chapter 5

# Kernel Hardening Inside and Around the Sandbox

*A well-configured container can still present a surprisingly large kernel attack surface to hostile or confused tool code. This chapter asks a sharper question: once a process is already inside the sandbox, how do you make the kernel and surrounding host policy say “no” in narrow, predictable, auditable ways?*

### **5.1. Syscall Allowlisting with seccomp: From Generic to Workload-Specific**

Many teams say they “use seccomp” when they have only accepted a vendor default. That default is useful, but it is not a contract with the workload. A runtime-supplied profile usually reflects compatibility

pressure across a broad ecosystem, not the narrow needs of one tool. If you want seccomp to do real security work, you have to derive the policy from what the tool actually does, what it is permitted to do, and what you are willing to break when behavior drifts.

A practical starting point is a profile skeleton that is intentionally conservative. For an OCI runtime, that usually means a default-deny posture with explicit per-syscall exceptions and an architecture list that matches the deployment target. A minimal policy often looks like this:

```
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "architectures": [
    "SCMP_ARCH_X86_64"
  ],
  "syscalls": [
    {
      "names": [
        "read",
        "write",
        "exit",
        "exit_group",
        "futex",
        "clock_gettime",
        "mmap",
        "munmap",
        "brk",
        "rt_sigaction",
        "rt_sigprocmask"
      ],
      "action": "SCMP_ACT_ALLOW"
    }
  ]
}
```

That file is not meant to be complete; it is meant to be a boundary marker. You run the tool, observe which calls it actually makes, and then decide which calls belong in steady-state operation versus transient setup. The distinction matters because startup often exercises code paths that never recur once the tool is warm: dynamic loader activity, locale probing, plugin discovery, certificate loading, and allocator initialization can all expand the syscall set in ways that are irrele-

vant after bootstrapping. If you do not separate those phases, you end up with a profile that is either too wide to matter or so narrow that every upgrade looks broken.

### **Treat seccomp as an API contract, not a denial list**

seccomp filters the syscall entry point itself, so it reduces the kernel attack surface before the kernel has to interpret file paths, sockets, or user-controlled arguments. That makes it qualitatively different from post hoc sandboxing. You are not trying to catch bad behavior after it has already crossed multiple subsystems; you are deciding which kernel interfaces the process may even attempt to use. The policy therefore needs to express the workload’s legitimate contract, not just a list of “dangerous” calls.

The right mental model is a stable interface between the tool and the kernel. If the tool only needs ordinary file I/O, memory management, time queries, and thread synchronization, you can lock the profile around those classes and reject everything else. If it needs a broader set because it shells out, loads plug-ins, or uses seccomp-unfriendly libraries, you can still isolate the broadened surface to that one tool instead of imposing the same superset on every workload in the fleet. That distinction is especially important in heterogeneous agent systems, where a browser-driven scraper, a PDF converter, and a compiler helper have very different syscall shapes even if they all run in the same container platform.

### **Build from observed behavior, then prune by intent**

The most reliable workflow begins with tracing a representative run. Capture startup and steady-state execution under inputs that exercise normal functionality, not just the happy path. You want the calls that occur when the tool opens its working files, resolves libraries, authenticates to its local runtime dependencies, and completes its main task. You also want to know which calls only appear because you attached

a debugger, enabled verbose logging, or launched the tool from a shell with unusual environment state. Those incidental calls are not part of the contract.

A disciplined derivation sequence is:

- Start from the narrowest reasonable baseline for the target architecture and runtime.
- Trace multiple representative runs.
- Group syscalls by phase: initialization, steady state, shutdown.
- Remove calls that are artifacts of tooling rather than workload need.
- Re-run with the candidate profile and compare failures to the expected behavior.

This is where engineers often make the wrong optimization. They look for the longest syscall list common to all observed runs and treat it as “safe.” That is backwards. The common set may still be polluted by libraries or one-off side effects, and it can still include calls that are unnecessary but merely tolerated by the tool. Better to ask which calls are necessary for the tool’s declared purpose, then test whether the tool remains correct when the rest are removed.

The workflow is easier to manage when you annotate each syscall class with a reason to exist. For example, memory management calls may be required by the language runtime, while `clone`, `unshare`, or `setns` may be present only because a dependency probes container state. File-system-related calls often deserve extra review because they can indicate paths into broader object access, but the syscall itself may still be essential if the workload creates temporary scratch files or uses memory-mapped data. That is why the profile should be written by a

human who understands the tool's operational model, not assembled mechanically from one trace.

### **Choose actions by failure mode, not by habit**

The seccomp action you select is part of the policy, not an implementation detail. `SCMP_ACT_ERRNO` returns a predictable failure and is useful when you want the application to continue in a degraded but diagnosable state. `SCMP_ACT_TRAP` can surface a SIGSYS and is better when you want the process to fail loudly under policy violation. `SCMP_ACT_KILL_PROCESS` or related kill actions are appropriate when any violation is treated as a hard security fault. `SCMP_ACT_LOG` is useful during tuning, but it is not a control in itself. `SCMP_ACT_NOTIFY` delegates the decision to a user-space supervisor, which is powerful but operationally more complex because the policy no longer terminates inside the kernel alone.

The choice depends on what the syscall represents. A blocked call to `mount`, `ptrace`, or `unshare` often deserves a hard fail because those calls indicate a privilege boundary is being probed. A blocked optional filesystem query may deserve `ERRNO` if the workload can proceed without the feature. If the tool needs to degrade gracefully under feature flags, `ERRNO` can preserve control flow while still preventing access. When you are not sure, prefer the action that makes the failure easiest to detect during testing without introducing silent fallback behavior in production.

Argument filtering can tighten the policy further when the runtime exposes it. Instead of allowing a syscall in all cases, you can allow it only for a particular set of arguments or flag combinations. That is especially valuable for calls that become risky only when specific bits are set, such as interface-creating operations, namespace-related flags, or file-opening modes that expand privilege. Use argument filtering sparingly and only when the criteria are stable across kernels and libc vari-

ants. Overfitting a filter to one libc implementation can create a brittle profile that breaks on the next distribution refresh.

### **Separate compatibility pressure from genuine need**

Generic profiles are usually compatibility profiles. They allow enough to keep common software working across many deployments, which is why they are a weak baseline for a specific tool. A browser profile and a database helper profile may both need `futex` and `mmap`, but they differ sharply in their access to process control, namespace operations, file creation patterns, and networking-related behavior. If you reuse one broad profile across both, you give up the very property `seccomp` is meant to provide: a reduction in the reachable kernel surface that tracks the actual task.

The compatibility problem becomes visible when you upgrade the runtime stack. A new libc may introduce a different path through `stat`, `openat`, `clone3`, or time-related syscalls. A newer kernel may expose interfaces that older tools never used, while also changing which architecture aliases a runtime can resolve. If your profile was written as a lowest-common-denominator allowlist across unrelated workloads, you have no clean place to absorb that drift. If the profile is tool-specific, you can decide whether the new call belongs to the tool contract or whether it is an implementation artifact that should be removed from the dependency chain.

That is why the maintenance burden is lower when you split profiles by tool family or by syscall shape. A compact allowlist for a code formatter, a package indexer, or a single-purpose converter is easier to reason about than a fleet-wide superset. The goal is not only security. It is also operability: when a denial happens, you want to know whether you have found a real regression, a new feature, or a hidden dependency you should eliminate.

### **Use runtime and kernel features intentionally**

The profile surface exposed by OCI runtimes is richer than a simple allowlist. You can specify the default action, per-syscall rules, architecture lists, and, where supported, finer-grained flags and notification paths. That lets you express a policy that is explicit about what it expects from the platform. If you know a workload only ever runs on one architecture, declare that architecture instead of relying on a multi-arch fallback. If the workload must support multiple kernels or runtime versions, keep the profile aligned with the least common operational baseline and introduce feature-specific branches only when the policy review justifies the added complexity.

That said, do not confuse feature richness with better security. A complicated policy is harder to audit and easier to accidentally widen. A good profile is usually small, legible, and purpose-built. If you need a large number of special cases, ask whether the workload is too heterogeneous to benefit from one seccomp contract or whether you should split it into separate execution modes with distinct profiles.

The kernel also exposes useful capability information that helps you avoid mismatches. If an action is unavailable on one host class or if logging support differs across nodes, treat that as a deployment constraint, not as a reason to silently weaken the profile. Heterogeneous fleets need policy definitions that fail visibly when the assumed seccomp behavior is missing.

### **Test the deny path as aggressively as the allow path**

A seccomp profile is incomplete until you verify that the blocked calls fail the way you expect. Negative testing should be part of the release gate. Intentionally invoke syscalls that are known to be out of contract and confirm that the workload receives the intended response: an error code, a signal, a kill, or a notification. If the process keeps running as if nothing happened, the profile is not doing the job you think it is.

The negative test suite should include at least three classes of probes:

obviously forbidden operations, edge-case variants of allowed operations, and calls that are currently allowed only because a dependency requires them. The first class verifies obvious security boundaries. The second catches argument-filter mistakes and architecture mismatches. The third guards against accidental breakage when libraries evolve. This is especially useful for language runtimes and tooling ecosystems that spawn helper processes or load modules dynamically, because the syscall profile often needs to survive behavior that the primary tool author never explicitly documented.

When negative tests fail, the response should not be to widen the profile reflexively. First determine whether the blocked call is required, incidental, or evidence of an unexpected code path. If it is required, document why. If it is incidental, consider removing the dependency or altering the runtime configuration so the call no longer appears. Only if the behavior is truly legitimate and stable should you expand the profile.

### **Expect profile drift and manage it as configuration**

Profiles drift for the same reason applications drift: dependencies change faster than operators expect. A new runtime version may add syscalls, reorder initialization, or probe kernel features more aggressively. A new deployment environment may run under a different seccomp-aware orchestrator or container runtime version. A new architecture class may require different syscall names for equivalent behavior. If you treat the profile as a static artifact, those changes show up as noisy failures. If you treat the profile as managed configuration with explicit ownership and review, the failures become upgrade signals.

Version control helps, but only if the profile is written in a way humans can review. Keep separate files for distinct tools instead of accumulating giant shared policies. Group syscall names by purpose. Annotate why each exception exists. Avoid one-off emergency exceptions unless

you also schedule a cleanup task. For larger fleets, keep a generated baseline and a hand-reviewed delta rather than a single opaque artifact. That arrangement makes it obvious which rules came from observed behavior and which came from a deliberate risk decision.

The maintenance model also benefits from policy tiers. A strict profile can serve the steady-state hot path, while a slightly looser profile can cover migration windows, debugging sessions, or rare administrative modes. You do not have to choose between absolute rigidity and permanent permissiveness. You can expose multiple explicit profiles and attach them to known operational states, which makes the policy easier to reason about than a single overgrown superset.

### **Avoid the failure patterns that quietly erase the benefit**

Three mistakes recur in real deployments. The first is starting from a permissive vendor default and never tightening it because the profile already exists. The second is adding a broad exception for one troublesome library and then leaving it in place for every tool that shares the image. The third is merging unrelated workloads into a single profile so that the noisiest one defines the allowed surface for all of them. All three convert seccomp from a control into a checkbox.

A subtler mistake is treating every denial as proof that the profile is wrong. Some denials are exactly the intended behavior. They reveal code paths you did not mean to expose. Others are useful signals that a dependency is probing more of the system than it should. If your process for handling denials begins with “widen the allowlist,” you will gradually erase the security value of the policy.

A better pattern is to classify each blocked syscall by intent. If the call is part of a documented capability, add it with a comment or metadata that explains the requirement. If the call is a convenience feature, decide whether the feature should remain available. If the call is unexplained, treat it as a bug, a library probe, or an attempted escape until

you have evidence otherwise. That discipline keeps the profile tied to the tool's actual contract instead of the accidental behavior of its transitive dependencies.

### **Use seccomp where it is strongest, then hand off the rest**

A narrow syscall contract is valuable because it shrinks what the process can ask the kernel to do. It does not, by itself, determine which files, sockets, IPC objects, or labeled resources the process may touch after a syscall is allowed. That is why syscall allowlisting is only one layer in the chapter's overall design. Seccomp is strongest at removing broad classes of kernel entry points, especially those that open the door to privilege escalation, namespace manipulation, or introspection. It is less expressive for object-level access and path-based policy, which is where MAC and self-confinement mechanisms do more of the work.

That boundary is operationally useful. Once you have a tight seccomp profile, the remaining attack surface becomes easier to reason about in later controls. You can then use MAC to shape file and object access and Landlock to let a process narrow its own reach when host-managed policy is unavailable. The result is not one perfect barrier but a layered model in which each control removes a different kind of failure mode.

The practical outcome is a profile that is small enough to audit, strict enough to matter, and stable enough to survive the next runtime upgrade. When you keep the profile attached to one tool, one behavior set, and one explicitly reviewed risk tolerance, seccomp stops being a generic checkbox and becomes a precise kernel-facing contract.

## 5.2. Failure Handling and Observability for seccomp Denials

A profile that blocks the right syscall at the right moment is only half the job. The other half is deciding what the system should do when a legitimate workload hits the policy, and how much of that failure you should expose without turning the feedback path into a leak. A runtime update, a changed libc, or a new kernel can turn a previously stable tool into a deniable one overnight. If your only signal is a terse crash or a noisy audit line, you will spend more time guessing than fixing. Good seccomp observability makes denials actionable without giving away secrets or widening the attack surface.

A useful operational template is to normalize every denial into a small, structured event at the tool runner boundary. The kernel still makes the enforcement decision, but the runner translates the result into a run-scoped record:

```
{
  "run_id": "run-20260501-184233",
  "tool": "pdf-extract",
  "policy_version": "seccomp-v12",
  "phase": "steady-state",
  "syscall": "openat",
  "arch": "x86_64",
  "action": "ERRNO",
  "expected": false,
  "result": "denied"
}
```

That event shape is intentionally sparse. It gives operators enough context to identify the affected invocation, enough detail to correlate the blocked call with a policy version, and enough structure to feed automated triage. It does not include command lines, prompt text, raw file paths, environment dumps, or file contents. Those belong to the workload itself, not to the denial record.

## **Treat the denial lifecycle as a control loop**

A seccomp denial has a recognizable lifecycle. First, the kernel matches the syscall against the active filter and chooses an action. Then the process experiences that action as a failure mode: an `errno`-style return, a signal, a trap, termination, a log entry, or a handoff to user space. After that, the runner or host captures the event, if the chosen action makes it visible. Finally, operators or developers decide whether the block reflects intended policy, a broken dependency, a compatibility regression, or suspicious behavior.

That sequence matters because the right remediation depends on the outcome. A process killed by policy is not the same thing as a process that receives `EPERM` and falls back to a different code path. A trap that raises `SIGSYS` may surface as an application crash unless you handle it carefully. A logged-only denial can disappear into the noise unless you attach it to a run identity and policy version. If you do not classify the outcome at the moment it occurs, later analysis becomes ambiguous.

The most important distinction is between deterministic policy failures and incidental software failures that happen to surface after the denial. For example, a blocked `openat` may cause a parser to report “file not found,” even though the real cause is a policy reject. A blocked `clone3` may cause a runtime to retry with a different thread model, which changes the eventual symptom. Your observability contract should preserve the causal chain: kernel decision, process response, runner observation, and user-visible result. That is the minimum needed to separate misconfiguration from actual incompatibility.

## **Pick an action based on the failure semantics you want**

seccomp gives you multiple outcomes, and they do not behave alike. `SCMP_ACT_ERRNO` is the most operationally forgiving because it lets the process continue with a predictable failure code. That makes it a good default for features that can fail closed without corrupting state.

SCMP\_ACT\_TRAP is more forceful and useful when you want a clear crash signal during development or in a tightly controlled runtime. SCMP\_ACT\_KILL\_PROCESS is appropriate for high-confidence escape attempts or syscalls that represent unacceptable risk under any circumstance. SCMP\_ACT\_LOG is valuable when you are tuning a profile, but it does not stop the syscall. SCMP\_ACT\_NOTIFY delegates the decision to a supervisor and is powerful when policy needs context outside the kernel, though it creates a live dependency on the notification path.

You should not choose the action merely because it exists. Choose it by asking what the workload should do after the block. If the denied call is part of optional feature probing, `ERRNO` preserves the rest of the run while still recording that the policy rejected the request. If the denied call indicates a direct attempt to escape confinement, killing the process is defensible because it prevents fallback paths from continuing after a serious violation. If you need to inspect arguments or make an allow/deny decision dynamically, notification can help, but only if the supervisor is highly available and carefully scoped.

The action also determines what you can observe. A kill action gives you a hard stop and usually a clear incident record. An `errno` action can be harder to diagnose because the application may report a generic failure and keep going. A trap can be useful in test environments because it surfaces the exact point of failure, but it may not be a good production default for noisy workloads. Pick the action that balances security with the observability you actually need, not the one that merely seems strict.

### **Normalize the signal before it reaches developers**

Raw kernel artifacts are rarely the right developer-facing interface. Audit records, SIGSYS crashes, and runtime-specific exception text often contain too much, too little, or the wrong kind of information. A good runner converts those artifacts into a normalized event that answers

five questions: which run was affected, which syscall or syscall family was blocked, during which phase it happened, whether the denial matched expected policy, and whether the event should be classified as user error, software drift, or suspicious behavior.

That normalized event should be stable across environments. If one host emits a seccomp audit line, another emits a signal, and a third routes notification through a supervisor, the downstream developer workflow should still see the same logical event. Stable shape matters more than exact textual fidelity. A runner that emits a consistent schema can support dashboards, alerts, and policy review without exposing the raw kernel payload to every observer.

Keep the schema sparse on purpose. Include a run identifier, a tool identity, a policy version, the architecture if it matters to interpretation, the syscall name, and a coarse execution phase such as initialization, steady state, or shutdown. If the runtime can provide a safe classification of the denied syscall family, keep that too. Avoid command arguments, entire environment blocks, and path strings unless you have a strong, documented reason to include them. Even then, prefer redacted or hashed forms that preserve correlation without revealing content.

### **Correlate by run, not by payload**

The safest way to tie a denial back to a tool invocation is to use a stable run identifier emitted by the tool runner. That identifier should be generated before the workload starts, attached to the invocation record, and carried through any audit or notification pipeline. The same identifier can connect the seccomp denial to the container, the image digest, the policy version, and the execution phase without requiring the logs to contain the actual command line.

This design avoids a subtle but common leak path. If you rely on raw command strings or file paths to correlate events, you will eventu-

ally expose user input, secret-bearing arguments, or customer-specific data to logs that were meant only for security analysis. In agentic or multi-tenant systems, that is a real data handling failure, not a theoretical risk. Correlation IDs solve the operational problem without exporting the underlying payload.

There is also a practical reason to keep the correlation mechanism outside the denied syscall path. The syscall itself may fail for a valid reason, but the runner still needs to publish the event reliably. If the only way to correlate is to inspect the process after it has already been blocked, you may lose the evidence in the very cases that matter most. Preallocated run metadata and a side-channel from the runner to the audit pipeline are more robust than trying to recover the context after the fact.

### **Use notification carefully and treat the supervisor as part of the TCB**

User-space notification is the most flexible seccomp outcome, but it changes the operational shape of the sandbox. A matched syscall no longer ends inside the kernel alone; it is forwarded to a supervisor that must receive the event, validate it, decide on a response, and potentially inject file descriptors or a verdict back into the target. That makes the supervisor part of the trusted computing base for the policy path.

If you use notification, you need to handle the mechanics correctly. The notification channel has its own lifecycle, and the supervisor must deal with variable-sized structures by querying the kernel for the current notification sizes before allocating buffers. It must also validate notification IDs and confirm that the target task is still live before responding. A receive operation can block if the target has already terminated, so the supervisor needs a timeout strategy and a liveness check rather than a blind wait. The longer the policy loop stays open, the more important these failure modes become.

Notification is best reserved for cases where the policy decision genuinely depends on context the kernel cannot see. Examples include delegating access to a brokered file, applying a higher-level business rule, or allowing a request only when an external policy engine approves it. Do not use it as a default substitute for static allowlists. Every extra moving part in the denial path increases the chance that a policy event turns into an availability incident.

