Secure Execution Environments, Workload Identity, Just-in-Time Credentials, Secrets, Network Egress, and Sandbox Lifecycle for Agent Fleets

Executive Summary

Autonomous software agents introduce profound execution risks because they dynamically generate and run code based on natural-language inputs that may be entirely attacker-controlled. Standard containerised execution environments, designed for deterministic applications with fixed instruction sets, are fundamentally inadequate for agentic workloads. This research report details an exhaustive, implementation-focused architecture for safely executing untrusted code, tools, dependencies, and shell commands across an agent fleet. The resulting design establishes a provider-neutral platform that mitigates severe threats—including prompt injection, cross-project memory contamination, credential exfiltration, and sandbox escapes—by implementing hardware-level microVM isolation, brokered out-of-band credentials, default-deny network topologies, and cryptographically verified workload identities. The architecture ensures that workloads operate with the absolute least privilege, eliminating long-lived credentials and persistent memory access outside heavily authorised boundaries.

Recommended Direction

The architectural imperative is to strictly decouple the orchestration control plane from the untrusted execution plane. The agent platform must initiate an incremental maturation path, commencing with a lean baseline of hardened rootless containers for internally trusted operations. This initial phase must instantly enforce rigid API boundaries, generic sandbox abstractions, workload identity protocols, and zero-trust network policies. Following this, the platform must systematically transition to hardware-accelerated microVM isolation for all multi-project, internet-connected, and untrusted workloads to ensure kernel-level separation. Furthermore, operations demanding high-privilege credentials—such as production cloud mutations or core repository modifications—must rely entirely on an out-of-sandbox tool proxy pattern, ensuring raw secrets never materialise within the agent's memory space.

Scope and Assumptions

This design is governed by strict practical maturity constraints, assuming incremental development from an early milestone (version 0.4) to a mature production state (version 1.2 and beyond). A complete bare-metal microVM infrastructure is deferred to the later milestone

to prevent over-engineering. However, the early milestone must rigidly define the foundational abstractions: the provider-neutral sandbox interface, lifecycle contracts, identity schemas, and network-policy enforcement mechanisms. Establishing these contracts immediately prevents technical debt and ensures that the eventual migration from containers to microVMs does not require refactoring the control plane, task model, or policy engine. The platform is assumed to launch under the management of a small operating team, necessitating low operational complexity initially, while preserving a seamless architectural migration path toward a secure, heavily isolated multi-tenant enterprise system.

Threat Model and Trust Boundaries

Agentic systems operate via probabilistic logic, combining natural language with executable code, which inherently collapses the traditional distinction between data and instructions. Conventional static analysis fails because the agent generates novel, unreviewed code at runtime.[1] The threat model accounts for vectors unique to this paradigm.

Prompt injection represents an unpreventable, deterministic threat where untrusted instructions—harvested from external repositories, issue trackers, or web pages—manipulate the agent into executing malicious operations.[1] Repository poisoning introduces another severe vector; attackers can inject malicious configurations, such as overwritten .git/hooks or manipulated .gitmodules containing trailing carriage returns (CVE-2025-48384), to achieve remote code execution during routine repository checkouts.[2] Sandbox escapes involve malicious code traversing poorly configured mount points, abusing host runtime sockets, or exploiting shared-kernel vulnerabilities to compromise the underlying node.[5] Credential theft occurs when compromised workloads scrape memory, parse environment variables, or scan disk caches to locate and exfiltrate long-lived tokens for lateral movement.[5] Resource exhaustion manifests as denial-of-service attacks, where runaway agents initiate fork bombs, infinite build loops, or massive external downloads to exhaust host capacity.[1] Finally, cross-project contamination threatens multi-tenant integrity, where a workload accesses data, dependency caches, or memory belonging to an unrelated project executing on the same physical host.

To neutralise these vectors, explicit trust boundaries must be enforced. The core trust boundary exists between the execution plane (the sandbox workers) and the control plane (the orchestration system). The sandbox environment, the executed repositories, and the external internet must be treated as universally hostile and completely untrusted. Conversely, the credential broker, secrets manager, policy engine, identity provider, and task ledger reside firmly inside the trusted control plane. These trusted components must never accept unauthenticated connections from the execution plane, mediating all requests via cryptographically verified APIs and enforcing strict unidirectional control flows.

Trust Tiers and Execution Profiles

Every task or run must be deterministically assigned to an execution profile based on its inherent risk, authorised by the policy engine prior to sandbox instantiation. Agents are strictly

prohibited from determining or lowering their own isolation requirements. The profile evaluation flow begins when task metadata is submitted to the scheduler; a policy-as-code engine evaluates the task's origin, intended destination, and requested toolsets to map it to a specific trust tier.

Profile Name	Permitted Runtime	Filesystem & Memory Access	Network Egress	Credential Availability	Approval
Trusted Read-Only	Hardened Container	Read-only repository mount	Blocked entirely	Scoped read-only access	Automatic
Controlled Build	Rootless Container / microVM	Ephemeral /tmp, isolated cache	Specific package registries	Task-bound identity only	Automatic
External Research	Remote Browser Sandbox	Volatile memory only, strict quota	Specific allowlist + Proxy	None	Automatic
Untrusted Execution	Hardware microVM	Isolated copy-on-write	Locked to LLM API only [5]	Short-lived, tool-proxied	Required
Cloud Mutation	Hardware microVM	Isolated, no persistence	Cloud API only	JIT, Operator approved	Human
Quarantined	Hardware microVM	Differential snapshot capture	Severed completely	Revoked instantly	Manual

Isolation-Technology Comparison

Selecting the appropriate execution-isolation technology dictates the structural resilience of the execution plane. The architecture evaluates multiple primitives to balance security, performance, and complexity.

Hardened rootless containers provide a foundational isolation layer by executing the container

daemon and workloads without root privileges. Utilizing subordinate user and group mappings, namespaces, and cgroups v2, this approach mitigates basic privilege escalation.[1] Security is enhanced by applying strict seccomp profiles, dropping all Linux capabilities except those strictly required, and setting PR_SET_NO_NEW_PRIVS to prevent child processes from gaining elevated permissions.[5] However, rootless containers fundamentally share the host kernel. They do not protect against kernel zero-day exploits, making them insufficient for executing arbitrary code downloaded from untrusted internet sources or supporting true multi-tenant workloads.

The gVisor runtime provides a more robust boundary by intercepting system calls and processing them through a user-space kernel (runsc) written in memory-safe Go. This drastically reduces the attack surface of the host kernel. gVisor offers a worthwhile security improvement for internal multi-project use where code is generally trusted but requires protection against accidental cross-contamination. However, it suffers from compatibility limitations with certain low-level system calls and nested container workflows, and imposes a moderate performance overhead on filesystem and networking operations.

Firecracker and microVM isolation represent the gold standard for executing unvetted, third-party code. By leveraging the hardware Memory Management Unit (MMU) via KVM, microVMs enforce absolute physical separation between the guest and host.[5] Advanced implementations utilize pre-warmed pools of initialized snapshots, reducing cold-start latency to mere milliseconds, thus rivalling container startup speeds.[6] MicroVMs support standard nested Docker workflows and provide unparalleled resistance to cross-project compromise. The primary drawback is operational complexity, as they typically require bare-metal infrastructure and sophisticated networking orchestration.

Remote ephemeral sandbox services offer a managed alternative, abstracting away the operational burden of maintaining hypervisors. These services frequently utilize microVMs under the hood and provide robust APIs for programmatic sandbox lifecycle management. While they eliminate infrastructure overhead, they introduce dependencies on vendor-specific APIs, higher per-run costs, and potential data residency challenges.

A weighted evaluation indicates that hardened rootless containers are suitable only for early trusted internal dogfooding. Production internal multi-project environments should graduate to gVisor. However, for untrusted repository execution and future multi-tenant use, self-managed microVMs or managed remote microVM services are absolutely mandatory to guarantee hardware-level isolation.

Sandbox Runtime Abstraction

To avoid vendor lock-in and preserve provider independence, the platform must implement a generic sandbox runtime abstraction. This interface allows the orchestration scheduler to swap execution backends seamlessly without altering task semantics. The abstraction must eschew lowest-common-denominator limitations by adopting a capability-discovery model.

The generic interface mandates standard responsibilities: create allocates logical resources; prepare retrieves images; attest cryptographically verifies the environment; start initiates execution; stream_output connects log buffers; checkpoint captures state; quarantine isolates

the workload; and destroy permanently obliterates the instance and its cryptographic keys. The scheduler requests environments based on an array of required capabilities, such as linux-x86_64, snapshot-support, nested-virtualisation, private-network, or gpu-enabled.
To support provider-specific optimisations without polluting the generic interface, the schema includes an extensions or provider_overrides payload. This allows a microVM provider to accept a proprietary pre-warmed snapshot ID to accelerate startup [6], while a container provider simply ignores it. This architectural pattern guarantees that the control plane remains entirely agnostic to the underlying execution mechanics, enabling a smooth migration from version 0.4 containers to version 1.2 microVMs.

Sandbox Lifecycle

The sandbox lifecycle is governed by a canonical state machine managed deterministically by the control plane. This ensures that transient network failures or worker crashes do not result in orphaned, zombie workloads.
The lifecycle initiates in the Requested state, pending policy engine evaluation. Upon approval, it transitions to Authorised and then Provisioning, where logical resources are allocated. During Attesting, the identity provider verifies the node and issues a workload identity. The sandbox becomes Ready and, upon accepting commands, enters the Running state. If a long-running task is suspended, it transitions to Paused while a snapshot is captured, and later to Resuming. Should the policy engine detect anomalous behaviour, the state forcefully shifts to Quarantined. Upon completion or timeout, the sandbox enters Destroying, followed by Destruction-Verifying, and finally Destroyed.
Robust failure handling is critical. If the control plane crashes during the Provisioning phase, the execution worker will eventually notice a lost heartbeat lease and forcefully terminate the local instance. If a sandbox fails to prove destruction, it is flagged as Orphaned in the task ledger, triggering an aggressive out-of-band reconciliation daemon that sweeps worker nodes, matches running process IDs against active leases, and issues SIGKILL to unmapped processes.

Workload Identity

Identities must never be granted to a conceptual "agent role"; authority belongs exclusively to a specific task run executing within a specific sandbox. The architecture must adopt the Workload Identity in Multi-System Environments (WIMSE) paradigm, leveraging SPIFFE (Secure Production Identity Framework for Everyone) and SPIRE (SPIFFE Runtime Environment) for cryptographic identity attestation.[7]
The identity model distinguishes the human operator, the logical project, the semantic task, and the physical sandbox. The authoritative execution identity is a composite, binding the task intent to the attested sandbox environment. When the sandbox boots, a SPIRE agent on the worker node attests the workload's properties (e.g., namespace, cgroup, microVM ID) and issues a SPIFFE Verifiable Identity Document in JSON Web Token format (JWT-SVID) via the Workload API.[7]
The JWT-SVID claims schema must be strictly defined. The sub (subject) claim must contain

the unique SPIFFE ID (e.g., spiffe://platform.internal/project/123/task/456). The aud (audience) claim must be tightly restricted to the target service (e.g., the credential broker) to prevent cross-service replay attacks.[10] The exp (expiration) claim must enforce an aggressive time-to-live, typically under five minutes. Additional custom claims include the execution profile, repository context, and an approval reference. Static, long-lived API keys are classified as an architectural anti-pattern and must be wholly eradicated.[8]

Authorisation

Authorisation must be decoupled from application logic and enforced via policy-as-code, utilizing engines such as Open Policy Agent (OPA) or Cedar.[12] Role-based access control is insufficiently granular for dynamic agent tasks; attribute-based and relationship-based access control paradigms are required to evaluate the rich context of each tool call.