### **Log enough to diagnose, not enough to reconstruct secrets**

The cleanest observability model is one in which the denial record helps you answer what changed, not what the user typed. That means you log the minimal state needed to compare one run against another: run ID, image or binary digest, tool name, policy version, syscall name, action, phase, and a coarse verdict such as expected or unexpected. In many cases, that is enough to separate a policy bug from a real regression.

Do not let the convenience of audit tooling push you into overcollection. Raw audit lines often include command fragments, pathnames, and process metadata that are useful to a low-level investigator but excessive for routine developer workflows. If you forward those lines into a searchable system, you create a secondary data store with its own access and retention obligations. That may be acceptable for a security operations team with strict controls, but it is a poor default for broad developer visibility.

A practical rule is to log context at the granularity of the policy decision, not the user input. “Blocked openat during startup under policy seccomp-v12” is usually enough. “Blocked openat on /home/alice/.ssh/id\_rsa after the process read prompt content” is too much. When you need richer detail for a narrow incident, generate it from a controlled debugging path, not from the routine denial feed.

### **Classify denials before tuning the policy**

Not every denial is a bug, and not every bug is a reason to widen the profile. The response workflow should start with classification. Is the call clearly outside the intended contract? Is it a legitimate feature that should remain available? Is it a transient side effect of a runtime or library update? Is it suspicious enough that the denial is doing exactly what you wanted?

That classification can be mostly mechanical if the policy is well maintained. A denial that matches a known prohibited family such as namespace manipulation or process inspection is usually expected. A denial that appears after a dependency upgrade and only during startup may indicate profile drift. A denial that surfaces only under unusual user input may be part of a bug, a fallback path, or an adversarial probe. The operator's job is to distinguish these cases before changing the policy.

The most damaging anti-pattern is to treat every denial as evidence that the profile should expand. If you do that, the policy will steadily lose value as edge cases accumulate. A better habit is to record the reason for each exception and require a review when the exception ceases to be temporary. That keeps the runtime compatibility burden visible instead of hidden inside a growing allowlist.

### **Expect compatibility drift and make it visible**

Seccomp observability is especially important at upgrade boundaries. New kernel versions can introduce new syscall interfaces. New libc releases can change which syscalls a common operation uses under the hood. New container runtimes can alter how denials are surfaced to the operator. If your policy and telemetry are not versioned together, a denial after an upgrade will look like a random failure instead of a predictable contract mismatch.

The fix is to attach policy versioning to the run record and to compare denials against a known baseline. If a workload starts failing after

a runtime or image change, the first question should be whether the syscall shape changed, not whether the user did something wrong. Stable policy versions let you bisect that difference quickly. They also let you distinguish a genuine regression from a profile that was already too loose and merely happened not to fail before.

This is where a lightweight review gate pays off. When a new denial appears, compare the syscall, phase, and policy version against historical runs. If the same tool under the same policy used to succeed, you likely have drift. If the denial was always expected but never surfaced because logging was incomplete, then the observability layer, not the policy, needs improvement. A complete record turns those cases into configuration work instead of incident archaeology.

### **Keep the feedback loop safe for multi-tenant and adversarial settings**

A denial feed can become a covert channel if you let too much detail escape. Attackers can use repeated blocks to infer which syscalls are allowed, whether particular files exist, or how a policy responds to different arguments. That risk is highest when the observability system reflects raw reasons back to the user or exposes detailed audit output to broad audiences.

Reduce that risk by separating operator telemetry from user-facing feedback. Operators can see the normalized event stream with enough context to maintain policy. Users or developers can see a sanitized explanation that tells them a syscall family was denied and which policy version applied, but not the exact internal rules or sensitive surrounding state. If you need richer diagnostics, gate them behind explicit debug modes, scoped access, and retention limits.

The same principle applies to storage. A centralized audit pipeline can be helpful, but it also creates a durable record of denials that may reveal system behavior over time. Retain only what you need for the opera-

tional window you have defined. The more sensitive the environment, the more aggressively you should minimize detail and lifespan.

### **Use denials as product feedback, not as an excuse to generalize policy**

A well-instrumented denial path gives you something better than a stack trace: it tells you which assumptions about the tool were wrong. Sometimes the issue is an over-tight profile. More often, it is an uncatalogued dependency, a probe for kernel features, or a startup-time behavior that never should have been present in the first place. The observability contract should help you separate those cases.

That is why the best workflows treat seccomp denials as structured feedback. The runner records the event, the policy owner reviews the classification, and the profile is either corrected, intentionally left in place, or paired with a narrower compatibility exception. The result is a controlled loop rather than a reactive widening of the allowlist.

When you keep that loop tight, denials become useful. They tell you when a runtime changed, when a dependency drifted, when a feature probe is too ambitious, and when a blocked syscall represents a real security boundary being enforced. A good seccomp deployment does not hide those events; it reduces them to the smallest useful signal and keeps the rest of the system from learning more than it needs to know.

## **5.3. SELinux/AppArmor: MAC as a Second Line of Defense**

A tight seccomp profile narrows which syscalls can reach the kernel, but it does not decide what happens after a permitted syscall opens a file, talks to a socket, or touches a shared object. That gap is where Mandatory Access Control earns its place. If a tool can still read the

wrong path, connect to an unexpected Unix socket, or write into a workspace it should not control, `seccomp` has done its job and then stopped. MAC picks up from there and constrains the resources that the process may touch after admission.

A practical baseline in a containerized fleet usually combines a container runtime MAC setting with per-workload policy. For SELinux-aware runtimes, the container config can carry a mount label that keeps files inside the container domain distinct from host labels:

```
{
  "process": {
    "args": ["tool", "--input", "/work/in"]
  },
  "mounts": [
    {
      "destination": "/work",
      "type": "bind",
      "source": "/srv/work",
      "options": ["rbind", "rw"]
    }
  ],
  "linux": {
    "selinuxLabel": "system_u:system_r:container_t:s0"
  }
}
```

That snippet is not a full policy, and it is not meant to be. It shows the layering point: the runtime can attach a label context, while the policy author still decides which files, sockets, transitions, and object classes the workload may reach. In an AppArmor environment, you would express the same idea differently: a task-centered profile would name the executable and confine it by path-based rules. The mechanism differs, but the operational objective is the same—deny ambient access that a syscall allowlist cannot express cleanly.

**Use MAC where `seccomp` stops: object access, not syscall admission** `seccomp` answers a binary question at the syscall boundary: may this process attempt this kernel entry point? MAC answers

a different question: assuming the syscall is allowed, may the resulting object access proceed? That distinction matters for files, sockets, IPC endpoints, pseudo-filesystems, and helper binaries. A tool may stay entirely inside a narrow seccomp profile and still exfiltrate data if it can read an unexpected mounted path. It may also stay within its syscall budget while connecting to a privileged local socket or invoking a helper with a label or path it should not reach.

You should think of MAC as a policy layer that constrains the shape of the resource graph available to the process. Paths can be writable, readable, execute-only, or inaccessible. Sockets can be confined to specific domains or labels. IPC objects can be narrowed to the channels you expect. That is materially different from syscall filtering because the same syscall may be safe in one context and dangerous in another. `openat` on a scratch directory is one thing; `openat` on a host-mounted secret volume is another. MAC lets you distinguish those cases.

That is especially useful in multi-tenant tool runners. One workload may need read access to its input bundle and write access to a temporary scratch tree. Another may need only a log sink and a single artifact export path. A third may shell out to helpers and therefore needs a much narrower transition model than the default container runtime would otherwise permit. A single seccomp profile cannot encode those differences with enough fidelity. A MAC policy can.

### **Choose the model that matches your operational reality**

SELinux and AppArmor both implement kernel-enforced MAC, but they optimize for different operational styles. SELinux is label-driven and type-oriented. You reason about sources, targets, and permissions in a way that can become very precise, especially in environments that already standardize around SELinux labels. AppArmor is path and task oriented. You attach a profile to a task or executable and express what it may read, write, execute, and mediate by path and capability

classes.

The difference is not just philosophical. It affects authoring, rollout, and troubleshooting. SELinux policy can be more expressive and can integrate cleanly with a larger label discipline, but it usually asks more of the operator: policy modules, type transitions, file relabeling, and a habit of interpreting denials through label context. AppArmor often offers a faster path to a useful profile because the rules are easier to read and the feedback loop is more direct, especially when you are tuning one workload at a time. That simplicity is valuable when you need to confine a diverse set of agent tools quickly.

A good decision rule is to align with the platform you already operate. If your hosts, images, and administrative practice already revolve around SELinux labels, lean into that model rather than layering an AppArmor mindset on top. If your distribution and debugging workflow are already AppArmor-friendly, use that to your advantage. Do not force the team to maintain two competing MAC mental models for the same fleet unless the deployment truly demands it.

**Translate tool intent into resource categories** Policy authoring gets easier when you stop thinking in terms of arbitrary files and start thinking in terms of workload categories. A sandboxed tool usually needs some combination of these:

- read-only input paths,
- bounded writable scratch space,
- explicit artifact export locations,
- limited execution of helper binaries,
- controlled access to runtime sockets,
- constrained IPC domains,

- and defined interactions with pseudo-fsfilesystems.

Those categories are the bridge between a tool specification and a MAC policy. If the tool consumes a user-supplied archive and emits a transformed artifact, the readable scope should be the input area and the writable scope should be only the scratch tree and the export directory. If the tool shells out to a helper, you should decide whether that helper inherits the same domain, transitions to a narrower one, or is forbidden entirely. If the tool needs to inspect runtime state, you should decide which pseudo-fsfilesystem nodes it may touch and whether the access should be read-only.

This translation step is where many overbroad policies begin. Operators look at a runtime image and grant access to the whole workspace because the tool is “supposed” to manage files there. That attitude dissolves the boundary between what the tool must do and what the tool happens to be able to do. A better policy treats the workspace as a set of named subdomains: input, scratch, export, and maybe a tightly controlled cache. That makes the rules legible and the review process tractable.

**Favor narrow per-tool profiles over shared supersets** The temptation in a fleet is to create one giant profile that covers every tool class. It feels efficient because you only have one object to maintain. In practice it is usually a hidden permission expansion. The noisiest workload defines the baseline, and every quieter workload inherits the same excess rights. That is exactly the wrong direction for a defense layer meant to complement seccomp.

A per-tool policy gives you sharper failure signals. If a PDF converter fails because it cannot read a helper font cache, you know where to look. If a compiler helper fails because it cannot write a scratch file, you can decide whether the scratch path belongs in the contract or the build

system should stop depending on it. A shared profile makes those failures ambiguous, because it blurs unrelated behaviors into one policy envelope.

Per-tool specialization also helps with drift. One image may update its helper binaries more often than another. One runtime may touch a different set of sockets. One deployment may use a read-only root filesystem while another still writes local cache files. If you collapse those into one policy, every change in one workload becomes a maintenance burden for all of them.

**Use AppArmor when you want a fast, iterative workflow** AppArmor is often the faster path when you need immediate operational value and the workload can be described by path rules. A task without a profile runs unconfined, so the deployment choice is explicit: either you attach a profile or you do not get MAC beyond normal discretionary access control. That makes the rollout model easy to understand, but it also means the policy must be loaded from userspace before enforcement is meaningful.

The practical advantage is the iteration loop. Denied operations are logged, complain mode lets you observe missing permissions without blocking, and tools such as `aa-logprof` and `aa-enforce` help you move from observation to enforcement. That workflow is useful when you are introducing MAC to a fleet that already has many moving parts and you need a low-friction way to discover what the tool actually touches. You can start permissive, collect denials, trim the profile, and then turn enforcement on once the policy stabilizes.

The cost of that ease is that you still have to be disciplined about rollout. Complain mode is not a permanent state. If you leave a profile there indefinitely, you have telemetry but no confinement. You should treat the logged denials as policy design input, not as a substitute for the pol-

icy itself. In a multi-tenant environment, a profile that never graduates out of complain mode can give operators a false sense of hardening.

**Use SELinux when label discipline and richer semantics matter** SELinux gives you a more explicit security model around labels, types, and transitions. That precision is useful when you need stronger separation among host objects, container domains, and shared system services. It also plays well with environments that already standardize on label-based administration. In those settings, a container runtime can carry a SELinux mount label or process label, and the rest of the policy can remain consistent with host-level labeling practice.

The power of SELinux is also why its policy management feels heavier. You manage modules, port and file labels, transition rules, and the resulting allow matrix. You may use tools such as `semodule`, `semanage`, `audit2allow`, `audit2why`, `restorecon`, and `setenforce` to inspect, adjust, and activate policy. That toolchain gives you fine-grained control over how denials are translated into policy changes, but it also means you need a real policy lifecycle rather than ad hoc exceptions.

A disciplined SELinux rollout usually starts with a narrow domain for the tool, then explicit file and process transitions, then relabeling or mapping the specific paths that the tool must touch. If you let a workload inherit broad host labels by default, you lose the benefit of the model. SELinux shines when the labels reflect actual trust boundaries. If the labeling is sloppy, the policy becomes hard to reason about and easier to overextend.

**Package the policy as part of the workload contract** MAC is most maintainable when the policy lives with the workload definition. That means you version the profile alongside the image, the runner config, or the deployment manifest. If the tool changes its helper layout, its cache behavior, or its artifact export path, the policy change should

travel through the same review path as the application change.

For AppArmor, that usually means a profile file and a deployment step that loads it before the workload starts. For SELinux, it often means one or more policy modules plus any required file-context adjustments. In either case, the policy should be reviewed as code. You want a diff that shows what access was added, why it was added, and whether the new permission is temporary, permanent, or specific to one runtime class.

The packaging boundary matters because policy drift is otherwise invisible. If the tool image evolves but the MAC profile stays static, the first symptom is a denial in production. If the policy evolves independently of the workload, you can accidentally widen access without a clear trace back to the application change that justified it. Binding them together makes the security contract auditable.

**Treat denials as evidence, not just as errors** A MAC denial is a policy signal. Sometimes it indicates a real misconfiguration. Sometimes it reveals an unexpected dependency. Sometimes it shows that the workload is probing a path or object it should never have reached in the first place. Your response should depend on which of those cases you are seeing.

AppArmor is often friendlier for this kind of triage because the log output is direct and complain mode lets you collect missing permissions before enforcement. SELinux can be equally actionable, but the operator must be comfortable reading type and label context, and the remediation path may involve relabeling or module updates rather than a single rule edit. In both cases, avoid the reflex to disable enforcement as soon as a denial appears. First determine whether the denied access belongs in the workload contract.

This is where cross-layer reasoning matters. A denied file open may

actually be evidence that the seccomp profile is working too, because the tool is trying to reach a resource outside its intended scope. Conversely, an allowed syscall with a denied MAC access usually means the syscall filter was necessary but insufficient. That is a healthy outcome. It tells you the layers are doing different jobs.

**Manage profile rollout in noisy fleets** Multi-tenant fleets amplify small mistakes. One overly broad exception can spread quickly if it is embedded in a shared base image. One permanently permissive profile can hide the fact that the workload is changing. One mislabeled directory can break many runs at once. The rollout process therefore needs explicit staging and clear ownership.

A sensible sequence is to introduce MAC in audit or complain mode where the platform supports it, capture the denials from representative workloads, refine the policy, and then move to enforcement on a small slice of traffic or a subset of nodes. That gives you time to separate true access requirements from incidental probes caused by package managers, language runtimes, or helper binaries. It also gives you a safe place to catch path drift when images or mount layouts change.

For SELinux, the staging process should include relabel verification, because a policy that references the wrong labels will fail in ways that look like ordinary file errors. For AppArmor, the staging process should include realistic path coverage and helper invocation coverage, because a rule set that only models the main executable will miss the shell-outs and helper binaries that often carry the interesting behavior. In both cases, the rollout should be reversible without editing the policy by hand on a live system.

**Watch for the common failure modes** The easiest way to weaken MAC is to treat it as a compatibility layer instead of a confinement layer. Three mistakes recur. First, teams ship one

giant shared policy for unrelated tools. Second, they leave policies in complain or permissive mode long after the tuning period ends. Third, they grant broad filesystem access because a runtime is dynamic and difficult to characterize.

All three failures are understandable, and all three erase most of the value of the layer. Shared policies are convenient until one workload expands the entire fleet's access. Permissive modes are useful during tuning but pointless as a steady state. Broad access is tempting when a language runtime shells out often, but it usually means you have not separated the true contract from the implementation details of one library version.

A stronger posture is to accept some policy work up front. Narrow the readable paths. Separate scratch from export. Name the helpers explicitly. Keep the logging precise. Use the policy to make the tool's resource contract visible, not to encode every possible path the process might stumble into.

**Let MAC define the object boundary seccomp cannot** The practical value of SELinux and AppArmor is not that they duplicate seccomp. It is that they fill in the object-level boundary after syscall admission. Files, sockets, IPC channels, helper binaries, and labeled resources all sit outside seccomp's expressive sweet spot. MAC gives those resources structure. In a containerized tool runner, that means you can let the process make the system calls it needs while still denying access to the wrong files or the wrong peers.

That combination is what makes the second line of defense worthwhile. seccomp reduces the set of kernel entry points; MAC reduces the set of objects those entry points can meaningfully reach. When you align the policy with the workload, package it with the deployment, and stage it carefully across the fleet, the result is a boundary that is narrow enough

to matter and operationally clear enough to maintain.

## 5.4. Landlock for Unprivileged Self-Confinement

When you cannot depend on a privileged orchestrator to configure every sandbox boundary, the process itself can still narrow its own reach before it handles untrusted input. That is the practical value of Landlock. It does not replace container isolation, seccomp, or host-managed MAC; it gives you a way to add a filesystem-oriented restriction layer from inside the process, even when you do not control the outer environment. In agent systems, that matters whenever a tool wrapper, plugin host, or embedded runtime needs to harden itself before parsing content from a user, a network service, or another tool.

A minimal runtime probe looks like this:

```
#define _GNU_SOURCE
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <linux/landlock.h>

#ifdef LANDLOCK_CREATE_RULESET_VERSION
#define LANDLOCK_CREATE_RULESET_VERSION 1U
#endif

static int landlock_version(void) {
    return syscall(__NR_landlock_create_ruleset,
                  NULL, 0,
                  LANDLOCK_CREATE_RULESET_VERSION);
}

int main(void) {
    int ver = landlock_version();
    if (ver < 0) {
        perror("landlock_create_ruleset");
        return 1;
    }
    printf("Landlock ABI version: %d\n", ver);
}
```

```
    return 0;  
}
```

That probe is the first decision point. If the kernel does not support the Landlock ABI you expect, you can fall back cleanly instead of assuming a policy that will never enforce. If the ABI is present, you still need to enable only the rights that the kernel actually supports. That runtime negotiation is part of Landlock’s design: you query capability first, then build a ruleset that matches the kernel’s current feature set.

### **Landlock is self-confinement, not privilege creation**

Landlock lets a process restrict itself and its future descendants. It does not grant access, and it does not expand privilege. That self-restriction model changes the deployment story. You can apply it from an unprivileged process when `no_new_privs` is set, or when the caller has the required capability in its user namespace. That makes it especially useful in environments where you cannot rely on a host operator to install a bespoke profile for every tool invocation.

The monotonic model is the key property. Once you add restrictions, you cannot relax them later in the same thread tree. That makes Landlock suitable for narrowing trust after startup initialization but before handling untrusted data. A wrapper can open the files it truly needs, then apply Landlock, and then parse the input. The important detail is timing: the restriction only applies to future access decisions. If a file descriptor is already open, Landlock does not retroactively strip that access away. That means you should open trusted resources first, then seal the process before it touches anything derived from untrusted input.

This is a stronger design constraint than it first appears. It forces you to separate initialization from steady-state work. If a tool needs broad filesystem access during startup but only narrow access during processing, Landlock fits well. If the tool lazily opens configuration, caches,

or plugins after parsing user input, the policy will need to account for those accesses explicitly or move the open earlier.

### **Probe the ABI and enable only the rights you need**

The practical deployment pattern starts with capability discovery and ends with a ruleset that names only the rights the workload actually requires. Landlock is stackable, so multiple restrictions compose monotonically. That is useful when different parts of a program contribute different policy fragments, but in practice you usually want to consolidate those fragments into one tailored ruleset. A single reviewable policy is easier to audit than a set of incremental restrictions whose combined effect you have to reconstruct mentally.