When a workload attempts an operation, the infrastructure validates the caller's JWT-SVID, verifies the task's ownership within the project boundary, and cross-references the requested action against the task's approved execution profile. The policy engine evaluates whether human approval exists for destructive actions, whether the task budget is unexhausted, and whether the JWT-SVID remains valid.

To prevent privilege accumulation across retries or sandbox re-instantiations, authority must not survive task reassignment. Furthermore, the architecture must enforce "delegation without escalation".[13] If an agent spawns a child sub-task, the child's capability scope must be a mathematically provable subset of the parent's authority, and the child's token expiry timestamp must be strictly less than or equal to the parent's.[13] This guarantees that compromised child agents cannot elevate privileges or persist beyond the original intent.

Just-in-Time Credentials

Workloads must receive short-lived, narrowly scoped authority strictly on a Just-in-Time (JIT) basis. The credential architecture leverages workload identity federation to exchange the agent's SPIFFE JWT-SVID for target-specific temporary tokens, such as OAuth 2.0 session tokens or cloud provider assumed roles.[8]

Credential lifetimes must precisely map to the operational context: a single tool call may receive a token valid for 5 minutes; a repository clone may receive 15 minutes; while a long-running build may receive a 60-minute token.[13] Renewal policies must never silently extend task authority beyond the absolute maximum lifetime approved by the operator. If a task requires extended execution, the agent must present an updated cryptographic approval receipt to the credential broker. The platform vastly prefers OIDC federation and temporary session credentials over stored personal access tokens or deploy keys, enforcing revocation-by-expiry as the primary security mechanism alongside out-of-band revocation channels for immediate compromise response.[13]

Credential Broker and Tool Proxy

A critical architectural decision involves comparing two credential delivery approaches.

Approach A issues narrowly scoped temporary credentials directly into the sandbox, allowing the workload to call external services. Approach B entirely abstracts the credential, utilizing a trusted tool proxy outside the sandbox; the agent sends an intent payload to the proxy, which authenticates the request, appends the secret, and executes the operation on the agent's behalf.[15]

Approach B (the Tool Proxy pattern) is vastly superior and absolutely mandatory for high-impact administrative actions, production mutations, database modifications, and interactions with proprietary model-provider APIs. By keeping the raw credential outside the untrusted execution environment, the risk of credential theft via memory scraping, prompt injection exfiltration, or accidental inclusion in log dumps is entirely eliminated.[5] Approach A is permissible only for high-volume, low-risk operations—such as pulling public dependencies or streaming large artifact uploads—where the overhead of proxying massive payloads introduces unacceptable latency and bottlenecking, provided the credential scope is strictly read-only and ephemeral.

Secrets Management

The lifecycle of secrets—from creation to destruction—must preclude persistent exposure. The platform should leverage robust secret managers (e.g., Vault or cloud-native equivalents) to support per-task secret grants, dynamic secret generation, and automatic rotation.

Secret delivery mechanisms carry varying risks. Environment variables (.env) are notoriously insecure; they routinely leak into child-process inheritance trees, crash dumps, build logs, shell histories, and AI model context windows.[5] Relying on regular expressions to redact secrets from output streams is a flawed, reactive strategy that attackers easily bypass via encoding or obfuscation.

Instead, the architecture enforces strict prevention. Secrets must be delivered via the brokered Tool Proxy whenever possible. When a secret must reside locally, it must be injected into an ephemeral, in-memory tmpfs volume or accessed strictly via runtime API retrieval using a local socket descriptor. The architecture acknowledges the severe limitations of memory zeroisation; high-level garbage-collected languages (like Python or Node.js) do not guarantee deterministic wiping of memory registers before reallocation.[16] Therefore, prohibiting secrets from entering the sandbox memory space is the only mathematically sound defense.

Git Identity and Credentials

Providing an agent with arbitrary execution capabilities against Git repositories introduces profound supply-chain risks. The Git-specific security model must restrict access to narrowly scoped installation tokens bound to a single repository and base commit, effectively neutralizing cross-repository movement.

An attacker can weaponise repository operations by manipulating configuration files. If an agent executes within a poisoned workspace containing a malicious .git/config, an attacker can override core.hooksPath to point to a malicious executable. When the agent performs routine Git operations (like commit or merge), Git will automatically execute the attacker's script,

resulting in a catastrophic sandbox escape (a vector recently exploited in CVE-2026-26268).[2] Furthermore, CVE-2025-48384 demonstrates that trailing carriage returns in .gitmodules can trick Git into path traversal, overwriting .git/hooks during recursive submodule initialisation.[4]
To mitigate these threats, the platform must deterministically force git config --global core.hooksPath /dev/null within the hardened base image, rendering hook execution impossible.[3] Recursive submodule cloning must be strictly disabled for untrusted inputs.[4] Furthermore, repository checkouts must be pre-materialised by the trusted control plane and mounted into the sandbox as read-only volumes where mutation is not explicitly authorised.

Cloud Workload Credentials

To interact with major cloud providers (AWS, GCP, Azure) without embedding long-lived IAM keys, the platform relies on Workload Identity Federation. The control plane maps the agent's SPIFFE JWT-SVID to an OIDC federation trust policy within the target cloud.[10] The cloud's Security Token Service (STS) validates the JWT-SVID's iss (issuer), sub (subject), and aud (audience) claims, returning a short-lived assumed role.[10]
The platform enforces rigid differentiation between authority levels. Read-only diagnostic authority (e.g., viewing logs) maps directly to the automated sandbox identity. Conversely, destructive authority or production mutations mandate a conditional policy evaluation outside the sandbox, requiring an independently validated, cryptographically signed human approval reference before the STS exchange is permitted. This ensures that a prompt-injected agent cannot autonomously delete cloud infrastructure.

Network-Egress Architecture

Unrestricted network egress is the primary vector for command-and-control establishment and data exfiltration. The architecture mandates an identity-aware, default-deny network-egress posture.[5] Standard namespace-level isolation and basic container network policies are insufficient.
The data plane implementation must avoid traditional iptables, which suffers from O(n) latency degradation as rules scale across thousands of dynamic services. Instead, the egress proxy must utilise nftables, leveraging verdict maps to achieve highly performant O(1) packet processing.[20]
Network policies are enforced dynamically via DNS interception. A stateful packet filter intercepts all outbound DNS queries from the microVM. If the requested domain (e.g., api.anthropic.com) matches the task's allowlist, the proxy resolves it, snoops the returned A-record, and dynamically generates an nftables rule permitting egress to that specific IP address.[5] The firewall rule inherits the exact Time-To-Live (TTL) of the DNS record, expiring automatically.[5] Because IP addresses must be learned and validated through DNS first, hardcoded IP addresses initiated by malicious code fail instantly.[5]
Furthermore, the proxy must neutralize DNS rebinding attacks, where an attacker's DNS server rapidly shifts responses to target private internal networks.[21] CoreDNS plugins must be configured to aggressively strip RFC1918 private addresses (10.0.0.0/8), loopback addresses

(127.0.0.0/8), and cloud metadata service endpoints (169.254.169.254) from all upstream DNS responses, rendering internal enumeration impossible.[22]

Browser and Internet-Research Isolation

Agents requiring internet research face unique risks, including drive-by downloads, browser exploits, and persistent tracking. Workloads requiring browser automation must never run in the same sandbox as repository code. They must execute in physically separate, ephemeral remote browser sandboxes.

To prevent session theft and cookie exfiltration, authenticated browsing must utilise a content-extraction proxy. The trusted proxy holds the long-lived authentication cookies and renders the target application, streaming only a sanitized, read-only representation of the Document Object Model (DOM) or a visual VNC feed to the agent sandbox. This guarantees that even if the agent is fully compromised by malicious webpage content, it cannot extract session credentials or persist access.

Filesystem and Resource Isolation

Filesystem isolation must distinguish between volatile caches and persistent state. Host directories must never be bind-mounted into the sandbox.[1] Repositories and worktrees are cloned into ephemeral tmpfs mounts or isolated copy-on-write overlay filesystems, ensuring destruction upon termination.

Guest hardening is paramount. The sandbox init process (PID 1) must be a hardened, statically compiled binary that replaces traditional vulnerable init systems.[5] Kernel sysctls must be heavily restricted: kernel.unprivileged_bpf_disabled=1 neutralises eBPF JIT spraying attacks, kernel.kptr_restrict=2 prevents kernel memory mapping, and kernel.sysrq=0 disables hardware interrupt keys.[5] Standard Linux capabilities must be dropped, retaining only minimal requirements (e.g., net_bind_service), reinforced by PR_SET_NO_NEW_PRIVS to eliminate setuid binary escalation.[5]

Resource governance leverages cgroups v2 to enforce strict quotas on cpu.max, memory.max, and process counts (pids.max). Bounding the PID limit is critical to containing fork bombs that would otherwise exhaust host scheduling capacity.[1] Soft limits trigger operator warnings, while hard limits result in emergency termination.

Image and Supply-Chain Security

Sandbox environments must boot from minimal base images, explicitly omitting unnecessary utilities like curl or wget unless required by the task profile. Image provenance must be cryptographically verified, enforcing digest pinning (e.g., sha256:...) over mutable tags to prevent upstream supply-chain injection. Software Bill of Materials (SBOM) generation and vulnerability scanning are mandatory admission checks.

To balance startup performance with reproducibility, base images are converted into pre-warmed microVM snapshots.[6] Developer-supplied container images, such as custom development-container definitions, must be treated as inherently hostile and restricted

exclusively to the highest-isolation microVM execution tier, ensuring that embedded malware cannot compromise the host.

Checkpointing and Resumability

Long-running or highly parallel agent tasks require checkpointing to optimize compute expenditure. MicroVM technologies allow the complete memory and disk state of an environment to be snapshotted and resumed in milliseconds.[6]

However, checkpointing introduces profound security implications: a snapshot freezes volatile memory containing active credentials, cookies, and decrypted secrets.[6] The architecture dictates that snapshots logically invalidate all local session state. Upon resumption, the runtime invokes a dedicated lifecycle hook (e.g., /aws/lambda-microvms/runtime/v1/resume).[25] This hook suspends application traffic while it deterministically flushes expired tokens, re-authenticates via the SPIFFE Workload API, and establishes fresh network proxies.[6] High-risk untrusted workflows must forbid memory snapshotting entirely, mandating metadata-only preservation in the task ledger to eliminate secret-retention risks.

Evidence and Forensic Capture

Transparent, auditable execution mandates an immutable forensic bundle for every run. Evidence must encompass task and sandbox identity claims, image digests, runtime versions, file diffs, executed shell commands, and detailed DNS request logs. The platform explicitly does not require the storage of hidden model reasoning or unrestricted chain-of-thought, which preserves privacy and intellectual property while focusing strictly on observable execution actions.

Evidence must be cryptographically hashed, signed, and streamed synchronously outside the sandbox via a dedicated, write-only local file descriptor. This prevents a compromised workload from tampering with its own historical audit trail. If evidence streaming unexpectedly halts, or if the workload attempts to manipulate the log buffer, the control plane must instantly assume compromise, issue a SIGKILL, and transition the sandbox to a quarantined state.