A typical flow is:

- Query the ABI version.
- Enumerate the rights supported by that version.
- Decide which actions the tool needs on which paths.
- Create a ruleset that grants only those actions.
- Attach the ruleset before processing untrusted input.

That sequence matters because the ABI is not static across kernels. If you write against a newer right and deploy to an older kernel, the policy may fail to load or silently omit the right if you do not check carefully. A good wrapper treats unsupported rights as a feature-negotiation problem, not as a reason to assume the policy succeeded. In a fleet, that distinction keeps partial kernel support from becoming a latent security regression.

The implementation style is also different from `seccomp`. With `seccomp`, you usually think in terms of allowlisted syscalls and chosen

denial actions. With Landlock, you think in terms of objects and access classes. That is a better fit for file-centric workloads: read-only inputs, bounded scratch trees, explicit export paths, and inherited execution trees that should not grow beyond the narrow set you already approved.

### **Use Landlock to model file and path boundaries**

Landlock is especially effective when the risk is exfiltration or unauthorized mutation through the filesystem. A tool that reads one input directory, writes to one scratch tree, and exports one artifact path is exactly the kind of workload Landlock was built to constrain. You express the accessible paths and the permitted operations, then deny everything else.

That makes the policy naturally aligned with agent tool runners. The runner can mount or expose a minimal workspace, and the tool can further confine itself to the subset it actually needs. If a prompt-injected plugin tries to read neighboring files or wander into the parent directory tree, Landlock can stop that access even if the process is still running inside a container with a permissive runtime profile.

The path model is monotonic in another sense too: once a process is confined, it cannot later decide that a broader directory should be trusted. That favors a design in which all writable scratch space is explicit from the start. You should name export directories, temp directories, and cache directories directly instead of letting the process discover them opportunistically. The more explicit the paths, the easier the policy is to keep narrow.

### **Understand what Landlock does not cover**

Landlock is not a general sandbox boundary. It is path-centric and filesystem-centric, which is precisely why it is valuable and why its limits matter. It does not retroactively restrict already-open file descrip-

tors. It cannot make an inherited descriptor safe after the fact. It also does not replace the need for outer controls on process creation, IPC, or syscall filtering. If a process can still invoke dangerous kernel interfaces, seccomp remains relevant. If it can still touch host objects beyond the filesystem, MAC remains relevant.

The current kernel documentation also calls out special filesystem gaps and topology constraints. Some special kernel filesystems and `proc-fd` style access paths are not fully covered in the same way as ordinary path access. That means you should not assume Landlock can express every resource boundary the way a host-managed MAC policy can. In practice, the best candidates are ordinary filesystem trees that you control: input directories, scratch directories, artifact export directories, and internal workspaces.

Those limitations shape the trust model. Landlock is strongest when the process already has a minimal outer environment and you want an additional self-imposed barrier around file access. It is weaker when the process needs to mediate complex object graphs or special kernel interfaces. That is why it fits naturally as an inner layer in this chapter, after seccomp and alongside host-managed MAC, not as a substitute for them.

### **Model the policy around the tool's phase structure**

The cleanest Landlock deployment separates startup from steady state. During startup, the wrapper may load configuration, resolve plugins, populate caches, or pre-open trusted resources. After that phase, the wrapper applies Landlock and hands control to the untrusted or semi-trusted workload. That pattern lets you keep the policy narrow without forcing the application to relearn its own bootstrap sequence.

This phase split is especially important in language ecosystems where helper processes, dynamic modules, or shell-outs are common. If you wait too long to confine the process, the application may open files that

should never have been reachable during the input-processing phase. If you confine too early, you may break legitimate initialization. The design question is not whether to apply Landlock, but which phase boundary you want to treat as the last trusted moment.

A useful mental model is to ask which resources are truly static. Configuration that never changes during a run can be loaded before confinement. Inputs from the user, derived files, and generated artifacts should be governed by the ruleset. If the application needs to discover new paths after confinement, that is usually a sign that the program contract is not explicit enough yet.

### **Combine Landlock with seccomp and MAC rather than choosing one**

Landlock is not in competition with seccomp or SELinux/AppArmor; it fills a different gap. Seccomp limits what kernel entry points the process can reach. MAC constrains which objects the process may access after it has entered the kernel. Landlock lets the process impose an additional filesystem boundary on itself. Those layers compose well because they attack different failure modes.

That composition is especially valuable when the outer environment is incomplete. You may be running in a container where the runtime seccomp profile is broad, the host MAC policy is unavailable, or the platform operator has no reason to trust every tool-specific policy. In that case, Landlock gives the workload some self-service confinement instead of leaving it entirely dependent on host configuration. You can still add a better seccomp profile and host MAC when available, but the process does not have to wait for those controls to exist before it narrows its own reach.

This layering also helps with least-privilege reasoning. If a tool must read a dataset from one directory, the seccomp profile can keep the kernel surface small, the host MAC policy can prevent the process from

wandering into unrelated labeled objects, and Landlock can prevent the tool from escaping its own declared workspace. A failure in one layer does not nullify the others. That redundancy is exactly what you want when the risk is accidental misbehavior as much as deliberate escape.

### **Treat ruleset composition as a design choice, not a convenience**

Because Landlock rulesets compose monotonically, you can build policy in stages. Different components can contribute their own restrictions, and the process inherits the combined effect. That sounds convenient, and it is, but it also makes policy reasoning more difficult if you let the fragments accumulate informally. In most deployments, a single tailored ruleset is easier to review and less error-prone than a sequence of partial updates from different subsystems.

The reason is simple: every additional fragment is another place where the policy can diverge from the intended contract. One module may assume a scratch directory is writable; another may assume the same tree should be read-only after startup. If those assumptions are merged by accident, debugging becomes unnecessarily hard. A consolidated ruleset forces you to settle the contract in one place and document the operational intent explicitly.

That said, incremental construction can still be useful when different libraries or wrappers own distinct stages of setup. The rule of thumb is to use composition when you need clear ownership boundaries and to collapse the final state into a single reviewable policy before untrusted work begins. The process should leave the startup phase with a completed policy, not with a stack of half-understood restrictions.

### **Prefer filesystem-centric workloads that benefit most**

Landlock gives the best payoff when the dominant risk is filesystem

misuse. Tool runners, document processors, archive extractors, code analyzers, and content transformers are all strong candidates. These workloads often need a narrow writable area and a few read-only inputs, which makes the path-based model a close fit. They also tend to run inside larger agent systems where the surrounding environment is useful but not fully trustworthy.

The model is less attractive when the core risk is not filesystem access. If the workload is primarily about socket mediation, process control, or label-based object access, host-managed MAC is usually the better tool. If the workload needs to remove kernel attack surface broadly, seccomp is still the first line of defense. Landlock shines when the question is, “Can this process read or modify that path after it has started handling untrusted content?”

That use-case focus also explains why it is so useful in less privileged environments. A plugin host in a developer tool, a per-request helper in a SaaS system, or an embedded converter launched by a larger agent can all apply Landlock without asking the platform for new privileges. That is a practical advantage when deployment speed matters and the operator cannot afford a host-wide policy cycle for every small tool.

### **Watch the failure modes that matter in production**

The most common mistake is to apply Landlock after the sensitive descriptors are already open. At that point, the policy may still protect future opens, but it cannot undo what the process already possesses. The second mistake is to assume that Landlock covers special kernel filesystems, inherited descriptors, or topology-changing operations as broadly as ordinary file trees. It does not. The third mistake is to treat self-confinement as a substitute for external audit. It is not.

There is also a deployment pitfall around runtime portability. If you assume the newest ABI everywhere, older kernels will fail to enforce the policy you thought you had. That is why version probing is mandatory.

A robust wrapper should query the ABI, enable only the rights that the kernel actually supports, and fail visibly if the minimum required feature set is absent. Silent weakening is the wrong outcome.

When you avoid those traps, Landlock becomes a very practical control. It is easy to place after initialization, cheap to reason about compared with a sprawling host policy, and strong enough to meaningfully reduce the blast radius of file-centric mistakes. Inside a container, that self-imposed boundary can be the difference between a narrow error and a broad exfiltration path.



## Chapter 6

# MicroVMs for Stronger Isolation and Multi-Tenant Safety

*A microVM is not just a smaller virtual machine; it is a security decision about where the kernel boundary belongs when agent-invoked tools are allowed to process hostile inputs, operate under automation, and share infrastructure with other tenants. This chapter shows how to gain the isolation benefits of separate guest kernels without treating microVM adoption as a magic shield: the real work lies in constraining the VMM, shrinking interfaces, and designing execution flows that remain fast, disposable, and scope-driven.*

## 6.1. Why VM Boundaries Matter for Untrusted Tools

A hardened container runtime narrows risk, but it does not change the fact that the workload and the host still share one kernel. When you let an agent run arbitrary parsers, browser automation, code generators, or document converters, the shared kernel becomes the common failure domain for every tenant on the machine. That is acceptable for many internal services; it is a different claim entirely when the workload itself is untrusted, attacker-influenced, or able to select its own inputs at runtime. A guest kernel changes that equation by moving the primary isolation boundary out of the host kernel and into a per-workload virtual machine. The workload can still be compromised, but the compromise no longer begins and ends with direct access to the host kernel ABI. It has to cross a virtualization boundary whose attack surface is narrower and whose mediation points are easier to enumerate.

```
# A minimal Firecracker-style launch flow
# 1. Boot the microVM with a guest kernel and rootfs
# 2. Attach only the devices the tool run needs
# 3. Inject input, execute, collect output, then destroy

fc-cli --api-sock /run/fc.sock \
  --kernel /images/vmlinux \
  --root-drive /images/rootfs.ext4 \
  --tap-device tap0 \
  --vsock-id 5

curl --unix-socket /run/fc.sock \
  -X PUT http://localhost/actions \
  -d '{"action_type":"InstanceStart"}'
```

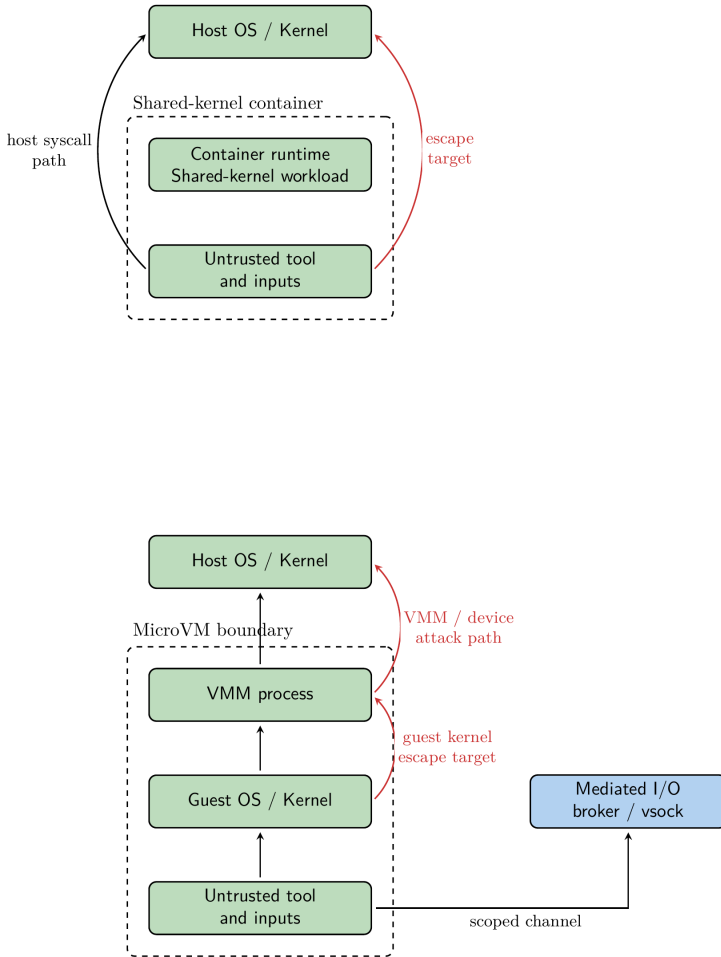
The practical value of that launch pattern is not the command itself; it is the boundary it creates. A host-side broker chooses the image, starts the microVM, injects only scoped inputs, and tears the instance down after the run. The guest gets a kernel of its own, a minimal device set,

and a much smaller set of host-visible channels than a containerized process would normally inherit. The control plane still matters, and the VMM still matters, but the host kernel is no longer the immediate target for every guest exploit.

### **Shared-kernel containment versus guest-kernel isolation**

Container isolation works by partitioning a single operating system: namespaces reshape what a process can see, cgroups constrain what it can consume, and mandatory access control reduces ambient privilege. Those mechanisms are useful, but they remain overlays on the same host kernel. If the guest workload finds a kernel bug in a filesystem path, a network path, a packet parser, or a namespace transition, the exploit path terminates in the host kernel itself. For benign services, that may be an acceptable residual risk. For hostile code execution, especially in multi-tenant settings, the shared kernel becomes the thing you are betting against.

A microVM changes the threat model by moving the guest workload behind its own kernel. The host still runs a kernel, but the workload no longer speaks to it directly. It speaks to a small VMM process, emulated or mediated devices, and a guest kernel that is only as trusted as the image pipeline and boot path allow. This is why microVMs matter for untrusted tools: they do not eliminate kernel bugs, but they make the first successful compromise land inside the guest boundary rather than inside the host kernel boundary.



That boundary shift is the whole point, but it is easy to oversell it. The guest kernel is not a magic shield. It is a different trust boundary, and it introduces new trusted components that must be kept intentionally small: the VMM, the launch broker, the image pipeline, the device

model, and any brokered path that injects files, credentials, or network access. The more you rely on the microVM boundary, the more carefully you need to define each of those components.

### **Where the risk actually moves**

In a containerized design, a successful escape often means arbitrary code execution in the host kernel context or a privilege transition that lands very close to it. In a microVM design, the first-stage target is usually the guest kernel. That is a meaningful improvement when the workload is hostile, because the guest kernel can be treated as disposable and can be given a far narrower interface to the host. The attacker may still reach the VMM, but that path is narrower than the full host syscall surface and device stack.

That does not make the VMM irrelevant. It becomes part of the trusted computing base, and if you expose too much device emulation, too many host services, or too many control APIs, you rebuild a large attack surface by accident. The reason specialized microVM systems exist is precisely to avoid that mistake. They keep the machine model tiny, reduce the number of emulated devices, and force host interaction through explicit channels such as block I/O, network mediation, and vsock-like control paths. For untrusted tools, that reduction is valuable because it turns an open-ended operating system boundary into a purpose-built execution envelope.

The dominant failure mode also changes. With containers, the main concern is often a kernel exploit that spreads laterally or reaches the host. With microVMs, the concern becomes one of boundary abuse: does the guest get a path to the host that is broad enough to matter? Does the VMM expose unnecessary device behavior? Does the launch broker hand out credentials that outlive the run? Does the image pipeline let a compromised guest image become a persistence vehicle? Those are different questions, and the answers require different

controls.

### **When the stronger boundary is warranted**

A guest kernel is worth the cost when the workload is both risky and difficult to fully trust. The clearest cases are workloads that parse attacker-controlled content, execute code selected by users or agents, or interface with formats that historically accumulate memory-safety bugs. Document conversion, archive unpacking, PDF rendering, browser automation, media inspection, and language-model tool execution all fit that pattern. So do multi-tenant platforms where one customer's tool run must not inherit another customer's kernel attack surface. The stronger boundary is also justified when regulatory or contractual isolation requirements call for a separation story you can explain clearly to auditors: one tenant, one guest kernel, one limited control path, no shared host-kernel execution domain.

The boundary is less compelling when the workload is narrow, predictable, and already constrained by a strong application-layer protocol or when startup overhead would destroy the operational design. A microVM is a control-plane decision as much as a security decision. If the tool is simple and the host is already isolated from the tenant by another strong boundary, the operational cost may outweigh the gain. That cost includes launch latency, image management, snapshot compatibility, host virtualization dependencies, and the added complexity of meditating every guest-host interaction.

The question is not whether microVMs are “more secure” in the abstract. They are. The question is whether your threat model justifies moving the boundary when you account for the new trust base you have to own.

### **New trust boundaries you must account for**

A practical microVM stack introduces at least five meaningful bound-

aries.

First, there is the host virtualization layer. The host kernel must support hardware virtualization correctly, and the CPU features it relies on must be available and current. If the host kernel, microcode, or virtualization extensions are weakly maintained, the guest boundary loses much of its value.

Second, there is the VMM process. This process mediates guest execution and device behavior, so its memory safety, parser correctness, and seccomp profile matter directly. A specialized VMM is attractive because it narrows this surface, but it does not remove it.

Third, there is the guest image pipeline. You do not want arbitrary, mutable root filesystems being launched as if they were trusted artifacts. Signed images, reproducible boot assets, and provenance checks reduce the chance that “the guest” becomes a covert persistence layer or a way to smuggle in unexpected privileges.

Fourth, there is the host-side launch broker or control plane. This is the component that decides which image, which resources, which credentials, and which network policy a tool run receives. It is not just orchestration; it is policy enforcement. If it can be tricked into widening scope, the microVM boundary will not save you.

Fifth, there are mediation services for network, filesystem, metadata, and secrets. Any service that injects input into the guest or extracts output from it can become an attack path if it is too permissive. A file import service that mounts a shared host directory into many guests is not a neutral convenience. It is a data boundary with real blast-radius implications.

Those boundaries are not theoretical. They are the places where microVM deployments usually fail in practice: not because the guest kernel was absent, but because the host channels were too generous.

## **Operational conditions that sharpen the case for microVMs**

The first condition is hostile or semi-hostile input. If the tool consumes untrusted files, user-supplied web content, arbitrary archives, or generated code, you should assume the guest will eventually be reached by malicious payloads. A guest kernel makes that compromise less likely to become a host compromise.

The second condition is multi-tenancy. The moment two different tenants share hardware, the question becomes not only whether the workload is isolated from the host, but whether the compromise of one tenant can materially affect another. A guest kernel gives you a cleaner per-tenant story than process isolation alone.

The third condition is autonomous agent behavior. Agents do not always follow a human operator's intuitive path. They may retry, branch, inspect, and fetch in ways that expand the executed input surface. If the agent can invoke tools on its own, the sandbox boundary has to assume the tool chooser is not perfectly aligned with the operator's mental model. MicroVMs are useful precisely because they limit the damage from unexpected tool selection.

The fourth condition is high bug density in the execution workload. Browsers, PDF engines, archive libraries, interpreters, and codec stacks are all rich targets. If the workload's dependency chain is long and the input grammar is complex, the probability of a memory-safety or parser bug rises enough that shared-kernel containment becomes a weaker proposition.

The fifth condition is blast-radius intolerance. Some deployments can tolerate a host-level incident. Others cannot. If a successful escape would mean access to secrets, customer data, or a shared control plane, the additional boundary is often justified even if it costs you more to operate.

### **What the guest boundary does not buy you**

A guest kernel does not give you blanket trust in the guest. The guest can still be owned, and a compromised guest can still abuse every interface you exposed to it. If you attach a writable host volume, give it broad network egress, or hand it long-lived credentials, the guest can use those gifts as effectively as any container could. The microVM boundary only constrains the path from guest compromise to host compromise; it does not constrain the guest from misusing delegated authority.

It also does not absolve you from hardening the VMM. People often assume the guest boundary is enough and then leave the control plane broad, the API socket accessible, or the device model feature-rich. That is how strong isolation gets diluted back into general-purpose virtualization. You want the exact opposite: a very small set of host-visible operations, each tied to a declared scope.

Signed images are another common source of overconfidence. A signed image proves origin and integrity at build time. It does not prove that the guest runtime is benign, that the tool will not be exploited after boot, or that the guest will not misuse network or filesystem access. The image is trusted only to the extent that the policy around it is trusted.

### **Compatibility and version sensitivity**

MicroVM capability depends on the host kernel, CPU virtualization features, guest kernel support, and the chosen VMM release. Snapshot behavior, device availability, vsock implementation details, and performance characteristics all vary across versions. A design that is safe and fast on one host family may be fragile on another, especially if the fleet mixes CPU models or kernel builds. You should treat these dependencies as part of the platform contract, not as implementation trivia.

That version sensitivity affects security as much as performance. If host and guest kernel versions drift too far, you can lose snapshot portability, break device assumptions, or create unstable launch behavior that encourages operators to widen permissions “just to make it work.” Stability discipline matters because operational friction often leads to unsafe shortcuts.

### **Decision criteria that map to a real deployment**

When you decide whether the guest boundary is worth the operational cost, ask four questions.

How much do you trust the input? If the answer is “not much,” a guest kernel is a strong candidate.

How bad is a successful escape? If the answer includes shared secrets, regulated data, or cross-tenant impact, the boundary is usually warranted.

How much complexity can the platform support? If the team cannot maintain image provenance, launch brokering, and device mediation, the security gain will be eroded by operational drift.

How much interaction with the host does the workload truly need? If you can keep the guest’s channels narrow, the microVM boundary becomes much more valuable than if you plan to expose broad file, network, and metadata access.

These are not abstract questions. They force you to align isolation strength with the actual tool risk profile instead of with a generic preference for “more security.”

### **Common anti-patterns**

One anti-pattern is to trust the guest because the image is signed and then allow runtime customization through broad mounts, mutable overlays, or ad hoc package installation. That turns a controlled artifact

into a semi-persistent system with unclear provenance.

Another is to treat the VMM as outside the threat model. If you do that, you will likely leave its API socket too reachable, expose more device functionality than necessary, or run the process with more privilege than the workload deserves.

A third is to confuse isolation with privilege. A stronger boundary does not justify broader host access. In fact, it should make scope stricter, because the guest is now the place where untrusted execution happens and the host is the place where policy must remain intact.

A fourth is to ignore the mediation layer. If file import, secret injection, or outbound network access is too permissive, the guest can still accomplish damaging actions while staying perfectly inside the microVM boundary. The boundary is only as strong as the narrowest channel you actually enforce.

For untrusted tools, the guest kernel is worth the added complexity when you need to contain hostile execution, separate tenants cleanly, and preserve host integrity under unpredictable workloads. The gain is not abstract; it is the difference between sharing an operating system with the tool and placing that tool behind a purpose-built boundary whose remaining channels you can inspect, restrict, and account for explicitly.

## **6.2. Designing MicroVM Lifecycles for Fast Boot and Ephemeral Tool Runs**

A microVM only pays off for tool-running agents when you can launch it quickly enough that isolation does not feel like friction. The scheduling unit is not a long-lived server but a narrow execution slice: one task, one scope, one teardown. That changes the lifecycle problem. You are

no longer optimizing for persistent uptime or elastic service discovery; you are optimizing for deterministic startup, bounded state, and reliable disposal under repeated invocations. In practice, the difference between a safe design and an awkward one is often whether you can make the guest behave like a disposable function rather than a miniature pet VM.

```
# A representative per-invocation launch pattern
# Keep the guest image immutable and the run state disposable.

fc-cli --api-sock /run/fc.sock \
  --kernel /images/vmlinux \
  --root-drive /images/rootfs.ext4 \
  --tap-device tap0 \
  --vsock-id 7

curl --unix-socket /run/fc.sock \
  -X PUT http://localhost/actions \
  -d '{"action_type":"InstanceStart"}'

# Inject only scoped input and collect only declared output.
```

That shape matters because the lifecycle determines both latency and residual risk. A short-lived guest that starts from a minimal image, receives a bounded input set, writes to disposable storage, and exits cleanly is far easier to reason about than a general-purpose VM that happens to be launched on demand. The operational goal is to keep every run reproducible enough to audit and isolated enough to destroy without hesitation.

### Define the run as an immutable contract

The first lifecycle decision is to treat each invocation as a contract with explicit inputs, outputs, and limits. You should know which tool is being run, what files or arguments it receives, which network destinations it may reach, what resource ceilings apply, and what artifact classes are allowed to leave the guest. Once those fields are fixed, the microVM can be assembled from a known image and a short-lived overlay or scratch area rather than from mutable state.

That contract should drive the image shape. A microVM intended for ephemeral tool runs does not need a broad userspace, a package manager cache, a full desktop stack, or a general-purpose service manager. Every additional package increases boot work, expands the filesystem, and creates more state to scrub. Minimal guest userspace reduces attack surface and shortens boot time at the same time. The guest should boot into the exact runtime it needs, not into a shell environment that makes later drift tempting.

The same rule applies to boot-time configuration. Prefer deterministic initialization over general-purpose startup scripting. If a tool needs a single binary, a small runtime library set, and a few environment variables, bake those into the image or pass them as immutable boot parameters. Every runtime script you add becomes a branch in the lifecycle and a place where failures can leak state across invocations.

### **Make cold boot cheap enough to be viable**

Cold boot remains the simplest lifecycle path because it is easiest to reason about. A tiny image, a narrow device model, and a guest configured for the target workload can reach application code quickly enough for many agent workflows. The published Firecracker measurements are the right mental anchor: boot to application code in under 125 ms in a tuned setup, with under 5 MB of memory overhead per containerized workload and creation rates high enough to support bursty launch patterns. Those numbers are not a guarantee for every stack, but they show that a microVM need not be slow by construction.

Boot speed, however, depends on more than the VMM. Guest kernel configuration has a large effect on startup latency. A generic kernel with broad driver support can spend time probing hardware, initializing subsystems that the workload does not need, and emitting console output that is useful for interactive diagnostics but wasteful for ephemeral execution. A lean kernel with only the required drivers and

a trimmed boot path reduces that cost materially. When you see a microVM boot slowly, do not assume the VMM is the culprit. The guest image is often the larger variable.

Example outcome for a tuned guest boot:

- kernel starts quickly
- only required drivers initialize
- application enters its main path
- no interactive shell remains resident

That result is what you want for short-lived tool runs. The guest should spend its time doing work, not preparing to be a workstation.

### **Use snapshot and restore where the compatibility story is clear**

Snapshot/restore is the strongest latency optimization available when the workload and host fleet support it. Instead of booting from scratch, you pre-initialize a guest to a useful point and restore that state when a run starts. That can cut startup time sharply, especially when the workload requires expensive initialization such as language runtime warmup, certificate loading, or large static tables. The key constraint is that snapshots are compatibility-bound artifacts, not generic images. They depend on the guest kernel, the VMM, and the host CPU model.

You should treat a snapshot as a captured execution state with a narrow restore contract. Snapshots are not portable across CPU architectures, and they can fail or misbehave across CPU models unless the exposed feature set is effectively invariant. The operational implication is simple: if your fleet is heterogeneous, snapshot pools need scheduler awareness. If you cannot guarantee homogeneous hosts, you should not assume restore will be universally safe. In a mixed fleet, cold boot may be more boring but far more dependable.

Snapshot timing also matters. A snapshot taken after authentication, after tool discovery, or after arbitrary user input can preserve sensitive

or stale state that should not survive between runs. The safest pattern is to snapshot only from a clean, pre-scoped, pre-credentialed state or from a state whose residual contents you fully control. If the guest contains mutable data that you would not want another invocation to inherit, do not treat it as a reusable snapshot base.

### **Warm pools reduce jitter but increase state discipline**

Warm pools sit between cold boot and snapshot restore. You keep a small number of already-running microVMs ready to accept work, which reduces tail latency and smooths bursty demand. That is attractive for agent systems that issue synchronous tool calls and cannot wait for a cold launch on every request. The trade-off is that every warm guest is a potential carrier of residual state. Filesystem caches, environment variables, open sockets, process memory, package-manager artifacts, and per-run credentials all become things you must scrub or reinitialize.

Warm pools therefore demand stronger lifecycle hygiene than they first appear to. A pool is safe only if you can prove that each recycled guest returns to a known baseline. That usually means guest-local cleanup hooks, host-enforced identity rotation, strict storage reinitialization, and aggressive rejection of runs that do not restore cleanly. If the platform team cannot make that guarantee, the warm pool becomes an optimization that quietly erodes isolation.

For many agent workloads, the right answer is a small pool of ready-to-launch images plus cold start or snapshot restore on demand. That keeps the control plane simpler while still reducing the worst launch spikes.

### **Separate reusable image content from per-run state**

A recurring lifecycle mistake is to let image content and per-run state blur together. The base image should contain reusable code, fixed de-

dependencies, and a small amount of deterministic configuration. Per-run state should hold the tool inputs, ephemeral credentials, scratch output, and any temporary working files. If those layers mix, you lose the ability to reason about provenance and teardown.

The cleanest implementation pattern is an immutable root image plus one disposable data path. That data path can be a blank block device, an overlay layer, or a scratch volume with a lifetime tied to the invocation. The important rule is that the guest must not rely on that data path for durable identity. If you need to preserve artifacts, export them explicitly at the end of the run. If you need to persist logs for debugging, store them on the host after the guest exits, not in a guest volume that may get reused by accident.

This separation is also what prevents a “temporary” debug volume from becoming a semi-permanent pet. Once a volume is reused across runs, you have to account for hidden files, cached tokens, leftover package indexes, and assumptions about file ownership. At that point, the lifecycle is no longer ephemeral, regardless of what the orchestration metadata says.

### **Treat storage as disposable, not durable**

Disposable storage is more than a cleanup convenience. It is part of the isolation boundary. A guest that writes to a clean root volume, a per-run scratch volume, or a transient data disk leaves far less residual state than one that mounts a shared filesystem or a long-lived writable overlay. The latter may be easier to debug, but it weakens the claim that one tool run cannot influence the next.

For short-lived tool invocations, design for explicit export and implicit destruction. Inputs arrive through a scoped channel. Outputs leave through a declared channel. Everything else is destroyed with the guest. If the guest produces an artifact, copy that artifact out in a controlled form, verify it on the host, and then discard the guest storage. Avoid

the temptation to leave a mounted volume around “just in case.” That is usually how scope begins to leak.

Memory needs similar discipline. A teardown sequence should close guest processes, revoke credentials, clear transient network identities, and release the microVM’s RAM back to the host allocator. If the platform claims to scrub memory, verify what that means in your environment. In layered storage systems and shared infrastructures, claims about secure deletion are often weaker than operators assume. Treat host-side destruction as a policy boundary, not as a cryptographic guarantee unless the underlying storage layer actually provides one.

### **Teardown must be as deterministic as launch**

A microVM lifecycle is only ephemeral if teardown is reliable. The guest should not be left hanging because an output exporter stalled, a log collector timed out, or a control-plane component lost track of state. You want an explicit teardown path with bounded time, failure handling, and idempotent cleanup. If the run exits normally, destroy the microVM. If the run crashes, destroy the microVM. If the control plane loses contact, destroy the microVM.

That logic sounds obvious, but it is where many systems become sticky. Operators keep failed guests around for inspection, then reuse them for debugging, then leave them around because a later incident might need the same state. A safer pattern is to move inspection data out of the guest before destruction. Capture logs, exit codes, traces, and declared artifacts on the host. Then tear down the VM immediately. Debugging should work from exported evidence, not from a live sandbox that has already deviated from its intended lifecycle.

A good teardown sequence also rotates any associated identity material. If the guest used a one-shot credential, expire it when the run ends. If the guest had a network identity or a broker token, revoke it. If the run had a unique metadata record or job identifier, mark it closed so that

later automation cannot accidentally attach to a stale instance.

### **Choose the lifecycle model from the latency budget upward**

The right lifecycle strategy depends on the latency budget, the degree of run variance, and the team's tolerance for state complexity. If the workflow can tolerate a modest startup delay and you want the simplest possible security story, cold boot from a minimal image is the safest path. If the workflow has strict synchronous latency needs and the fleet can maintain compatibility discipline, snapshot restore can be a strong option. If jitter matters more than absolute simplicity and the team can enforce rigorous reset semantics, a warm pool may be acceptable.

The decision criteria are straightforward:

|                                 |                                                                      |
|---------------------------------|----------------------------------------------------------------------|
| <b>Startup SLA</b>              | Tighter SLAs push you toward snapshot restore or warm pools.         |
| <b>Residual-state tolerance</b> | Lower tolerance favors cold boot and one-shot teardown.              |
| <b>Fleet homogeneity</b>        | Mixed host CPUs make snapshots harder to operate safely.             |
| <b>Image churn rate</b>         | Frequent image changes reduce the value of long-lived warm state.    |
| <b>Complexity budget</b>        | More lifecycle stages require stronger automation and observability. |

That table hides the most common trap: teams often optimize for the first metric they can measure, usually startup time, and then discover they have created a more fragile sandbox. The safer lifecycle is not the fastest one in isolation; it is the fastest one that still preserves clear teardown, limited state, and auditable provenance.

### **Keep the lifecycle aligned with the threat model**

Ephemeral tool runs are strongest when the guest is treated as a disposable execution cell rather than as an adapted server. That means no broad mutable image, no lingering credentials, no long-lived disks, and no expectation that one invocation should prepare the next. Each run starts from a known base, receives a narrow scope, produces a bounded set of outputs, and disappears.

That model works because it keeps the operational shape of the system aligned with the security model. A hostile input can still compro-

mise the guest, but the guest has little to preserve and little to reuse. A launch broker can still make a mistake, but the mistake is less likely to become durable if the guest is destroyed cleanly. And when a workload needs more speed, you can reach for snapshots or warm pools only after you have the discipline to make their state semantics explicit.

### 6.3. Hardening the Firecracker Execution Model: Jailer, API Surface, and Guest Trust

Firecracker only helps if you keep the boundary as small as the architecture allows. A fast microVM launcher is easy to oversimplify into “a better VM.” That framing misses the point. The useful security property comes from a deliberately narrow execution model: one process per microVM, a small device set, a constrained control plane, and a guest artifact pipeline that you can authenticate and reason about. Once you treat the launch path as a security system instead of a provisioning trick, the important questions become concrete: which process is trusted to create the jail, which socket can change guest state, which artifacts can be booted, and which host services are allowed to influence the guest at all.

```
# Minimal launch shape: a privileged setup step,
# then a narrow API surface during guest startup.

firecracker --api-sock /run/firecracker.sock

curl --unix-socket /run/firecracker.sock \
-X PUT http://localhost/boot-source \
-d '{"kernel_image_path":"/images/vmlinux"}'

curl --unix-socket /run/firecracker.sock \
-X PUT http://localhost/drives/rootfs \
-d '{"drive_id":"rootfs","path_on_host":"/images/rootfs.ext4"}'

curl --unix-socket /run/firecracker.sock \
-X PUT http://localhost/actions \
-d '{"action_type":"InstanceStart"}'
```

That example is intentionally small. The security value is not the commands themselves; it is the fact that the API can be held at arm's length, configured once, and then removed from the steady-state path. If your control plane needs to keep writing to the socket after boot, you have already expanded the trusted computing base beyond what the model was designed to support.

### **Treat the launch path as the security boundary**

Firecracker splits responsibilities in a way that helps you enforce discipline. The control plane decides whether a run is allowed, the jailer constrains the Firecracker process, the VMM exposes a limited API, and the guest kernel runs the tool workload. Those parts are not interchangeable. The control plane should remain policy-oriented and short-lived; the jailer should harden the host-side process environment; the VMM should do as little as possible beyond guest execution and device mediation; and the guest should be assumed untrusted once it starts executing arbitrary tool logic.

That decomposition matters because it changes where mistakes become dangerous. If you expose too much authority in the control plane, the problem is no longer a VM launch problem; it becomes a policy escalation problem. If you leave the VMM reachable after boot, the control plane can mutate a live guest in ways that blur scope. If you trust the guest image too much, you may grant it privileges that only look safe because they are packaged in a signed artifact. The model only stays coherent when each layer has a narrow job and a narrow interface.

### **Use the jailer to reduce ambient privilege, not to redefine trust**

The jailer is often the first component people over-credit. It does important work, but it does not make arbitrary launch inputs safe. The jailer builds a chroot-like confinement context by creating a new mount

namespace, making mounts private, bind-mounting the jail root, pivoting into that root, and then applying chroot as an additional restriction. It can also join a specified network namespace and set cgroup parameters before Firecracker starts. That reduces the amount of privileged post-launch manipulation you need, which is exactly what you want for a short-lived execution service.

The critical point is trust. The jailer's inputs, paths, and resources inside the jail root are assumed trusted. You do not hand unvetted filesystem paths to the jailer and then expect it to sanitize them into safety. The operator invoking the jailer is part of the trusted computing base. That means filesystem ownership, path hygiene, namespace setup, and the contents staged into the jail root are security decisions, not packaging details. If those inputs come from a less-trusted layer, the jailer has already lost the ability to confine them meaningfully.

That assumption changes how you build orchestration. A robust launcher prepares all filesystem objects and namespaces before invoking the jailer, and it does so using a trusted control path. Do not let a partially trusted agent construct jail paths on the fly. Do not let user-provided filenames reach the jail root without policy validation. Do not treat “the jailer will isolate it” as a substitute for verifying what actually enters the jail.

### **Keep the API socket privileged and short-lived**

Firecracker's API is intentionally separated from the VM fast path. That is not a convenience feature; it is a safety feature. Configuration operations happen during launch, while the running guest should spend its time executing the workload rather than servicing administrative mutations. The API socket therefore deserves the same treatment you would give any privileged control surface: strict filesystem permissions, a bounded lifetime, and removal from reach once the guest is running.

The simplest operational rule is to make the socket unavailable to untrusted orchestration layers. If a scheduler, broker, or agent can speak to the socket after the guest starts, it can often alter guest-visible behavior in ways that defeat scope controls. You want launch-time configuration, not open-ended management. If you need continuous post-boot interaction, move that interaction to a much narrower host-mediated channel such as an authenticated broker or a purpose-specific vsock path that carries only the declared run semantics.

That separation also clarifies failure handling. If the socket is only needed during boot, then a socket outage after launch should not compromise guest execution. Conversely, if the socket remains essential during the run, you have coupled the guest's safety to a live host control path, which is exactly the dependency you were trying to avoid by choosing a microVM boundary.

### **Prefer the default seccomp posture**

Firecracker's production guidance recommends the default restrictive seccomp filters and discourages disabling seccomp or replacing it with looser custom filters in production. That recommendation is worth treating as a baseline, not as an optional hardening layer. The point of the model is to reduce the host-facing system call surface of the VMM itself. A custom seccomp policy only helps if you can prove it is at least as tight as the maintained default and still correct across upgrades. Most teams cannot do that reliably.

Seccomp belongs in the same category as the jailer: it reduces ambient capability without changing the trust relationship. The VMM remains in the trusted computing base, but the system call filter makes exploitation harder by narrowing what the process can ask the kernel to do. If you disable that protection because a custom launch path is inconvenient, you are not merely changing a setting; you are widening the host-side attack surface that an attacker can target through the VMM.

## **Treat guest artifacts as potentially hostile**

A signed image is not the same thing as a trusted execution context. Even when your team builds the kernel and root filesystem, the guest is still where untrusted tools, untrusted inputs, and parser-heavy dependencies execute. That means the guest should be handled as a potentially hostile environment. Provenance and verification matter, but they only answer one question: “Did this artifact come from the expected build pipeline?” They do not answer “Will the workload behave safely after boot?”

The practical consequence is that image provenance should be strict and boring. Use immutable identifiers, verify kernel and rootfs artifacts before boot, and refuse to launch assets whose origin is unclear. Then make the image as small and as purpose-specific as possible. A broad, general-purpose rootfs invites runtime customization that erodes the benefit of controlled artifacts. If you need to vary tool dependencies across runs, prefer multiple narrow images over one mutable image that accumulates ad hoc changes.

The guest image pipeline also influences auditability. When a run fails, you want to know exactly which kernel, rootfs, and launch parameters were involved. If the image is mutable or assembled from loosely tracked pieces, you cannot reliably distinguish a platform bug from a poisoned guest. Immutable artifacts and explicit provenance records keep those failure modes separable.

## **Assume metadata-style channels are a privilege boundary**

Cloud systems often normalize metadata services as innocuous bootstrap helpers. In a sandboxed microVM, that habit becomes dangerous. A metadata-style endpoint can expose credentials, network topology, identity tokens, or other host-derived bootstrap data. If the guest can reach such a service without tight scoping, the service becomes a credential broker for whatever code happens to run inside the VM.

The safest stance is to deny metadata-like access by default and reintroduce only the minimum host information the workload demonstrably needs. If the guest needs a run identifier, pass a run identifier. If it needs a short-lived credential, inject a short-lived credential. Do not leave a broad bootstrap API reachable because it is operationally familiar. Familiarity is not a security argument.