Quarantine and Incident Response

A deterministic quarantine mechanism protects the platform when preventive controls detect anomalies. Trigger conditions include blocked metadata-service access, unauthorized egress attempts, evidence tampering, or writes to restricted paths like .git/hooks/.

Operationally, quarantine does not mean immediate destruction; it means isolation for forensic analysis. The control plane instantly severs all network access via nftables and revokes the workload's SPIFFE JWT-SVID credentials. Execution is frozen via process suspension. The system captures a differential filesystem snapshot to analyze the payload. However, full memory dumps are strictly prohibited for multi-tenant untrusted workloads, as capturing memory risks preserving unrelated cross-tenant secrets or API keys that the attacker may have intentionally injected into the compromised space.

Teardown and Destruction Verification

Terminating a task successfully is meaningless unless the platform can cryptographically guarantee the complete destruction of the sandbox. The secure teardown sequence initiates by ceasing to accept new commands and stopping all child processes. The control plane revokes the workload identity from the SPIRE server, flushes external evidence streams, and detaches execution volumes.
Destruction verification relies on cryptographic erasure. The ephemeral encryption keys assigned to the per-task volumes are destroyed, rendering the physical block data mathematically inaccessible even if the underlying storage media is immediately reallocated to another tenant. Finally, to combat phantom billing and zombie processes, an out-of-band orphan-resource reconciliation daemon continuously polls the worker nodes, cross-referencing active PIDs against the authoritative task ledger, immediately culling any unmapped environments.

Control-Plane Protection

The execution plane must never possess direct, unrestricted access to the orchestration control plane. Authenticated interfaces must govern all interactions—including heartbeats, status updates, tool requests, and evidence streaming.
To prevent command injection or deserialization attacks originating from a compromised worker node, all payloads traversing the boundary must undergo strict schema validation and size limiting. The control plane relies on mutual TLS (mTLS) and request signing. Crucially, non-deterministic decisions, credential scoping, budget ceilings, and execution-profile selections must remain exclusively within the deterministic control-plane services, fundamentally untrusting any status message generated by the execution worker.

Deployment Topology

Deploying untrusted execution environments within the same Kubernetes cluster as the control plane introduces unacceptable container-escape risks, where a single kernel exploit could grant an attacker access to the orchestration database and identity provider.
The platform must adopt a cleanly segregated deployment topology. The trusted control plane is hosted in a managed Kubernetes or serverless environment. The execution workers operate on physically isolated, dedicated bare-metal hosts (a prerequisite for microVM hardware virtualization) residing in a separate Virtual Private Cloud (VPC). The execution plane communicates back to the control plane exclusively via an authenticated gRPC API. This hybrid topology provides the leanest operational burden for a small team while preserving a hardened migration path to enterprise-grade isolation.

Build-Versus-Buy Analysis

Resource allocation for a small team demands a strategic build-versus-buy approach to avoid dangerous custom implementations of critical security infrastructure.

Component	Strategy	Rationale & Recommendation
MicroVM Orchestration	**Buy / Adopt**	Building hypervisors is highly dangerous. Adopt open-source orchestrators (Firecracker, Cloud Hypervisor) or managed services.[6]
Workload Identity	**Adopt**	Leverage SPIFFE/SPIRE for cryptographic identity attestation; custom identity issuance often introduces fatal flaws.[7]
Policy Engine	**Adopt**	Integrate standard policy-as-code engines (OPA, Cedar) to separate authorisation logic from application code.[12]
Egress Gateway	**Adopt & Extend**	Utilise CoreDNS for rebinding protection [24] and NFTables for performance [20], extending them with custom dynamic DNS snooping.[5]
Credential Broker	**Build**	Highly specific to organisational task identity and execution profiles; requires custom integration with the specific tool proxy architecture.

Reference Architecture

The architecture strictly enforces unidirectional trust boundaries, ensuring the execution plane remains isolated from critical control components.

```
Code snippet
flowchart TD
  subgraph Control_Plane
    API
    PDP
    SPIRE
    Broker
    Ledger
  end

  subgraph Execution_Plane
    SandboxMgr
    Egress

    subgraph Sandbox["Hardware-Isolated MicroVM"]
      PID1
      Agent
      Repo
    end
  end

  API -->|Authorises via| PDP
  API -->|Dispatches task to| SandboxMgr
  SandboxMgr -->|Provisions & Attests| Sandbox
  SPIRE -.->|Issues JWT-SVID to| PID1
  Agent -->|Requests Tool Execution| Broker
  Agent -->|External Internet Traffic| Egress
  PID1 -->|Streams Tamper-Proof Logs| Ledger
```

Schemas and Contracts

Defining schemas early ensures decoupling between orchestration and execution. The examples below illustrate the required capability-driven abstractions.

Sandbox Request Schema (JSON):

```
JSON
{
  "task_id": "tsk_8f92a",
```

```
 "project_id": "prj_991b",
 "profile": "untrusted_execution",
 "capabilities": ["linux-x86_64", "microvm_isolation"],
 "image_reference": "sha256:4a3b...c9d1",
 "repository": {
  "url": "https://git.internal/project",
  "base_commit": "7f8a9b...",
  "submodules": false
 },
 "network_policy": "locked_llm_only",
 "resource_limits": { "cpu_cores": 2, "memory_mb": 4096, "pids_max": 512 },
 "teardown_policy": "cryptographic_erasure"
}
```

Workload Identity Claims (SPIFFE JWT-SVID):

JSON

```
{
 "iss": "https://spire.internal.platform",
 "sub": "spiffe://platform.internal/project/prj_991b/task/tsk_8f92a",
 "aud": ["https://credential-broker.internal"],
 "exp": 1722055000,
 "iat": 1722051400,
 "platform/execution_profile": "untrusted_execution",
 "platform/nonce": "8f39a0b1"
}
```

Security Fitness Functions and Conformance Tests

Prior to production deployment, every sandbox provider must pass an executable suite of adversarial security tests to guarantee the integrity of the execution boundary.