This is especially important for agent-driven runs, because the agent may request capabilities that the operator never intended to expose. A metadata channel that looks harmless in a conventional VM becomes a hidden escalation path when the guest workload is selected dynamically by software. The boundary has to be explicit enough that you can explain every piece of host-derived information the guest can observe.

### **Minimize the trusted computing base around the guest**

Firecracker's appeal is that it narrows the machine model, but the gains disappear if you build a sprawling ecosystem around it. The trusted base should include only the components that are required to launch, confine, and monitor the run. That usually means the control plane, the jailer, the VMM process, the verified guest image pipeline, and the specific mediation services that inject inputs or collect outputs. Anything else is a candidate for removal.

A useful test is to ask whether a host-side service needs direct knowledge of guest internals. If it does not, it should probably not be in the critical path. Logging, artifact storage, policy evaluation, and network brokerage can often be isolated from the VMM itself. Every additional dependency that can write to the VMM configuration or reach the guest-control socket increases the chance that a low-privilege mistake becomes a high-impact one.

The one-process-per-microVM model helps here because it simplifies ownership. A process corresponds to a single guest instance, which makes lifecycle accounting, disposal, and per-instance debugging eas-

ier to reason about. That simplicity is valuable when many runs are short-lived and scope-bound. It also means that if the process is compromised, the blast radius should remain limited to that guest instance rather than to a shared launcher handling many tenants at once.

### **Design launch brokers as policy enforcers**

The launch broker is one of the most important parts of the stack because it decides what the guest receives. It should not merely forward requests into Firecracker. It should enforce policy on image selection, memory and CPU limits, network attachment, credential scope, and teardown behavior. If the broker accepts arbitrary parameters from a less-trusted caller and turns them into Firecracker configuration without validation, it becomes a privilege expansion point.

That is why the broker should understand run scopes. It should know whether a run is allowed to reach the network at all, which files may be injected, which outputs may be exported, and how long credentials remain valid. It should also know how to revoke those credentials at teardown. In other words, the broker is part of the security model, not just a provisioning utility.

The easiest mistake is to let orchestration layers accumulate power gradually. A scheduler starts by choosing an image. Then it starts choosing network settings. Then it injects paths. Then it manages credentials. By the time the team notices, the scheduler has become a de facto superuser for the sandbox. Keep those responsibilities explicit and tightly scoped.

### **Use concrete failure modes to guide hardening**

The right hardening choices emerge quickly when you model realistic failures. If a malicious guest escapes its own process, the next barrier should be the VMM and then the host kernel, not a host filesystem mounted broadly into the guest. If the control plane is compromised,

the attacker should not be able to rewrite arbitrary guest state without passing through a policy gate. If the image pipeline is poisoned, the damage should be visible through provenance checks before boot, not hidden inside mutable guest state. If a metadata service is exposed too broadly, the guest should not inherit host credentials by default.

Those scenarios point to the same operational discipline: keep interfaces narrow, keep state explicit, and keep trust boundaries easy to describe. Firecracker gives you the structure for that discipline, but it does not apply it for you. The jailer confines the process, seccomp narrows the syscall path, the API socket exposes a limited configuration surface, and the guest image pipeline defines what runs. The security outcome depends on how carefully you keep each of those pieces in its intended role.

## 6.4. Minimizing Guest/Host Interfaces and Mediating I/O by Scope

The guest kernel is only the first layer of the boundary. Once you adopt a microVM, the remaining host-visible interfaces become the real security decision. If you expose too much device surface, route too many requests through convenience paths, or let guest identity and host identity blur together, the isolation boundary shrinks quickly. The right design is not “a VM with everything turned on.” It is a narrow execution cell with explicit channels for the exact inputs and outputs a tool run needs.

```
# Host-side mediation should keep channels explicit and narrow.
# This example maps a scoped file input and a single control channel.

firecracker --api-sock /run/fc.sock

curl --unix-socket /run/fc.sock \
  -X PUT http://localhost/drives/rootfs \
  -d '{"drive_id":"rootfs",
      "path_on_host":"/runs/42/rootfs.ext4"}'
```

```
curl --unix-socket /run/fc.sock \  
-X PUT http://localhost/actions \  
-d '{"action_type":"InstanceStart"}'  
  
# Host broker forwards only the declared artifact set.
```

That pattern is useful because it makes scope visible. The guest receives a root filesystem, an explicit start command, and whatever additional channels you deliberately open. Everything else is absent by default. Security improves when absence is the default state, because then every extra interface has to justify itself.

### **Start from the smallest useful device set**

Firecracker intentionally keeps its device model small. In practice, that means you should expect to work with VirtIO network, VirtIO block, vsock, serial, and a partial i8042 rather than the large grab bag of devices a general-purpose VMM might expose. That small set is a strength, but only if you resist the temptation to reintroduce broad functionality through convenience services.

The security reason for preferring block devices over filesystem passthrough is straightforward: block I/O is narrower than a full filesystem interface. Filesystems are rich, stateful, and easy to misuse as ambient host access. A block device gives the guest bytes; the guest kernel still interprets those bytes, but the host is no longer exposing a large filesystem protocol surface to the guest. For untrusted tools, that reduction matters. If the tool only needs a root filesystem, scratch storage, or a temporary artifact disk, block devices are the safer default.

The same logic applies to device count. Every device you expose creates parser code, state machines, and edge cases. If the workload does not need serial input, do not leave a writable console connected. If it does not need network access, do not attach a network device. If it does

not need host backchannel communication, do not mount a vsock endpoint just because it is convenient. The device model should reflect the declared scope of the run, not the platform team's preference for generic flexibility.

### **Treat network as a mediated capability, not a baseline entitlement**

Direct guest egress is usually too coarse for a tool runner. A guest with unrestricted network access can exfiltrate data, probe internal services, fetch unexpected payloads, or route around host policy through destinations that the operator never intended to permit. Network access should therefore be tied to scope. If a run needs the network, it should receive the narrowest viable path to the approved destinations, and that path should be mediated by a host-side broker or proxy.

Firecracker's design supports this style because network traffic is host-mediated and can be rate limited. Token-bucket controls on bandwidth and operations per second are not just fairness mechanisms; they are part of the isolation story. They help prevent one guest from consuming disproportionate host resources or turning a sandbox into a denial-of-service source. Rate limiting also forces you to define what the guest is actually allowed to do at the packet level rather than treating network usage as an unbounded side effect of launch.

The important distinction is between policy in the guest and policy at the host boundary. Guest-resident policy is easy to bypass if the guest is compromised. Host-mediated policy is harder to evade because the guest must cross the boundary to reach the network at all. For multi-tenant tool execution, that is the boundary you want to enforce. You can still use in-guest controls for defense in depth, but you should not rely on them as the primary guardrail.

### **Use DNS as part of the security boundary**

DNS often gets treated as plumbing, but it is security-relevant. A guest that can resolve arbitrary names can discover infrastructure, follow redirects, or reach services that the platform never intended to expose. DNS also matters because name resolution can become a covert path around destination controls if the enforcement layer only watches IP addresses and not the names that generated them.

A scoped microVM design should therefore decide whether the guest needs DNS at all. If the run only contacts a fixed set of IPs or host-proxied services, you can remove guest DNS entirely. If the run requires name resolution, give it a resolver that is itself policy-bound, observable, and limited to approved domains or upstreams. This is one of the easiest places for a sandbox to drift back into ambient capability, because DNS feels operationally harmless even when it is not.

You should also be alert to indirect DNS exposure through libraries and package managers inside the guest. A tool may appear to use a fixed endpoint, but a dependency may resolve telemetry, update, or certificate endpoints in the background. If the workload really needs a narrow network scope, inspect the full dependency chain rather than only the top-level command.

### **Treat metadata and bootstrap endpoints as privileged channels**

Cloud-style metadata services create a familiar trap. Engineers are used to treating them as instance conveniences, but inside a sandbox they can become credential delivery paths. If a guest can reach a metadata endpoint, the endpoint may hand it identity tokens, temporary credentials, or bootstrap data that should never have been visible to untrusted code. That is not a minor operational detail; it is a privilege boundary.

The safest default is to omit metadata access entirely. When the guest needs bootstrap information, inject that information directly and

scope it to the run. If the guest needs a short-lived identity, generate one for the invocation and expire it immediately after teardown. If it needs to know which job it is running, pass a run identifier. Do not expose a general-purpose metadata service because it is easier than curating the narrow facts the guest actually needs.

This is especially important for agent-driven tool execution. An autonomous agent may probe for additional capabilities, attempt retries, or follow links that operators never intended to authorize. A metadata channel that is broad enough for convenience is broad enough for abuse. Keep the channel narrow enough that every field is intentional.

### **Design host-side brokers as the enforcement point**

The safest interface model uses host-side brokers for network, files, and credentials. The broker receives a policy decision, stages the per-run inputs, launches the microVM, and mediates export after completion. That gives you a single place to enforce scope. It also lets you separate transient run state from durable platform state, which is exactly what you want when running untrusted tools repeatedly.

For file exchange, the broker should stage inputs in per-run locations and export only declared outputs. For network access, it should enforce destination policy and rate limits. For credentials, it should mint short-lived tokens and revoke them at teardown. That model is more work than attaching broad host resources directly to the guest, but it is far easier to audit and far less likely to create accidental ambient access.

The key design rule is to keep brokers declarative. The guest should not be able to ask for arbitrary extra access once the run begins. If the guest needs a new file, a new destination, or a new identity, that request should be treated as a new policy decision, not as a runtime convenience. That constraint preserves the per-invocation scope that justifies the microVM boundary in the first place.

### **Handle file and artifact exchange as a one-way contract**

File movement is one of the easiest ways to reopen host access accidentally. A shared writable directory, a broad host mount, or a general-purpose sync agent can undo a lot of the benefit of the guest boundary. The safer pattern is a one-way contract: the host injects a specific input set, the guest processes it, and the host collects only declared outputs.

A practical implementation usually looks like this: the host creates a fresh input directory, copies in the exact files the run needs, and mounts or stages that directory into the guest. The guest writes results to a scratch location or an output path that the broker watches. After the run ends, the broker copies out only the expected artifacts and discards the rest. The guest should not be able to see a broader host filesystem, and the host should not reuse the same staging directory across invocations.

This discipline matters because temporary data tends to become persistent data when debugging starts. A reusable “shared scratch” directory often accumulates logs, partial outputs, and forgotten secrets. Once that happens, the directory stops being per-invocation scope and starts becoming hidden state. The sandbox boundary is much easier to defend when every invocation gets a fresh path that disappears at teardown.

### **Constrain console and logging paths**

Small interfaces can still create large problems. Firecracker’s serial path is a good example: if you leave console output unbounded, a guest can generate enough logs to consume host memory or storage. That is not a theoretical edge case. It is the kind of failure mode that appears when a small control channel is treated as harmless because it is not a full filesystem or network device.

You should therefore impose quotas and backpressure on every output

path, including logs. If you want console visibility, cap the retention buffer and decide how much output you actually need for observability. If the guest emits verbose diagnostics, redirect them to a bounded collector or drop them after a defined limit. Logging is useful only when it stays subordinate to the run's resource budget.

The same principle applies to artifact export. A guest that can emit arbitrary amounts of output into a channel with no upper bound can become a resource drain or a data-smuggling path. Scope the channel by type, size, and lifetime. If an artifact is larger or stranger than expected, reject it rather than absorbing it into the host by default.

### **Remove channels rather than monitoring them**

It is common to see an engineering compromise take the form “we will allow the interface but watch it closely.” That is weaker than simply removing the interface when the workload does not need it. Monitoring helps with forensics, but it does not eliminate the attack surface. A guest that can reach a channel can still abuse it faster than a monitoring system can respond.

Use that test repeatedly: if the channel is not essential to the declared run scope, remove it. This applies to network devices, DNS, metadata endpoints, console input, shared filesystems, and backchannel sockets alike. The microVM should not be a full-featured guest operating environment with logging wrapped around it. It should be a narrow service envelope.

That reduction also simplifies troubleshooting. When a failure occurs, a small set of explicit channels is much easier to reason about than a pile of optional ones that may or may not be in play for a given run. Security and operability often align here: fewer interfaces mean fewer places to look and fewer places to fail.

### **Balance observability against exposure**

A tight interface model makes observability harder, so you have to be deliberate. You want enough telemetry to understand launch failures, resource exhaustion, and policy violations, but not so much that you reintroduce broad host access through the observability stack. The usual mistake is to solve debugging by exposing more of the guest. That is convenient in the short term and damaging in the long term.

Prefer host-side observability wherever possible. Capture start and stop events, policy decisions, resource metrics, and declared artifacts at the broker. Keep guest console access constrained and rate-limited. If you need deeper insight into a failure, capture a bounded trace or a short-lived debug artifact rather than leaving an open channel into the sandbox. The goal is to learn enough about the run to operate the platform, not to recreate a general-purpose remote administration path.

That trade-off is easiest to manage when you classify channels by necessity. Control channels, data channels, and debug channels should not all have the same lifetime or trust level. If they do, the guest boundary becomes ambiguous very quickly.

### **Scope drives interface design**

The cleanest way to reason about a microVM sandbox is to start with the declared scope and derive the interface set from it. Ask what the tool must read, where it may connect, what it may write, and which identities it may assume. Then expose only those channels. Everything else stays absent. That rule sounds austere, but it is what keeps the guest boundary meaningful when tool runs are short, repeated, and potentially hostile.

Firecracker makes this style practical because the guest boundary is already small enough to manage. The rest is discipline: narrow devices, brokered I/O, scoped credentials, explicit network policy, bounded logs, and disposable artifact paths. When you keep those channels precise, the microVM remains a sandbox. When you let them drift into

general-purpose convenience, the guest kernel stops being the boundary that matters.

## Chapter 7

# Composing Layers and Building the Tool Runner Control Plane

*A secure tool runner is not just a container profile or a microVM image; it is a control plane that decides, launches, mediates, records, and tears down every run under pressure from latency, scale, and adversarial inputs. This chapter shows how to compose layers without cargo-culting them, and how to build an execution path where every permission, network hop, image, and artifact is attributable, bounded, and policy-driven.*

## 7.1. Choosing Defense-in-Depth Compositions for Mixed-Risk Tool Fleets

A platform that runs only one kind of tool can often get away with one sandbox pattern. A real agent platform cannot. Internal formatting helpers, networked scrapers, tenant-supplied code interpreters, and build-time analyzers present different attacker capabilities, different provenance, and different blast-radius tolerances. If you force all of them through the same execution shape, you usually pay twice: you over-isolate the low-risk tools and waste latency and density, or you under-isolate the dangerous ones and inherit the weakest boundary as your common denominator.

A useful decision framework starts with a simple question: *what failure are you trying to prevent?* A container-only deployment primarily removes ambient host access, constrains filesystem and process scope, and gives you a policy surface that is easy to operate at scale. A microVM removes the host kernel from the workload trust boundary and raises the cost of kernel escape. A nested container-in-microVM design lets you keep container-style packaging and process structure while moving the tenant boundary out to the microVM. Each option removes a different class of risk; none is a universal answer.

**A concrete starting point** A mixed-risk platform usually benefits from explicit execution classes rather than one generic runner. A minimal Kubernetes example looks like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: tool-runner-low
spec:
  containers:
  - name: tool
    image: registry.example/tool:1.8.4
    securityContext:
```