- **Capability Dropping Test:** The sandbox initiates a script attempting to execute a setuid binary to escalate privileges. Expected outcome: EPERM rejection enforced by PR_SET_NO_NEW_PRIVS.[5]
- **Metadata Service Denial:** The sandbox attempts curl http://169.254.169.254. Expected outcome: Connection timeout or NXDOMAIN, verifying CoreDNS rebinding protection.[23]
- **Cross-Sandbox Replay:** Task A attempts to submit Task B's JWT-SVID to the credential broker. Expected outcome: Policy Engine DENY due to identity mismatch.
- **Fork Bomb Containment:** The sandbox executes :(){ :|:& };:. Expected outcome: pids.max limit reached instantly, terminating the process tree without degrading host

performance.[1]

- **Workspace Poisoning Resilience:** The sandbox attempts to execute git config --global core.hooksPath /tmp/malicious. Expected outcome: File permission denied by the hardened base image.

Attack-Scenario Validation

Attack Scenario	Preventive Control	Detective Control	Recovery & Residual Risk
Agent instructed to upload code to external server	DNS Interception allows only approved endpoints; blocks hardcoded IP connections.[5]	Egress proxy logs blocked connection attempts.	Sandbox quarantined. Residual risk: DNS tunneling if allowed domains are excessively broad.
Dependency install script reads metadata credentials	CoreDNS strips 169.254.169.254 from resolution.[23]	API logs unauthorized assumed role requests.	Immediate task termination. Minimal risk due to hard network boundary.
Workload accesses another project's worktree	MicroVM enforces hardware MMU separation per task.[5]	Memory mapping attempts flag hypervisor alerts.	Cryptographic erasure upon failure. Negligible cross-tenant risk.
Workload prints injected secret to logs	Tool Proxy pattern prevents secrets from entering sandbox memory entirely.[15]	Structured log filtering (secondary defense).	Token rotation via JIT expiry. Residual risk: Model context caching.
Browser session downloads malicious binary	Remote Browser Isolation separates web rendering from execution workspace.	Executable signature scanning on downloaded artifacts.	Ephemeral browser container destroyed instantly. No persistent risk.

Sandbox replays token from another run	JWT-SVID requires precise aud and short exp claims.[11]	SPIRE logs mismatched token validation.	Sandbox quarantined. Mitigation requires robust clock synchronization.
Malicious project attempts cache poisoning	Cache-key isolation enforces strict per-project and per-branch cache hashing.	Hash mismatch alerts in artifact store.	Cache invalidated. Residual risk: Hash collision vulnerabilities.
Evidence streaming interrupted during suspicious activity	Write-only socket requires continuous heartbeat.	Ledger records incomplete stream.	Instant SIGKILL issued to worker process.

Performance, Cost, and Operational Metrics

To transition from a prototype to a mature enterprise platform, precise service-level indicators (SLIs) must be established and continuously monitored.

- **Sandbox Provisioning Latency:** Time from task admission to the first executed command. Target: < 2 seconds leveraging pre-warmed microVM snapshots.[6]
- **Credential Issuance Latency:** Time for SPIRE to mint and deliver a JWT-SVID. Target: < 50 milliseconds.
- **Teardown Latency:** Time to cryptographically erase volume keys and terminate the process. Target: < 1 second.
- **Orphan Rate:** Percentage of sandboxes persisting beyond task completion. Target: 0.00%, enforced by reconciliation loops.
- **Cost per Sandbox Minute:** Tracked continuously against idle-pool maintenance costs to ensure microVM pre-warming remains economically viable.

Phased Implementation Roadmap

A pragmatic, incremental roadmap is essential to deliver immediate value without compromising the target architecture.

Phase A — Architecture and Contracts (Immediate)

- *Objective:* Solidify system boundaries.
- *Implementation:* Finalize JSON schemas for the generic sandbox-provider interface, WIMSE identity claims, and network-policy contracts. Establish the lifecycle state machine. No execution code is deployed until contracts are versioned.

Phase B — Trusted Internal Execution (Target: Version 0.4)

- *Objective:* Lean baseline for trusted internal operations.
- *Implementation:* Hardened rootless containers utilizing PR_SET_NO_NEW_PRIVS, strict cgroups, and read-only root filesystems.[5] Enforce default-deny network egress and short-lived Git credentials.

Phase C — Stronger Multi-Project Isolation (Target: Version 0.8)

- *Objective:* Multi-project capabilities for internal teams.
- *Implementation:* Migrate from iptables to NFTables for high-performance egress proxies.[20] Deploy the Credential Broker / Tool Proxy pattern.[15] Integrate Policy-as-Code (Cedar).[12]

Phase D — Untrusted Execution (Target: Version 1.2)

- *Objective:* Safe execution of unknown, third-party code.
- *Implementation:* Deploy hardware-backed microVMs (e.g., Firecracker) across dedicated bare-metal workers.[5] Implement pre-warmed snapshotting and cryptographic teardown verification.

Phase E — Mature Multi-Tenant Operation (Target: Version 2.0+)

- *Objective:* Enterprise capacity and compliance.
- *Implementation:* Regional execution pools for data residency, independent customer encryption keys, and formal automated incident response workflows.

Crucial Early Decision: The generic provider abstractions and JWT-SVID identity schemas must be explicitly finalized during Phase A, ensuring the microVM migration in Phase D requires zero refactoring of the orchestration control plane.