```
        allowPrivilegeEscalation: false
        readOnlyRootFilesystem: true
        capabilities:
          drop: ["ALL"]
---
apiVersion: v1
kind: Pod
metadata:
  name: tool-runner-high
spec:
  runtimeClassName: kata
  containers:
  - name: tool
    image: registry.example/parser:2.1.0
    securityContext:
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: true
      capabilities:
        drop: ["ALL"]
```

That small split encodes the central architectural move. The low-risk tool gets a hardened container profile. The high-risk parser gets the same container interface, but the runtime class shifts the trust boundary to a microVM-backed substrate. The control plane can then choose between profiles by policy instead of by ad hoc operator judgment.

**Decision criteria that actually matter** Selection should follow risk and operability, not preference. The main criteria are straightforward:

- Attacker capability. If the tool processes adversary-controlled bytes, scripts, or archives, assume active escape attempts, not just accidental misuse.
- Trust in provenance. Internal utilities with pinned builds and reviewed dependencies merit a smaller boundary than tenant-uploaded code or arbitrary file parsers.
- Kernel-sharing tolerance. If you cannot tolerate a container es-

cape reaching the host kernel, containers alone are too weak a boundary.

- Density and startup budget. Containers launch fast and pack densely; microVMs consume more memory and take longer to boot.
- Debugging and lifecycle friction. Nested isolation complicates tracing, log access, image refresh, and crash diagnosis.
- Policy drift exposure. The more layers you add, the more chances there are to let one layer silently contradict another.

These criteria rarely align perfectly. That is why the right composition is usually a compromise among residual risk, performance, and operator burden. You are not trying to maximize theoretical hardness everywhere. You are trying to place the strongest boundary where the attacker's leverage is highest.

### **Container-only: the right default for low-risk tools**

Container-only execution works well when the tool is trusted, its dependencies are stable, and the tool does not need to interpret hostile content. Examples include internal data transformers, static format converters, or glue utilities that touch a narrow set of approved inputs. In this pattern, you rely on hardened container settings, restricted capabilities, default-deny filesystem and network policy, and short-lived identity. The kernel remains shared, so the host boundary is still the dominant risk, but the operational cost stays low.

The value of container-only isolation is not that it is invulnerable. The value is that it is simple enough to operate correctly. Once you add a stronger boundary everywhere, the control plane often becomes the weakest part of the system because operators start working around la-

tency, image compatibility, or volume-mount friction. If a tool class does not justify that cost, do not pay it.

**MicroVM-only: strongest tenant boundary with guest-native workloads** MicroVM-only execution is appropriate when you want the workload to own its guest environment and you do not need container-native process packaging inside the sandbox. This pattern fits highly untrusted tools, especially those that must execute arbitrary binaries, inspect user-supplied files, or handle code whose behavior you cannot bound well at the process level. The host kernel is no longer part of the guest trust path, which materially reduces the impact of container escape classes that depend on kernel sharing.

The trade-off is that you now operate two systems: the host control plane and the guest runtime. You inherit guest image management, boot latency, network model complexity, and a different debugging surface. If your workload requires only one process and a few mounted inputs, a guest-native model may be clean. If your workload expects the ergonomics of containers, a pure microVM model often becomes awkward because the orchestration interface no longer matches the developer workflow.

**Nested container-in-microVM: packaging convenience with tenant-grade isolation** Nested execution is the most versatile pattern for mixed fleets. The outer microVM provides the trust boundary that matters for adversarial inputs. The inner container preserves artifact compatibility, layering, and process structure. That combination is valuable when you want to standardize on OCI images and familiar entrypoints while still insulating the host from a hostile tool.

This composition is justified when:

- the workload is high risk enough to warrant a microVM,

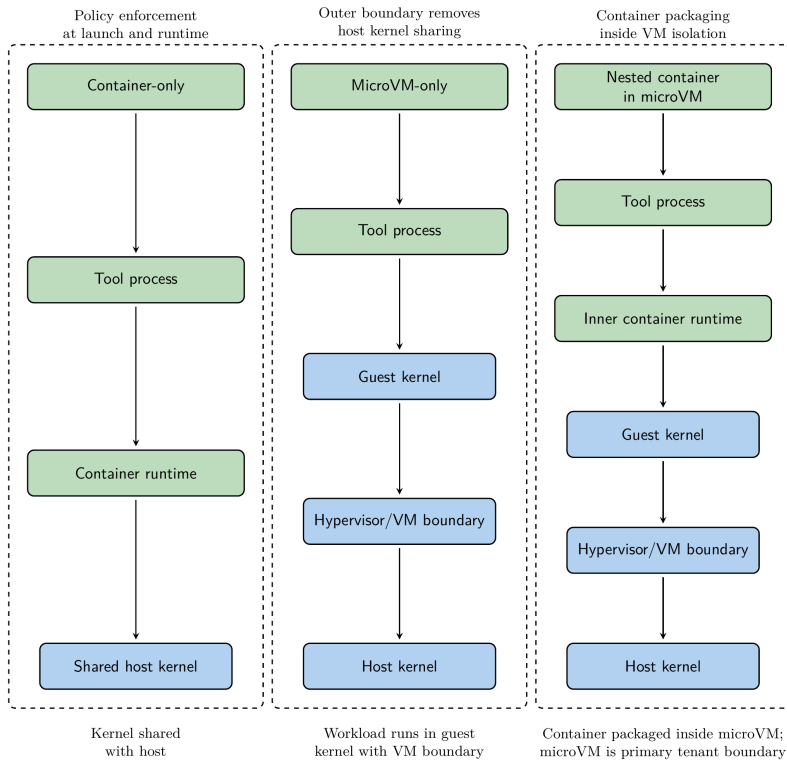
- the team already invests in container images and container-based build pipelines,
- and the tool needs process-level packaging or multiple binaries inside one invocation.

The inner container should not be treated as a second security perimeter unless you can prove that it adds a distinct control point. In many deployments, the inner container mostly improves packaging, dependency management, and reproducibility. The outer microVM is the real tenant boundary. Confusing those roles is a common design error.

**What each boundary actually removes** The three compositions remove different risks:

| Pattern                     | Primary risk removed                                                                                                                           |
|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| Container-only              | Direct ambient access to the host filesystem, PID space, and network namespace; casual privilege escalation through the runtime configuration. |
| MicroVM-only                | Host kernel exposure from the guest; many container escape classes that depend on shared-kernel attacks.                                       |
| Nested container-in-microVM | Both of the above, while preserving image packaging and process isolation inside the guest.                                                    |

The table hides an important subtlety: no pattern removes all risk. Containers do not remove kernel-sharing risk. MicroVMs do not remove guest compromise risk. Nested execution does not remove policy mistakes, credential leakage, or overly broad egress. If you leave a tool with static secrets and open outbound access, you have simply moved the compromise boundary without addressing the exfiltration path.



**A layered reference architecture** The diagram makes the architectural distinction explicit. Container-only execution keeps the shared kernel inside the trust boundary. MicroVM-only execution moves the workload behind a guest kernel and a hypervisor-mediated boundary. Nested execution adds a container runtime inside the guest, but the security boundary still comes from the microVM. That is the point where many designs drift: they treat the inner container as if it were a full security layer, then fail to enforce the outer VM boundary with the same rigor.

**When layering helps and when it does not** Layering helps when the layers defend against different failure classes. A container runtime can give you image structure, filesystem controls, and process framing. A microVM can give you a stronger tenant boundary. Combining them can lower risk if each layer addresses a different attack path. The pattern fails when both layers are configured to do the same job, or when the second layer is only cosmetic.

A weakly configured container inside a microVM is a good example of false defense in depth. If the container has broad outbound access, long-lived credentials, writable mounts, and unreviewed images, the microVM only prevents host escape. It does not stop the tool from exfiltrating data, abusing internal services, or persisting malicious outputs. The design reduced one class of compromise but left the operational impact largely unchanged.

That failure mode is common because teams confuse containment with authorization. A sandbox constrains execution. It does not define whether the execution was worth permitting in the first place. The control plane still needs tool-specific policy, identity scoping, and outbound mediation.

**Mixing runtimes safely on one platform** A mixed fleet needs explicit policy classes. Do not allow low-risk tools to inherit the runtime class, network posture, or credential bundle used by high-risk tools. If you do, the low-assurance tier becomes a policy bypass path.

A practical way to structure the fleet is:

- **Tier 1:** container-only for trusted internal utilities.
- **Tier 2:** container-only with stricter egress and identity for medium-risk tools.
- **Tier 3:** microVM-only for adversarial workloads that do not

need container packaging.

- **Tier 4:** nested container-in-microVM for adversarial workloads that still benefit from OCI packaging or multi-binary launch structure.

The control plane should choose among these tiers using tool metadata, declared risk class, and provenance status. A tool with unknown origin, dynamic dependencies, or user-supplied code should not be silently scheduled onto a lower tier just because the low tier is available. Conversely, a trusted internal formatter should not pay microVM overhead because a default policy is too coarse.

**Node pools, runtime classes, and policy templates** In Kubernetes-backed systems, separating node pools is often cleaner than trying to enforce every boundary with labels alone. High-risk tiers can land on dedicated nodes with the right hypervisor support, storage configuration, and observability hooks. Lower-risk container tiers can remain on denser pools optimized for throughput. `RuntimeClass`, node selectors, taints, and tolerations should be treated as part of the security model, not merely as scheduling hints.

Policy templates need to match the selected tier. A microVM pool should not accept broad `hostPath` mounts, privileged pods, or ambient access to cloud metadata endpoints. A container-only pool should still enforce read-only roots, dropped capabilities, and egress restrictions. The point is consistency: each tier gets a profile that reflects its intended risk envelope. If the policy differs only in name, the fleet will drift until operators normalize the weakest settings across all tiers.

**Compatibility and operational friction** Nested designs introduce the most friction. Startup latency increases because the guest must boot before the containerized tool starts. Debugging requires

visibility into both host and guest logs. Filesystem semantics can become awkward if the inner container expects host-mounted paths that do not map cleanly into the guest. Networking becomes more complicated because the control plane must decide whether the guest owns its own virtual NIC, whether the host proxies traffic, and where policy enforcement sits.

Those costs are acceptable when the tool class justifies them. They are not acceptable when the same tool would run perfectly well in a hardened container. The architecture should therefore remain selective. A platform that uses microVMs for every invocation tends to ossify around its worst-case path. A platform that uses containers for everything eventually treats the strongest boundary as a special exception nobody wants to take. The right balance is a portfolio of tiers with clear admission rules.

**The most common anti-patterns** Three mistakes recur in mixed-risk fleets.

First, teams over-trust packaging. They assume that because a workload arrives as an image, it is isolated enough. Image format is not a boundary.

Second, teams duplicate the same control in multiple layers without tightening the actual exposure. A container inside a microVM with the same broad credentials, same open egress, and same writable shared mounts simply adds complexity.

Third, teams omit the classification model. If every tool request uses the same sandbox profile, operators begin to carve out exceptions for latency, dependencies, or convenience. Those exceptions become the real architecture.

A sound composition strategy avoids all three. It classifies tools by risk,

assigns a runtime tier that matches that risk, and keeps policy, identity, network, and storage assumptions aligned with the chosen boundary. The platform then behaves like a portfolio of containment strategies rather than a single brittle sandbox with a different badge on each workload.

## 7.2. Designing the Tool Runner Control Plane for Policy, Launch, and Mediation

A strong sandbox without a coherent broker still fails in practice. The same platform that hardens containers and microVMs can become inconsistent if one launcher grants credentials before policy, another mutates arguments after approval, and a third emits logs that no longer identify the actual run. The control plane is the part that turns a request into an attributable execution, and it is where you make policy, identity, network mediation, and teardown behave as one system rather than four loosely connected scripts.

### A minimal launch path

A concrete launch flow helps anchor the architecture. A Kubernetes-backed broker can start with admission, then resolve the tool, then issue identity, and only then launch the sandbox:

```
apiVersion: control.example/v1
kind: ToolRun
metadata:
  name: run-7f4c2
spec:
  toolRef: parser-pdf:v2.1.0
  requestor: agent://workflow-44
  input:
    uri: s3://tenant-a/inbox/doc.pdf
  scope:
    network: deny
    secrets: ephemeral
    runtimeClass: kata
```

That request object is intentionally small. It should contain only the facts you need to decide whether the run is admissible and how to materialize it. Everything else belongs in the policy engine, the manifest, or the launch-time environment. If you let the caller smuggle launch parameters directly into the runtime, you lose the ability to explain or reproduce what actually ran.

### **The brokered lifecycle**

A tool run should pass through a fixed sequence of states:

- **Receive the request.** The control plane accepts a canonical request containing agent identity, workflow context, tool reference, declared inputs, and requested scope.
- **Resolve the tool.** The registry maps the tool reference to an immutable artifact, runtime profile, and trust class.
- **Evaluate policy.** The policy engine checks static constraints, contextual rules, tenant boundaries, and capability requirements.
- **Compute effective scope.** The broker intersects the request with defaults and policy to produce the actual run permissions.
- **Assign run identity.** The control plane creates a stable run ID, correlation ID, and per-invocation workload identity.
- **Issue credentials.** Short-lived credentials or tokens are minted only after authorization is final.
- **Select the substrate.** The launcher chooses a container, a microVM, or a nested composition based on the approved profile.
- **Mediate ingress and egress.** Files, network, and stdout or stderr flow through explicit policy points.

- **Record transitions.** The event bus captures state changes, policy decisions, denial events, and key artifact references.
- **Destroy the environment.** The teardown workflow revokes credentials, drains outputs, and removes transient state.

That sequence matters because every later subsystem depends on it. Identity issuance before policy creates a standing credential for an unapproved run. Launch before effective-scope computation creates a race that operators eventually learn to exploit. Teardown after output publication without revocation leaves a credential alive past the useful lifetime of the sandbox.

### Policy decision points and enforcement points

The control plane needs a clear separation between deciding and enforcing. A policy decision point answers *may this run happen, and under what scope?* An enforcement point makes the approved scope real in the launcher, runtime shim, mount namespace, network path, or artifact store.

The decision point should remain centralized and reviewable. That is where you encode canonical rules such as tool trust class, tenant isolation, runtime compatibility, and destination restrictions. The enforcement points should be distributed wherever the request can still be widened in practice: at sandbox launch, on mount construction, in the proxy, in the credential issuer, and in the artifact publication path. If you only enforce once, a later layer can re-open the policy by accident or design.

This split is especially important for mixed-risk fleets. A high-risk tool may be authorized only in a microVM-backed profile with network mediation and ephemeral identity. A low-risk internal helper may be authorized in a hardened container with the same identity model but a much smaller isolation cost. The decision point selects the tier; the

enforcement points ensure the chosen tier cannot be silently widened.

### **What the tool identity must contain**

A tool run needs more than an image tag and a user name. At minimum, you want four distinct identities:

- **Agent identity:** the workflow or agent instance that initiated the request.
- **Human or tenant identity:** the upstream actor or project boundary, when the workflow is user-facing.
- **Tool identity:** the approved tool name, version, and digest.
- **Per-invocation workload identity:** a short-lived identity bound to the specific run.

Keeping these separate avoids the usual audit failure mode where every downstream action is attributed to the same service account. That collapse is convenient until you need to answer which tool touched which secret, which workflow requested which outbound connection, or which tenant's data crossed a boundary. The control plane should mint a stable run identifier and a correlation identifier before any side effects occur, then thread those values through logs, proxy headers, and artifact references.

### **Request canonicalization and effective scope**

Do not authorize the raw request object. Canonicalize it first. Canonicalization means normalizing tool aliases, resolving the version to an immutable artifact, expanding request defaults, and rejecting ambiguous parameters. You want the policy engine to see one precise representation of the run, not a caller-shaped blob that differs by SDK or transport.

Effective scope computation follows canonicalization. The run request proposes a scope; policy overlays defaults, tenant rules, tool constraints, and platform invariants; the result is the effective scope. This step should be monotonic in the safe direction. If a tool asks for broad access and policy narrows it, the narrowed scope must be the only scope that reaches the launcher. Late mutation breaks this guarantee. Once a manifest is signed or stored, you should treat it as immutable for the life of the run.

A practical way to think about this is as a set intersection:

$$S_{\text{effective}} = S_{\text{request}} \cap S_{\text{tool}} \cap S_{\text{tenant}} \cap S_{\text{platform}}.$$

The actual policy may be more expressive than set intersection, but the mental model is useful: the broker only removes permissions unless an explicit, reviewable override exists.

### **Identity issuance and secret delivery**

Issue credentials only after policy has produced a final effective scope. The identity issuer should accept the stable run identity and produce short-lived credentials that are valid only for the intended audience. This is where per-invocation workload identity becomes operationally useful: the sandbox can authenticate downstream as itself, not as the agent host or the control plane.

Kubernetes-backed systems often use projected service account tokens or similar short-lived material; service-mesh and federated identity systems use the same principle with different delivery paths. SPIRE registration provides a clean model: selectors identify the workload, the registration entry defines what the workload may receive, and the workload API delivers the material in a standard way. The important control-plane property is not the specific protocol. It is the fact that the credential is minted for one run, not for the entire agent fleet.

A common implementation pattern is a mutating admission webhook

or a launcher-side injector that adds a credential volume or sidecar only after authorization. That pattern works, but only if the mutation itself is constrained by the already-approved manifest. If the mutator can add broader access than the policy engine allowed, you have merely moved the trust boundary into the webhook.

### **Launchers, shims, and substrate selection**

The launcher is an enforcement point, not a policy engine. Its job is to translate the approved manifest into a runnable environment, then refuse anything that does not match the manifest exactly. If the launcher resolves a different image, widens a mount, or starts with a different runtime class than the manifest specified, it has bypassed the broker.

This is where the earlier composition choices become concrete. The policy engine may route a low-risk tool to a hardened container, a high-risk parser to a microVM, or a packaging-heavy workload to a nested container-in-microVM profile. The launcher must preserve that selection. A low-risk tool should not silently inherit a microVM tier because a fallback path was more convenient. A high-risk tool should not fall back to a shared-kernel container because the stronger runtime is busy.

A robust launcher also records the substrate actually used. That record matters because the selected tier is part of the audit trail and part of the reproducibility story. If you cannot say which runtime class executed a run, you cannot tell whether a failure came from the tool, the policy, or the sandbox shape.

### **The lifecycle boundary diagram**

The diagram highlights the architectural rule that keeps the system honest: the decision and enforcement responsibilities are separated, but every step is chained by a stable run identity. The policy engine does not launch anything. The launcher does not redefine policy. The

mediation layer does not invent outputs. Each component has a narrow job and a clearly bounded trust relationship.

### **Mediated ingress and egress**

Ingress mediation covers files, streams, and any remote object the tool needs to consume. Egress mediation covers stdout, stderr, return values, network calls, and any output artifact that leaves the sandbox. You should not treat these as passive plumbing. They are part of the policy surface.

A useful discipline is to classify channels explicitly. Some channels carry approved input, some carry derived output, and some carry only operational telemetry. A run may be permitted to read one object and write another, but not to reach arbitrary storage or arbitrary hosts. The mediation layer should enforce that distinction at the boundary, not rely on the tool to self-limit. If the tool can see the destination directly, the policy only exists on paper.

For networked runs, the mediation layer should attach the run identity to downstream requests so proxies and gateways can log and authorize using the invocation principal rather than the node identity. For file-oriented runs, the same layer should hash or quarantine outputs before publication. That prevents the common leakage path in which a sandboxed tool writes sensitive data to a temporary directory that is later swept into an artifact bucket.

### **Why launch-time mutation must stay constrained**

Secret injectors and mutating webhooks are useful, but they can also become policy shortcuts. A mutator that adds a credential volume, a proxy sidecar, or a trust bundle must receive the already-approved manifest and must produce a diff that can be audited. If it silently broadens permissions, your admission layer has stopped being a control plane and started being a suggestion engine.

Kubernetes admission is especially relevant here. Built-in mechanisms such as `ValidatingAdmissionPolicy` give you in-process validation for invariants that do not need a remote webhook. External webhooks, in contrast, are appropriate when you need mutation, supply-chain verification, or richer contextual checks. Use the built-in path for the stable invariants: no privileged pods, no host networking unless justified, and no broad volume types unless the tool profile allows them. Reserve mutation for the cases where the control plane truly needs to materialize a derived object, such as a projected token volume or a proxy configuration.

### **Teardown, revocation, and artifact publication**

Teardown should revoke credentials before the sandbox disappears, not after. If a token remains valid while the environment is still reachable, a compromised run can continue using its identity during shutdown. The teardown workflow should therefore do three things in order: revoke or expire credentials, drain or seal outputs, and destroy the runtime instance.

Artifact publication is part of teardown, not an unrelated storage action. Only approved outputs should leave the sandbox, and they should do so with stable references to the run ID, policy decision, and artifact digest. If you store a raw result without those references, you lose traceability. If you export everything by default, you lose containment. The export step should be narrow, explicit, and idempotent.

### **Failure modes that matter operationally**

Three failures recur in brokered systems.

First, policy lives in the launcher. The launcher becomes a hidden policy engine, and nobody can review or test the actual authorization logic independently. That is fragile and hard to audit.

Second, parameters mutate after authorization. A run is approved un-

der one scope, then a sidecar, mount, or environment variable changes before launch. The final environment no longer matches the decision record.

Third, sandboxes are reused across invocations. Reuse seems efficient until residual files, open sockets, cached tokens, or process state leak across runs. For a tool runner, ephemeral execution is usually the safer default because cleanup is cheaper than proving isolation between unrelated invocations.

A subtler failure is coupling audit to agent session identifiers. Sessions are ephemeral and often change for reasons unrelated to security. Correlation should instead rest on the stable run ID and the manifest digest. That gives you a reproducible unit of work even when the caller connection drops, reconnects, or retries.

### **Version-sensitive interfaces and compatibility**

The control plane sits at the junction of APIs that evolve at different speeds. Kubernetes admission semantics, runtime shims, identity providers, and proxy sidecars can all change independently. That means you need compatibility checks at the boundaries: does the launcher understand the manifest version, does the identity issuer support the requested audience and TTL, does the runtime class exist on the selected node pool, and does the proxy understand the mediation headers you plan to log?

Kubernetes Pod Security Admission is now a stable baseline for namespace-level hardening, and `ValidatingAdmissionPolicy` gives you an in-process path for structural validation. Those features reduce reliance on remote webhooks for common invariants, which in turn lowers failure coupling in the control plane. For richer workflows such as secret injection or supply-chain verification, external controllers still make sense, but they should not own the core lifecycle of a run.

## What success looks like

A mature tool runner control plane gives you a reproducible answer to four questions for every invocation: what was requested, what was allowed, what actually ran, and what left the sandbox. The request is canonicalized, the policy decision is recorded, the manifest is immutable, and the enforcement points only realize that manifest rather than reinterpret it. Identity is per invocation, credentials are short-lived, output is mediated, and teardown closes the loop without leaving a live path behind. When those properties hold, the broker stops being a wrapper around execution and becomes the execution contract itself.

### 7.3. Implementing Default-Deny Egress with Proxies, DNS Policy, and Metadata Shields

A tool run that can read but cannot leave the sandbox is far easier to reason about than one that inherits the node's ambient network. The dangerous failure mode is subtle: you grant a parser or scraper only file access, then discover later that the sandbox could still resolve arbitrary names, reach package mirrors, or pull cloud credentials from a link-local metadata service. Once outbound traffic is implicit, every other permission becomes weaker because exfiltration and lateral movement remain available by default.

#### A concrete egress posture

A useful baseline is to make every outbound path explicit at launch time. In Kubernetes, that means default-deny network policy, DNS mediation through a controlled resolver or proxy path, and direct blocking of metadata endpoints:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-egress
```

```
    namespace: tool-runs
spec:
  podSelector: {}
  policyTypes: ["Egress"]
  egress: []
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns
  namespace: tool-runs
spec:
  podSelector:
    matchLabels:
      app: tool-runner
  policyTypes: ["Egress"]
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          kube-system: "true"
    ports:
    - protocol: UDP
      port: 53
    - protocol: TCP
      port: 53
```

That policy is not sufficient by itself. It blocks ambient egress at L3 and L4, but it does not express hostname policy, and it does not stop a workload from using DNS as a discovery channel before a later connection attempt. You still need a mediation layer that understands destinations as names or service identities, not only as IPs and ports.

### The layered shape of default-deny egress

Treat outbound control as three separate enforcement layers:

- **L3/L4 deny by default.** The sandbox gets no ambient egress unless policy grants it.
- **DNS mediation.** Name resolution occurs only through an approved resolver path, and the answers become part of the security decision.

- **Metadata shielding.** Link-local and host-local identity sources stay unreachable unless you explicitly mediate them.

That separation matters because a destination policy written only in terms of domains still depends on DNS behavior, and a proxy policy written only in terms of HTTP requests still depends on the ability of the sandbox to resolve names. The control plane should assume these dependencies are part of the attack surface. If DNS can drift around policy, the proxy becomes advisory. If the metadata service stays reachable, a tool can often recover credentials even when outbound Internet access is otherwise blocked.

### **Default-deny at the network layer**

Kubernetes NetworkPolicy gives you the first gate. When no egress policy exists, traffic is generally allowed; once you select all pods in a namespace with an empty egress rule set, you create a default-deny posture. That is a strong primitive for bounding IP, port, and protocol access, but it is intentionally low-level. It does not know about host-names, methods, or tenant-specific service identity.

Use this layer to make the absence of an allow rule meaningful. That means you should not add broad exceptions such as “allow all cluster DNS” or “allow all east-west traffic” unless the workload truly needs them. Every broad exception becomes a hidden dependency that later policies must work around. For a tool runner, the desired shape is usually narrow: allow only the resolver path, the mediation proxy path, and any explicitly justified internal service endpoints.

Where the runtime is a microVM rather than a container, the same principle applies. The guest network model may differ, but the policy intent does not: no permitted destination means no packet leaves. The control plane should not let the runtime default back to the host’s network posture just because the workload happens to boot in a different

substrate.

### **Why hostname policy needs DNS-aware mediation**

IP allowlists alone are brittle in systems where destination IPs change, load balancers front multiple services, or content-addressed infrastructure moves quickly. Hostname-only allowlists are also brittle if you do not control the resolver path. A sandbox that can send queries to an arbitrary resolver can learn disallowed IPs, bypass your view of the name-to-address mapping, and then connect directly.

That is why DNS-aware policy is more than a convenience feature. It makes resolution part of the security boundary. In a Cilium-style model, you can authorize egress by DNS name, and the policy engine learns the resulting IPs from approved DNS responses. This works because the policy decision is no longer detached from the transport path. If the DNS answer is not observed by the policy layer, the connection cannot be safely attributed to the allowed name.

The operational implication is simple: you must mediate both resolution and connection. If you allow direct DNS while assuming the proxy governs destinations, the sandbox can bypass name-based controls through alternate resolvers, cached answers, or hard-coded IPs. If you allow direct IP egress while assuming DNS controls the destination, the sandbox can skip the name entirely. Aligning resolver path, policy evaluation, and outbound connection enforcement keeps those escape hatches closed.

### **Proxy-mediated egress as the decision point**

The proxy or gateway should act as the last explicit authorization gate before a packet leaves the platform. It receives the per-run identity, the effective scope, and the resolved destination class, then decides whether the request matches the approved policy. That request may be an HTTP call, a TLS tunnel, a package download, or a generic TCP

session if the protocol stack supports it. The important property is that the proxy is policy-aware; it does not merely forward.

A clean implementation binds the proxy decision to the run identity and correlation ID introduced by the brokered control plane. That lets you log and revoke access per invocation rather than per node. It also prevents a workload from inheriting broader authority just because the host or node has an established identity. For multi-tenant agent platforms, that distinction is critical. A node principal is too coarse to represent a single tool invocation.

A practical example is destination classification by service identity rather than by raw address. You can permit access to a named internal API, a package mirror, or a tenant-approved external domain, then resolve that policy through the proxy at request time. The proxy records the run ID, the destination name, the method or protocol class when relevant, and the policy decision that allowed the request. That record becomes the accountability trail for every permitted byte.

### **Destination expression and policy shape**

A destination policy should be rich enough to express what you mean without becoming so specific that routine infrastructure changes break runs. The useful dimensions are:

- **Service identity or hostname pattern**
- **Protocol and port**
- **HTTP method or request class, when applicable**
- **Optional path class or API family**
- **Per-run identity and tenant boundary**

You want the least specific rule that still captures intent. If you overfit to raw IP addresses, a routine backend rotation causes outages. If you

overfit to broad domain wildcards, you create an authorization model that is too loose to defend. The control plane should therefore prefer semantic destinations wherever possible and then constrain transport behavior underneath them.

That policy shape also helps with revocation. If a downstream service begins to misbehave, you can revoke the named destination for the affected run class without rewriting launch logic. If a tenant's allowance changes, you update one policy source rather than a collection of node-specific firewall exceptions.

### **Metadata endpoints are special**

Cloud metadata services are not ordinary destinations. They are identity and configuration surfaces that often grant access far beyond what the tool was originally supposed to see. Blocking them is a baseline control, not an optional hardening tweak. Even when a provider uses session-oriented metadata access such as token-based flows, the endpoint remains sensitive because it can still be abused for credential theft or SSRF-style pivots.

The correct default is to block metadata endpoints outright for untrusted tool workloads. Treat `169.254.169.254` and similar link-local services as ambient trust sources that should never be reachable by accident. If a specific workload genuinely needs metadata access, mediate it explicitly and narrowly. Do not assume that IMDSv2-like protections make the endpoint safe for general use. They improve access semantics, but they do not remove the core problem that metadata often sits one hop away from the sandbox with high-value credentials behind it.

In mixed container and microVM fleets, this policy should remain consistent. The fact that one runtime uses a guest kernel and another uses a shared kernel does not change the exfiltration risk posed by metadata access. The control plane should encode the block at the network pol-

icy level, reinforce it at the proxy, and verify that the runtime does not expose a bypass through host-local routing or guest-side defaults.

### A minimal mediation pattern

A useful pattern is to combine three elements: default-deny egress, a policy-aware proxy, and explicit metadata blocking. The following shape is enough for many tool-runner deployments:

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: tool-egress
  namespace: tool-runs
spec:
  endpointSelector:
    matchLabels:
      app: tool-runner
  egress:
    - toFQDNs:
        - matchName: api.example.com
        - matchPattern: "*.pkg.example.com"
    - toEntities:
        - kube-apiserver
    - toCIDRSet:
        - cidr: 169.254.169.254/32
          except: []
    enabled: false
```

That snippet illustrates the policy intent rather than a universal deployment recipe. The exact syntax depends on your CNI and proxy stack, but the design pattern is stable: allow names, not ambient networks; mediate DNS, not just sockets; and make the metadata endpoint unreachable unless a specific policy exists.

### How the pieces interact at runtime

At launch, the broker assigns the run to a network posture class. The launcher then injects only the routes, resolver configuration, and proxy settings that match that class. During execution, the sandbox can only reach the resolver path and the mediation path unless an explicit al-

low rule grants more. If the tool calls a permitted domain, the proxy authenticates the request using the run identity and logs the decision. If the tool attempts to resolve or connect elsewhere, the denial should occur as close to the request as possible, ideally before the destination becomes observable outside the sandbox.

This sequence lets you preserve attribution. A denied outbound call is not just a packet drop; it is a policy event attached to a particular run. That attribution is what makes later incident response useful. You can tell whether the tool attempted a forbidden destination, whether the DNS path was the problem, whether the proxy denied the request, or whether the runtime never had the route in the first place.

### **DNS drift, rebinding, and cache behavior**

DNS-aware policy only works if you understand resolution drift. A hostname can map to multiple IPs over time, and the answer observed at policy time may not be the answer used later by the workload. That is why the resolver path and the proxy path must stay coupled. The proxy should observe the same name-to-address relationship that the sandbox used when making the request.

You should also account for DNS rebinding and cache effects. A malicious service can return one address to satisfy policy evaluation and another address later to redirect traffic. A sandbox can also cache answers locally and continue to use them after policy changes. To reduce this risk, keep the resolver path narrow, keep the cache lifetime short, and ensure the proxy or gateway validates the destination against the currently authorized name set before forwarding the connection.

The practical rule is that DNS policy is not a substitute for network policy. It is a second layer that makes network policy meaningful for named destinations. If you drop either layer, you weaken the other.

### **Consistent enforcement across containers and microVMs**

The main architectural challenge is avoiding policy gaps between runtime classes. A container runner might rely on CNI enforcement and sidecar proxying. A microVM runner might rely on guest networking with host-side mediation. A nested runner may use both. If you let each runtime class define its own egress semantics, the fleet will eventually drift, and the weakest path will become the default path.

Instead, define one policy model and project it into multiple enforcement mechanisms. The control plane decides whether a run can access DNS at all, which destinations it may resolve, which proxies it may use, and whether metadata is blocked. The runtime-specific adapter then materializes those choices using the local mechanism: network policy for containers, guest firewalling or virtio-NIC mediation for microVMs, and proxy configuration for both.

That projection step is where consistency lives. The broker should emit the same effective scope regardless of substrate. The substrate-specific code should only translate policy into the right boundary controls. If the container path allows DNS via a sidecar but the microVM path bypasses the sidecar, you have two security models. The correct design is one model, two adapters.

### **Operational trade-offs**

Default-deny egress adds latency, configuration complexity, and support cost. DNS-aware policy can slow first-request resolution. Proxies can introduce a failure point if they are unavailable. Metadata blocking can break tools that silently depended on cloud identity or instance-local configuration. Those costs are real, and you should expect them during rollout.

The trade-off is that they make outbound access legible. A tool that truly needs a package mirror, a web API, or a tenant-specific storage endpoint gets an explicit permission and a traceable path. A tool that does not need egress gets none. That distinction is much cheaper to

operate than debugging a breach in which every sandbox had network access because the platform considered connectivity a convenience rather than a privilege.

The cleanest implementation is to fail closed until the mediation path is ready, then open only the specific destinations required by the approved run. Avoid temporary broad allow rules during bootstrap. Once those exceptions exist, they tend to survive long after the original need has disappeared.

### **Edge cases and compatibility notes**

Some workloads need direct DNS for bootstrapping, but that should be rare. If you permit it, keep the resolver and the proxy under the same policy domain so destination identity does not split across two independently trusted paths. Some workloads also need access to internal service discovery or package mirrors; express those as named destinations, not as broad cluster-wide exceptions.

Be explicit about the runtime model as well. CNI behavior, host networking, sidecar interception, and guest networking do not align perfectly across orchestrators and microVM stacks. A policy that looks identical in YAML may produce different enforcement if the runtime adapter is not equally strict. Test the failure cases, not only the happy path: denied DNS, denied metadata, failed proxy startup, and destination drift after policy change.

The safest operational stance is to treat outbound traffic as part of the tool's permission set, not as a side effect of being scheduled. Once you do that, the absence of an allow rule has a clear meaning, and every permitted connection becomes an intentional part of the run rather than an accident of the node it happened to land on.

## 7.4. Issuing Ephemeral Workload Identity and Delivering Secrets Per Invocation

Shared credentials are the fastest way to lose the boundary you just built. If every tool in the fleet reads from the same API key bundle or the same long-lived service account, then attribution collapses, revocation becomes coarse, and the control plane can no longer tell which invocation used which capability. The right model is narrower: issue identity after policy has finalized the run, bind it to that run only, and deliver secrets through a channel whose lifetime and audience match the sandbox.

### A practical launch-time identity exchange

A short-lived, invocation-scoped token is the simplest place to start. In Kubernetes, projected service account tokens give you the primitive you want: explicit audience, explicit expiration, and automatic rotation before expiry. A minimal pod fragment looks like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: tool-runner
spec:
  serviceAccountName: run-sa
  containers:
  - name: tool
    image: registry.example/tool:2.4.1
    volumeMounts:
    - name: run-token
      mountPath: /var/run/token
      readOnly: true
  volumes:
  - name: run-token
    projected:
      sources:
      - serviceAccountToken:
          path: token
          audience: api.example.com
          expirationSeconds: 600
```

That pattern does two important things. First, it makes the credential temporary enough that the lifetime of the token follows the lifetime of the invocation rather than the lifetime of the node. Second, it gives downstream services a concrete audience to validate, which prevents the token from becoming a generic bearer artifact. If the workload needs a different downstream service, issue a different token for that audience rather than broadening one token until it becomes a standing secret.

### **The identity model you actually need**

A tool run should not inherit identity by accident. Define the identity structure explicitly:

- *Initiator identity*: the human, API client, or upstream workflow that requested the run.
- *Agent identity*: the orchestration principal that submitted the request.
- *Tool-run identity*: the per-invocation workload principal attached to one run only.
- *Audience*: the downstream service or policy domain the credential may present to.
- *Lifetime*: the expiration horizon for the credential.
- *Revocation hook*: the online check that can invalidate the credential before expiration.

The separation matters because downstream systems usually log only what they see. If a database, proxy, or object store only sees the agent identity, then every action appears to have come from the same broad

platform account. If it sees the tool-run identity, the audit trail becomes per invocation. That difference is operationally valuable even when the security policy itself has not changed.

### **Why projected tokens are the default**

Kubernetes moved away from legacy secret-backed service-account tokens because static token Secrets do not match the lifecycle of ephemeral workloads. A projected token is better suited to a tool runner because it expires automatically and kubelet rotates it before expiration. You can also set the audience explicitly, which keeps the credential scoped to one downstream trust domain instead of turning it into a general-purpose bearer token.

For revocation-sensitive systems, the key limitation of offline validation is that a token can be structurally valid even after the pod or bound object has been deleted. That is why online validation through Token-Review is often the safer choice when immediate revocation matters. Offline verification is useful when the downstream service can tolerate a short revocation window and wants to avoid the extra round-trip. It is not sufficient when the token itself is part of the containment boundary.

The practical rule is straightforward: use projected, audience-scoped, short-lived tokens by default; use online validation when deletion or deprovisioning must take effect quickly; and avoid static token Secrets for invocations that should disappear cleanly.

### **SPIFFE and SPIRE as a workload identity shape**

SPIFFE gives you a useful conceptual model even if you do not deploy the whole stack. A SPIFFE ID names a workload identity, and an SVID packages that identity as either X.509 or JWT material. The SPIFFE Workload API then delivers identity and trust bundles without embedding secrets in the image or manifest. That is exactly the shape a tool

runner wants: identity appears only after policy admits the run, and the workload receives only the material it needs to authenticate itself downstream.

X.509-SVIDs are usually the better default when you can use them, because they fit mTLS and are less replay-prone than JWT-SVIDs. JWT-SVIDs still have a place when a downstream system expects a signed token rather than a certificate chain, but they move more trust into bearer semantics. For a tool runner, the preference should usually be certificate-backed workload identity wherever the downstream protocol allows it.

SPIRE registration also fits the control-plane pattern well. You can match selectors to the workload that the launcher is about to start, then issue the identity only to that workload. That avoids a common mistake: preloading identity material into the image or into a shared secret volume long before the policy engine has approved the invocation.

### **Delivery channels and their exposure surfaces**

Three broad delivery mechanisms cover most real deployments:

- *Environment injection.* Easy to wire up, but high risk for accidental logging, process inheritance, and shell expansion leaks.
- *File or volume materialization.* Better than environment variables because you can constrain file permissions and lifetime.
- *Brokered retrieval.* A sidecar or identity service fetches secrets on demand, which reduces static exposure at launch time.

Environment variables are attractive only because they are simple. They are visible to the process and any child processes, they can leak into crash dumps or debug output, and they persist in ways that are hard to audit once exported. Use them only for low-value configura-

tion, not for credentials that would materially widen the blast radius if copied.

File- or volume-based delivery is usually the minimum acceptable pattern. It still creates a local secret, but you can bind it to a short-lived mount and remove it during teardown. It also works across container and microVM substrates because the file becomes part of the invocation environment rather than part of the image. If you need stronger control, mount the credential read-only and keep the path ephemeral.

Brokered retrieval is strongest when you want the secret to exist only after an internal authorization check and perhaps only for the duration of one request. That pattern is more complex, but it gives you better revocation leverage and less persistent material inside the sandbox. The trade-off is latency and operational complexity: the run now depends on an extra service or sidecar path that itself must be available, authenticated, and monitored.

### Choosing between delivery mechanisms

A useful decision matrix is not about ideology; it is about exposure and revocation.

| Method                         | Best use                                                                                         |
|--------------------------------|--------------------------------------------------------------------------------------------------|
| Environment injection          | Low-value settings, short-lived non-secret configuration, or legacy compatibility.               |
| File or volume materialization | Default choice for ephemeral credentials that must survive process startup but not run teardown. |
| Brokered retrieval             | High-value secrets, revocation-sensitive access, or workflows that need just-in-time issuance.   |

If the secret unlocks production data, object stores, signing keys, or cloud APIs, you should bias toward file-based or brokered delivery and away from environment variables. If the secret is only a limited-scope token for a single read-only service, the operational overhead of a broker may not be justified. The important point is that the control plane makes the choice, not the tool author.

## **Lifecycle ordering and issuance timing**

Credential delivery must happen after the effective scope is final. That means you first resolve the tool, compute the scope, select the runtime, and record the immutable manifest. Only then do you mint the token or materialize the secret. If you issue first and authorize later, you create a window in which an unapproved or partially approved run holds credentials that it should not have seen.

The control plane should also bind issuance metadata to the run record. Record the credential audience, TTL, issuer, and correlation ID. If a downstream service later needs to prove why it accepted a request, you want the audit trail to point back to the exact invocation and the exact policy decision that allowed the credential to exist.

This is also where secret delivery and launch-time mutation intersect. A mutating admission webhook or a launcher-side injector can add the secret volume or sidecar, but only as a consequence of an already-reviewed manifest. The mutation should be deterministic and reversible: if the manifest changes, the injected secret path must change with it, and if the policy denies the run, no secret object should be created.

## **Revocation is not the same as disappearance**

A token can expire and still leave traces. A connection can remain open after the token is invalidated. A downstream service can cache data. A copy of a secret can live in process memory, a file cache, or a log buffer. You therefore need to treat revocation as a boundary condition, not as magic erasure.

That is why online validation matters in revocation-sensitive systems. If a pod is deleted, a bound object disappears, or the control plane tears down a run, a downstream service that only validates signatures offline may continue to trust a token longer than you intended. TokenReview

closes part of that gap by asking the control plane whether the token still represents a live workload. The cost is an extra validation path; the benefit is that the token can die when the run dies.

The practical design question is how much revocation latency you can tolerate. For interactive tools with short-lived calls, you can often rely on expiration and periodic rotation. For tools that hold high-value credentials or perform long-lived automation, you should prefer online checks and short TTLs. The more sensitive the access, the less you should depend on eventual expiry alone.

### **Making downstream audit follow the workload**

Downstream systems should log the workload principal, not the agent host. That requires two things. First, the identity delivered to the sandbox must be stable enough for downstream correlation during the run. Second, the downstream service must record the received workload identity as part of its own access logs or request context.

This is where per-invocation identity pays for itself. If a scraper reads a protected endpoint, the endpoint should be able to say that invocation `run-7f4c2` accessed it under `audience api.example.com` with TTL 600 seconds. If the same host later runs a different tool, that is a distinct identity, a distinct scope, and a distinct audit line. You do not need to infer which process happened to occupy the node at the time.

The same principle applies to sidecar-mediated delivery. If the sidecar fetches a secret from a broker, the broker should issue the secret to the run identity, not to the sidecar as a generic helper. Otherwise the sidecar becomes the real principal and you lose attribution at the point where you most need it.

### **A launch path that keeps identity local**

A practical pattern uses the broker to create a run record, then the launcher requests credentials only for that record. A simplified se-

quence looks like this:

```
POST /runs
{
  "toolRef": "parser-pdf:v2.1.0",
  "audience": "api.example.com",
  "ttlSeconds": 600,
  "identityMode": "projected-token"
}
```

The broker returns a run ID and the approved issuance parameters. The launcher then requests the actual credential from the issuer, mounts it into the sandbox, and records the resulting artifact hash or token reference in the run manifest. That keeps the credential path local to the invocation and prevents the launcher from deciding policy on its own.

### **Compatibility constraints across containers and microVMs**

Containers and microVMs differ in how they expose the secret, but not in what the secret means. A container may consume a projected volume or sidecar file, while a microVM may receive the same material through a guest agent, a mounted virtio-fs path, or a guest-side identity service. The control plane should hide those differences behind a common issuance contract.

The main compatibility risk is persistence. Containers often make it easy to mount a short-lived file and remove it at teardown. MicroVMs may need guest coordination to ensure the same file disappears when the run ends. If you do not handle teardown carefully, the identity survives longer than the process that was supposed to own it. That is a lifecycle bug, not a runtime bug, and the control plane owns it.

Another compatibility concern is token audience support. Some downstream services expect JWT-style bearer tokens, while others expect mTLS certificates or OIDC-style assertions. Do not coerce every service into one format if the protocol naturally supports a stronger one.

Use the format that fits the downstream trust model, then keep the control-plane semantics consistent across formats.

### **Common mistakes**

Three mistakes recur in real systems.

First, teams bake credentials into images because it is convenient at build time. That turns every copy of the image into a latent secret leak.