Architecture Decision Records

ADR 001: Runtime-Provider Abstraction

- *Context:* Hard-coding execution logic to Docker limits future isolation.
- *Decision:* The control plane will invoke environments via a generic API interface requesting capabilities.
- *Rationale:* Prevents vendor lock-in; enables dynamic switching between containers and microVMs based on the execution profile.
- *Consequences:* Requires developing custom adapters for each underlying hypervisor.

ADR 002: WIMSE and SPIFFE for Workload Identity

- *Context:* Agents require authentication to cloud APIs and internal services.
- *Decision:* All task identities will use SPIFFE JWT-SVIDs; static API keys are strictly forbidden.
- *Rationale:* Eliminates long-lived secrets, allows cryptographic attestation of workloads, and supports short expiry windows.[8]
- *Consequences:* Requires deploying and maintaining a robust SPIRE server infrastructure.

ADR 003: Egress Proxy Utilizing NFTables and DNS Snooping

- *Context:* Unrestricted egress enables data exfiltration. Traditional iptables scales poorly.
- *Decision:* Network control will intercept DNS requests, mapping allowed domains to TTL-bound NFTables verdict maps.[5]
- *Rationale:* Achieves O(1) processing latency and completely prevents hardcoded IP

bypasses.
- *Consequences:* Requires custom DNS middleware integration and CoreDNS plugin management.

ADR 004: Tool Proxy Pattern over In-Sandbox Credentials

- *Context:* Injecting high-privilege secrets into sandbox memory risks exposure via prompt injection or log dumps.
- *Decision:* High-privilege actions must be routed through an external Tool Proxy.[15]
- *Rationale:* Physically prevents sensitive secrets from entering the untrusted execution plane.[5]
- *Consequences:* Increases implementation effort for proxying complex APIs.

ADR 005: Prevention of Git Workspace Poisoning

- *Context:* Attackers use repo config to manipulate Git hooks and submodules for sandbox escape (CVE-2026-26268, CVE-2025-48384).[3]
- *Decision:* Globally nullify core.hooksPath in the base image and disable recursive submodule cloning for untrusted inputs.
- *Rationale:* Closes documented critical vulnerabilities associated with automated repository checkouts.
- *Consequences:* Prevents agents from legitimately utilizing internal Git hooks during tasks.

Ordered Engineering Backlog

Category	Task	Priority	Description
Architecture	Finalise Sandbox Interface	1	Define JSON schemas for create, attest, start, destroy endpoints.
Identity	Deploy SPIRE Server	1	Provision SPIRE control plane and establish JWT-SVID minting logic.
Networking	CoreDNS Rebinding Config	2	Configure CoreDNS to strip RFC1918 and 169.254.0.0/16 responses.[23]
Networking	NFTables Egress	2	Build dynamic DNS

	Proxy		snooping daemon to generate TTL-bound verdict maps.[5]
Security	Git Hook Neutralisation	2	Bake git config --global core.hooksPath /dev/null into all base images.[3]
Runtime	Rootless Container Baseline	3	Implement cgroups v2 limits and PR_SET_NO_NEW_PRIVS restrictions.[5]
Secrets	Tool Proxy Implementation	3	Develop external proxy to broker high-privilege cloud API calls.[15]
Evidence	Synchronous Log Streaming	4	Establish write-only local sockets for immutable forensic capture.
Teardown	Orphan Reconciliation Loop	4	Build out-of-band daemon to sweep worker nodes for unmapped processes.
Testing	Adversarial Fitness Suite	5	Develop automated tests for fork bombs, capability dropping, and metadata access.

Final Recommendations

Recommended Initial Architecture: Commence immediately with a cleanly decoupled topology. Deploy the control plane on managed Kubernetes infrastructure, while execution

workers operate on physically isolated bare-metal instances. Utilize hardened rootless containers strictly for trusted internal workloads, governed rigidly by default-deny NFTables egress proxies and SPIFFE workload identity. This satisfies immediate operational needs without compromising future migration paths.

Recommended Target Architecture: Evolve the execution plane toward hardware-backed microVMs. This target state supports fully untrusted, multi-tenant repository execution by enforcing MMU-level hardware isolation, utilizing pre-warmed snapshotting to guarantee millisecond latency, and mandating out-of-sandbox credential brokering for all sensitive actions.

Minimum Safe Baseline: Real code execution must not commence without the following controls:

1. Hardware/Kernel capability reduction (enforcing PR_SET_NO_NEW_PRIVS).
2. Default-deny network egress combined with DNS rebinding protection (stripping RFC1918).
3. Deterministic nullification of Git core.hooksPath.
4. Absolute prohibition of persistent secrets injected into environment variables.

Migration Trigger: The transition from hardened containers to microVM isolation is non-negotiable and must be triggered the moment the platform accepts its first task involving unvetted third-party dependencies, unmodified community repositories, or arbitrary code downloaded from the internet. At this threshold, shared-kernel architectures introduce critical, unacceptable enterprise risk.

Research Gaps and Unresolved Decisions

- **MicroVM Orchestration Fragmentation:** While the Firecracker hypervisor provides robust underlying virtualisation, a universally adopted, open-source orchestration layer for ephemeral agent microVMs remains fragmented. Integrating complex microVM lifecycles into the generic sandbox abstraction may necessitate bespoke daemon development.
- **Memory Zeroisation Efficacy:** Validating the deterministic scrubbing of memory within high-level, garbage-collected runtime interpreters (e.g., Node.js, Python) inside the sandbox is mathematically challenging, as these languages do not guarantee memory wiping before reallocation.[16] Reliance must remain heavily on the Tool Proxy pattern to prevent secrets from ever entering the sandbox.
- **Cross-Cloud Attestation Parity:** Normalising node attestation via SPIRE across diverse cloud environments (AWS, GCP, Azure) requires extensive configuration to ensure workload identity claims remain unified and interoperable across federated boundaries.