Second, teams mount broad secret bundles “just in case.” The result is unnecessary exposure, unclear audit scope, and a wider blast radius if the run is compromised.

Third, teams rely on environment variables for high-value long-lived secrets and then assume process isolation will save them. It will not. Children inherit them, debug tooling can print them, and application code can accidentally serialize them.

If you avoid those patterns, the control plane can keep identity aligned with the invocation itself. The run gets what it needs, for as long as it needs it, and downstream systems can attribute every action to the tool that actually performed it rather than to the agent that happened to launch it.

## **7.5. Operating Audit and Forensics Pipelines Without Turning Logs into a Data Leak**

A tool platform that cannot explain what happened during an incident is already difficult to trust. A platform that explains everything by logging prompts, environment variables, and raw artifacts is worse: it turns observability into a second exfiltration channel. The design goal is narrow but strict. Capture enough to reconstruct policy, identity, substrate choice, and mediation outcomes for every run, while with-

holding the sensitive payloads that a sandbox exists to contain.

**A practical audit policy** A useful starting point is a metadata-first audit policy that records the control-plane facts without storing request or response bodies by default:

```
apiVersion: audit.k8s.io/v1
kind: Policy
omitStages:
- RequestReceived
rules:
- level: Metadata
  resources:
  - group: ""
    resources: ["pods", "pods/log"]
    omitManagedFields: true
- level: RequestResponse
  verbs: ["create"]
  resources:
  - group: "control.example"
    resources: ["toolruns"]
  namespaces: ["tool-runs"]
```

That example encodes the main discipline. Use metadata-level logging for routine Kubernetes activity, reserve request-response capture for tightly bounded control-plane resources, and omit managed fields to reduce both noise and accidental exposure. The point is not to maximize visibility everywhere; the point is to preserve the minimal evidence needed to answer a security question later.

**The telemetry contract** A tool run should produce a small, fixed set of audit events. You do not need every object body to reconstruct the run. You need stable identifiers and state transitions:

- request admission outcome,
- effective-scope reference or manifest digest,
- tool identity and image or snapshot digest,

- run ID and correlation ID,
- workload identity issuance event,
- sandbox launch success or failure,
- egress authorization decisions,
- denied outbound attempts,
- major resource breaches,
- teardown reason,
- approved output references.

Those events are enough to tell you what policy was in force, what substrate executed the work, what identity the run used, what external interactions were allowed, and which artifacts escaped the sandbox. Anything beyond that should be justified by a specific investigative need. If you capture raw prompts or full environment blocks by default, you make the telemetry system itself a data leak.

**Why metadata-level audit is the default** Kubernetes audit events can fire at multiple stages, including `RequestReceived`, `ResponseStarted`, `ResponseComplete`, and `Panic`. Every additional stage multiplies event volume, and the volume matters because the API server keeps request context in memory while the request is active. Detailed policies therefore carry a real control-plane cost, not just a storage cost. Metadata-level logging is cheaper, less leak-prone, and usually enough to answer who did what, when, and under which policy.

That trade-off becomes important when your control plane also handles tool runs, not only Kubernetes objects. A metadata record can preserve the identity of the caller, the resource touched, the approval outcome, and the resulting object name or digest. You do not need the full

request body to know that a run was denied for missing egress scope or that a launch succeeded with a particular runtime class. Where you do need deeper detail, restrict it to the control-plane API that computes or launches the run, not to the entire surrounding cluster.

**Correlation without over-logging** A forensic workflow lives or dies on correlation. The control plane should assign a stable run ID before it performs any side effects, then thread that identifier through audit annotations, proxy headers, credential issuance records, and artifact metadata. That gives you a single anchor for reconstruction. If the agent session changes, reconnects, or retries, the run ID remains stable.

Audit annotations are especially useful when you want to preserve policy decisions without logging full bodies. A controller can attach the effective scope hash, the selected sandbox type, the policy version, and the reason for denial or approval to the audit record. Those annotations are compact, reviewable, and much less likely to contain secrets than a raw request or response payload. They also survive across heterogeneous runtimes because they describe the decision, not the substrate mechanics.

The mistake to avoid is coupling audit to transient connection state. A websocket session, a proxy hop, or an agent process may come and go during a run. If the audit trail uses those unstable identifiers as the primary key, incident reconstruction becomes guesswork. Use them as auxiliary metadata only.

**What to redact and what to retain** Redaction policy should be explicit, not an afterthought. The safest default is to retain structure and discard content:

- retain resource names, digests, and run IDs;

- retain policy decisions, denial reasons, and timestamps;
- retain bounded labels or annotations that identify the run;
- redact prompts, environment variables, tokens, and file bodies;
- hash or reference artifacts instead of embedding them.

This matters because the same fields that help operators often carry secrets. Environment blocks may contain tokens. Prompt text may contain tenant data. Output artifacts may contain the exact data the tool was allowed to inspect but not to redistribute. A good audit pipeline preserves the fact that an artifact existed, who approved it, and which digest or reference identifies it, without storing the artifact wholesale.

Where you need to preserve something for later analysis, store a digest and a pointer to the controlled evidence store. That gives responders the ability to recover the object under proper process while keeping the audit index itself narrow.

**Forensic packaging and chain of custody** Investigation needs more than logs. When an incident crosses the threshold from routine debugging to evidence preservation, you need a workflow that can freeze selected state without creating an expensive image of every run. The useful set is usually small: a run manifest, a credential issuance record, selected runtime logs, the approved outputs, and any runtime-denial streams such as seccomp or MAC violations.

The chain-of-custody question is straightforward: can you prove that the evidence you later inspect corresponds to the run you think it does? Stable digests and immutable references are the answer. If the sandbox produced an artifact, hash it immediately and record the hash in the audit trail. If the runtime emitted a denial event, record the event stream reference and the run ID. If a responder needs to reconstruct the run

later, they can follow those references into a controlled evidence store rather than searching a broad log index that may contain sensitive user data.

A good forensic pipeline therefore separates three layers:

- *Audit index*: compact, searchable metadata.
- *Evidence store*: controlled retention of selected artifacts and run-time traces.
- *Incident bundle*: a temporary, case-specific collection of digests, references, and approvals.

That structure keeps the common path cheap and the incident path deliberate.

**A retention model that matches risk** Not every event deserves the same retention window. Admission outcomes, policy decisions, and run manifests often need longer retention because they are the only durable record of why a run was allowed. Ephemeral runtime traces can usually expire sooner once the incident is closed. Raw content, if you retain it at all, should live only in a restricted evidence store with explicit access controls and a short, justified lifetime.

The retention policy should also match the sensitivity of the event class. Metadata-only records are relatively safe to keep longer because they usually do not contain tenant content. Request-response bodies are different. If you ever capture them, you should do so only for narrow cases, with clear justification and short retention. A blanket request-response policy across all namespaces or all tool runs is how audit systems become shadow data lakes.

**How audit behaves across sandboxes** The runtime substrate changes the shape of the evidence, but not the telemetry contract. A container run may produce pod events, cgroup metrics, seccomp denials, and proxy logs. A microVM run may produce guest logs, VM life-cycle events, and host-side mediation records. A nested container-in-microVM run may produce both. Your audit pipeline should normalize those sources onto the same run ID and manifest reference.

That normalization is important because the incident responder does not want to learn three different lookup paths for three runtime classes. The run record should say which substrate executed the work, which runtime class it used, and where the correlated evidence lives. If the tool was launched in a guest, the evidence store should still speak the language of the run manifest, not the host's debugging conventions. That keeps the forensic workflow coherent across heterogeneous fleets.

**A minimal forensic event flow** The diagram shows the intended asymmetry. Most events stay compact and searchable. Only a narrow subset is promoted into evidence storage. Only an even narrower subset is assembled into an incident bundle. If everything lands in the same index, you have lost the distinction between observability and retention.

**Incident workflow and selective expansion** A responder should start from the run ID and the manifest digest, not from free-form logs. From there, they can pull the policy decision, the runtime class, the issued identity, the proxy decisions, the output references, and the denial events. If the case requires deeper inspection, they can expand only the relevant evidence objects.

That workflow is far more efficient than capturing everything by default. A metadata-level trail gets you to the relevant run quickly. A controlled evidence store lets you inspect selected runtime artifacts

without exposing every prompt, secret, or temporary file from every invocation. The responder can also prove that the collected evidence matches the recorded digests, which is essential when the output itself may be part of the incident.

One subtle advantage of this model is that it supports both fast triage and deep investigation. Triage usually needs only policy, identity, and output references. Deeper work may need host-level traces, guest logs, or artifact snapshots. You do not pay the cost of deep capture unless the case merits it.

**Avoiding common audit failures** Four mistakes recur in mature systems.

First, logging full environment blocks. That usually exposes secrets with almost no investigative benefit.

Second, storing raw prompts or response bodies in central audit indices. That creates a data leak in the very system meant to explain access to sensitive data.

Third, dropping correlation IDs at proxy or gateway boundaries. Once that happens, you can no longer join control-plane decisions to run-time actions.

Fourth, collecting so much denial noise that real incidents vanish into the baseline. A flood of low-value events is not observability. It is obscurity.

A better rule is to treat every new field as a liability until you can justify its retention, redaction, and access path. If a field does not help you answer a specific forensic question, do not retain it in the main audit path.

**Compatibility and implementation notes** The audit back end, the orchestrator, and the runtime often evolve independently. That means your event schema must tolerate version drift. When the runtime changes how it names a sandbox or how it emits denial events, the control plane should still map those details to the stable run ID. Likewise, when you change policy versions or admission controllers, preserve the policy version in the audit record so a later reviewer can reconstruct the exact rules that applied at launch time.

Audit policy should also be explicit about what it omits. Excluding `managedFields` reduces noise and keeps the log closer to the actual policy and identity story. Using audit annotations for policy outcomes gives you traceability without forcing request or response bodies into the record. Those choices are not cosmetic. They directly shape whether the telemetry system remains a trustworthy investigation aid or drifts into a secondary repository of sensitive user content.

The clean operating stance is simple: log the decision, not the payload; record the digest, not the object; preserve the reference, not the secret. When that discipline holds, you can investigate tool runs across containers and microVMs without turning the audit pipeline into the very leak it was meant to help you detect.

## **7.6. Enforcing Supply Chain and Cluster Policy at Admission Time**

A secure runner can still be undermined before the process ever starts. If the cluster accepts mutable image tags, privileged pods, or deployment specs that bypass the broker's assumptions, the runtime boundary only contains what survived admission. The useful mental model is simple: the control plane decides what may run, the cluster must refuse anything that does not match that decision, and the image provenance

layer must prove that the artifact admitted is the artifact launched.

### A concrete admission path

A practical Kubernetes-backed baseline uses three gates: a structural policy floor, an invariant validator, and an image-verification controller. The first gate rejects obviously dangerous pod settings, the second enforces cluster-native invariants, and the third ties the image to a verified digest and provenance record.

```
apiVersion: v1
kind: Namespace
metadata:
  name: tool-runs
  labels:
    pod-security.kubernetes.io/enforce: "restricted"
    pod-security.kubernetes.io/enforce-version: "v1.30"
---
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionPolicy
metadata:
  name: toolrun-invariants
spec:
  matchConstraints:
    resourceRules:
      - apiGroups: ["control.example"]
        apiVersions: ["v1"]
        operations: ["CREATE", "UPDATE"]
        resources: ["toolruns"]
  validations:
    - expression: "object.spec.scope.network == 'deny'"
      message: "tool runs must start with deny-egress"
---
apiVersion: policy.sigstore.dev/v1alpha1
kind: ClusterImagePolicy
metadata:
  name: tool-images
spec:
  images:
    - glob: "registry.example/tool-*"
  authorities:
    - keyless:
        identities:
          - issuer: "https://token.actions.githubusercontent.com"
            subject: "repo:example/tool:*"
```

That example shows the shape, not the only possible implementation. The point is that you enforce invariants at admission, not by convention in the launcher. Pod Security Admission blocks dangerous pod primitives. ValidatingAdmissionPolicy keeps scope rules close to the cluster and avoids a remote webhook for simple invariants. An image policy controller verifies the signature or attestation of what the scheduler is about to run.

### **Why admission is part of the security model**

Admission is not administrative plumbing. It is the point where you prevent a deployment object from widening the trust boundary after the broker has already narrowed it. If the control plane says a tool may run only with read-only roots, no privileged escalation, default-deny egress, and a verified image, the cluster must refuse a spec that reintroduces writable host paths or an unverified tag.

That matters because runtime isolation is always downstream of artifact and spec integrity. A compromised image can ship data exfiltration logic, extra binaries, or hidden startup paths. A privileged pod can undo mount, network, or capability restrictions regardless of how careful the broker was. Admission is the last place where you can still reject those mistakes without ever starting the workload.

### **The policy floor: Pod Security Admission**

Pod Security Admission gives you a built-in baseline for pod hardening. Use it as the floor, not the whole building. The `restricted` profile is the closest match for untrusted tool runners because it blocks the obvious escalation paths: privileged containers, host namespaces, unsafe capabilities, and other settings that should never be present in a controlled sandbox.

Version pinning matters. Pod Security labels can be pinned per namespace, which means you should record the policy version explicitly. That

avoids silent behavior changes when cluster semantics evolve. If one namespace runs a different policy version than another, the difference should be deliberate and visible, not an accidental consequence of a cluster upgrade.

Use the policy floor to eliminate classes of configuration that no later layer should have to inspect. Then let higher-level validators reason about tool-specific invariants such as runtime class, required annotations, or approved volume shapes. If you ask a signature controller to compensate for a privileged pod spec, you have already failed the admission design.

### **Cluster-native invariants with CEL**

`ValidatingAdmissionPolicy` is useful when you need in-process validation for rules that do not require a network round-trip. CEL expressions let you encode structural invariants directly in the cluster: required labels, forbidden fields, permitted runtime classes, or scope invariants attached to a custom resource such as a `ToolRun`.

That makes it a good fit for brokered execution because the broker emits a manifest and the cluster validates that manifest before it becomes a pod. If the broker says a run must use `runtimeClassName: kata`, a CEL rule can reject a spec that omits it or substitutes a weaker class. If the broker says egress must start in deny mode, the validator can reject any manifest that asks for open network posture.

The value here is not expressive power alone. It is operational clarity. A cluster-native validator is easier to reason about than an out-of-process webhook for simple invariants, and it avoids coupling launch correctness to an external service that might fail open or time out. Reserve external webhooks for the cases where you truly need mutation, richer contextual logic, or supply-chain proofs.

### **Image provenance and digest resolution**

Admission must verify the artifact that will run, not the tag that happened to name it when the request was admitted. Mutable tags create a race: the same `tool:latest` reference can point to a different image later, after review but before scheduling. Digest-based resolution closes that gap. A controller such as Sigstore Policy Controller can resolve the image to a digest, verify signatures or attestations, and reject admission if the artifact does not match policy.

That policy layer is stronger when you express trust in terms of verified identity and provenance rather than repository naming alone. Cosign keyless signing, for example, uses short-lived Fulcio certificates bound to an OIDC identity and records the signing event in Rekor. That gives you a provenance path that admission can inspect without embedding long-lived signing keys into the build system or the image. The admission decision can then answer a narrow question: did a trusted identity sign the exact digest the cluster is about to run, and does that identity satisfy the policy attached to this workload?

### **How image-policy matching actually composes**

When you use Sigstore-style policy matching, the semantics matter. Multiple matching `ClusterImagePolicy` objects compose conjunctively: all applicable policies must pass. Within one policy, multiple authorities compose disjunctively: any authorized signer may satisfy that policy. That gives you a clean way to separate organizational trust rules from workload-specific trust rules.

For example, one policy might require that all tool images come from a particular repository family and be signed by your build system. A second policy might require that images for high-risk interpreters also carry a specific attestation from a security pipeline. The cluster admits only when both policies succeed. That is exactly the behavior you want when a low-risk internal helper and a tenant-facing code executor share the same cluster but not the same assurance bar.

This composition is valuable because it avoids the common trap of creating one monolithic rule that is hard to review and even harder to evolve. Instead, you layer policies by concern: repository identity, signing identity, attestation type, and workload class.

### **Aligning admission policy with the runtime scope model**

The admission layer must encode the same scope assumptions as the broker. If the broker computes an effective scope that denies host networking, the cluster should reject any manifest that requests it. If the broker pins a digest, the cluster should reject a mutable tag. If the broker requires a specific namespace label or run annotation, the cluster should enforce that label before launch.

A robust control plane therefore treats the run manifest as a contract. The broker computes the contract; admission verifies that the deployment object still matches it; the runtime only executes the verified result. That chain prevents a later manifest edit from widening privileges after policy approval. It also makes audits simpler because the policy decision and the launched artifact refer to the same immutable manifest or digest.

### **The enforcement stack in practice**

A useful stack for Kubernetes-backed tool runners looks like this:

- namespace-level Pod Security Admission establishes a hard floor;
- `ValidatingAdmissionPolicy` enforces structural invariants;
- an image-policy controller verifies signatures, attestations, and digests;
- the launcher or runtime class applies the selected sandbox tier;
- the network policy and identity layers enforce the runtime scope.

The order matters. Structural rules should fail fast before a signature controller burns time resolving an image that should never have been eligible. Signature verification should happen before the scheduler pulls the image. Runtime class selection should happen before the pod is materialized. If you invert those steps, you create avoidable load and make failure modes harder to diagnose.

### **Admission event flow across build, registry, and launch**

The diagram captures the pre-flight lock. Build outputs are immutable. The registry stores a reference, not a trust decision. Admission checks the object against policy. Only then does the runtime start the sandbox. Any path that skips the gate reintroduces the very ambiguity the broker is designed to remove.

### **Cluster admission versus external webhooks**

You do not need an external webhook for every rule. In fact, you should avoid one when the rule is a simple invariant over the object being admitted. `ValidatingAdmissionPolicy` is a better fit because it lives in the API server path, uses CEL, and does not depend on another service being available at the exact moment a run is scheduled.

External webhooks still matter when the policy needs mutation, additional provenance checks, or integration with a system that computes trust dynamically. A signature controller is one such case. A secret injector may be another. The design principle is to keep the built-in admission path responsible for the platform floor and push richer behavior outward only when necessary.

That split reduces failure coupling. If the webhook serving signature verification is unavailable, you want to fail closed on image admission, but you do not want to block unrelated structural validation. If a CEL rule can reject an invalid runtime class immediately, that rejection should not wait on any external dependency.

## **Anti-patterns that weaken admission**

Four failures recur in practice.

First, teams admit mutable tags and rely on the registry to remain unchanged. That is a race condition, not a policy.

Second, they verify signatures but ignore identity context. A signature without a trusted signer or a policy binding does not tell you enough.

Third, they treat admission as advisory and allow the launcher to widen the spec if a later step fails. That turns the launcher into a policy bypass.

Fourth, they verify images but not deployment specs. A perfectly signed image can still be dangerous if the pod is privileged, host-networked, or mounted with a writable host path.

The correct response is to verify the whole execution envelope. The image must be approved, the spec must be admissible, and the runtime class must match the broker's scope. If one of those checks is missing, the platform can still execute something, but it can no longer claim that the execution matched the reviewed policy.

## **Compatibility and version discipline**

Admission features evolve, and the policy surface shifts with them. Pod Security Admission labels should be version-pinned per namespace so that a cluster upgrade does not silently change the interpretation of `restricted`. `ValidatingAdmissionPolicy` depends on API support in the cluster version you deploy. Signature controllers and policy engines also evolve their matching semantics, so a platform team should record the controller version alongside the policy spec.

That version discipline belongs in the run manifest and in the audit trail. If a run was admitted under one version of the policy engine and another run under a later version, you want the difference to be visible. That is especially important for high-risk tool classes where even

a small change in policy semantics can widen the trust boundary.

### **What a finished admission path gives you**

A mature pre-flight gate gives you a simple guarantee: if the cluster admitted the run, then the image was verified, the pod spec matched the broker's scope, and the deployment object complied with the hard floor for that trust tier. The runtime still needs egress policy, identity scoping, and forensic logging, but those layers now operate on a workload that was already screened for provenance and cluster safety.

That is the last place in the stack where you can stop a bad execution before it starts. Once the image is pulled and the sandbox is running, containment becomes a runtime problem. Admission keeps it a policy problem.