Final Synthesis

What is the leanest secure execution architecture that can support trusted internal agent workloads now, preserve a clean migration path to gVisor or microVM isolation, and eventually support untrusted multi-project execution without redesigning workload identity, credential issuance, network policy, evidence capture, or sandbox lifecycle

management?
The optimal architecture establishes an ironclad, generic abstraction layer between the orchestration control plane and the execution workers from day one. By deploying a SPIFFE/WIMSE identity model, an NFTables-backed default-deny network proxy, and an out-of-sandbox Tool Proxy for credential management, the platform firmly centralizes security within the trusted control plane. This enables the execution layer to begin with lean, hardened rootless containers for internal trusted tasks. Because the core abstractions remain agnostic to the underlying runtime, the platform can seamlessly migrate to hardware-isolated microVMs for untrusted multi-tenant workloads in later milestones—achieving maximum security scaling without requiring any structural redesign of the identity, policy, or credential architectures.

Works cited

1. What Is an Agent Execution Sandbox? | Augment Code, accessed July 27, 2026, https://www.augmentcode.com/guides/agent-execution-sandbox
2. A Deterministic Control Plane for LLM Coding Agents - arXiv, accessed July 27, 2026, https://arxiv.org/html/2606.26924v1
3. CVE-2026-26268: Anysphere Cursor Sandbox Escape RCE Flaw - SentinelOne, accessed July 27, 2026, https://www.sentinelone.com/vulnerability-database/cve-2026-26268/
4. CVE-2025-48384 - Red Hat Customer Portal, accessed July 27, 2026, https://access.redhat.com/security/cve/cve-2025-48384
5. Isolate Coding Agents with MicroVMs | Stacklok, accessed July 27, 2026, https://stacklok.com/blog/your-ai-agent-doesnt-deserve-root-access/
6. Announcing Lambda MicroVMs: serverless compute environments with VM-level isolation and near-instant startup - AWS, accessed July 27, 2026, https://aws.amazon.com/blogs/compute/announcing-lambda-microvms-serverless-compute-environments-with-vm-level-isolation-and-near-instant-startup/
7. What are SPIFFE and SPIRE? - Red Hat, accessed July 27, 2026, https://www.redhat.com/en/topics/security/spiffe-and-spire
8. AI Agent Authentication and Authorization - IETF, accessed July 27, 2026, https://www.ietf.org/archive/id/draft-klrc-aiagent-auth-00.html
9. SPIFFE Concepts, accessed July 27, 2026, https://spiffe.io/docs/latest/spiffe/concepts/
10. Configuring workload identity federation for SPIFFE | OpenAI API, accessed July 27, 2026, https://developers.openai.com/api/docs/guides/workload-identity-federation/spiffe
11. JWT-SVID - SPIFFE, accessed July 27, 2026, https://spiffe.io/docs/latest/spiffe-specs/jwt-svid/
12. Establishing Workload Identity for Zero Trust CI/CD: From Secrets to SPIFFE-Based Authentication - arXiv, accessed July 27, 2026, https://arxiv.org/pdf/2504.14760
13. (PDF) Delegation Without Escalation - ResearchGate, accessed July 27, 2026, https://www.researchgate.net/publication/410729828_Delegation_Without_Escala

tion
14. Configuring Workload Identity and AWS OIDC Federation - Teleport, accessed July 27, 2026, https://goteleport.com/docs/machine-workload-identity/workload-identity/aws-oidc-federation/
15. red-team | jerry - ClaudePluginHub, accessed July 27, 2026, https://www.claudepluginhub.com/skills/geekatron-jerry/red-team
16. Analysis of Encryption Key Zeroization from System-wide Perspective, accessed July 27, 2026, https://lab.iisec.ac.jp/~suzaki_lab/PDF/ACSAC2025-Sawa.pdf
17. Git - HackTricks, accessed July 27, 2026, https://hacktricks.wiki/en/network-services-pentesting/pentesting-web/git.html
18. CVE-2025-65964: Weaponizing Git Hooks in n8n for Critical RCE in AI Infrastructure, accessed July 27, 2026, https://www.penligent.ai/hackinglabs/cve-2025-65964-weaponizing-git-hooks-in-n8n-for-critical-rce-in-ai-infrastructure/
19. Git Security Vulnerability CVE-2025-48384 - GitLab Support, accessed July 27, 2026, https://support.gitlab.com/hc/en-us/articles/21285978227228-Git-Security-Vulnerability-CVE-2025-48384
20. NFTables mode for kube-proxy - Kubernetes, accessed July 27, 2026, https://kubernetes.io/blog/2025/02/28/nftables-kube-proxy/
21. Daikin Emura Series – Arbitrary Remote Control via DNS Rebinding - Hacktive Security, accessed July 27, 2026, https://www.hacktivesecurity.com/blog/2018/12/14/daikin-emura-series-arbitrary-remote-control-via-dns-rebinding/
22. Isolate AI code execution with Agent Sandbox | GKE AI/ML - Google Cloud Documentation, accessed July 27, 2026, https://docs.cloud.google.com/kubernetes-engine/docs/how-to/agent-sandbox
23. DNS Rebinding Protections | pfSense Documentation, accessed July 27, 2026, https://docs.netgate.com/pfsense/en/latest/services/dns/rebinding.html
24. GitHub - spr-networks/coredns-rebinding_protection, accessed July 27, 2026, https://github.com/spr-networks/coredns-rebinding_protection
25. Running and using MicroVMs - AWS Lambda, accessed July 27, 2026, https://docs.aws.amazon.com/lambda/latest/dg/microvms-launching.html