Project Onboarding and Extension Architecture for a Provider-Neutral Agent Platform

Executive summary

The modern agent platform operates within highly dynamic, varied computing environments where maintaining stringent security, auditability, and isolation is paramount. As platforms scale to support complex workflows and third-party integrations, the accumulation of custom integration code within the control plane introduces severe risks, including unmanageable dependency matrices, heightened attack surfaces, and control-plane destabilisation. This exhaustive technical report outlines an implementation-focused architecture for safely onboarding new projects and integrating third-party extensions without requiring modifications to core platform logic. By enforcing strict boundary definitions, zero-trust workload identities, and verifiable supply-chain attestations, the architecture ensures that extensibility does not compromise operational integrity. The core philosophy dictates that the control plane provides stable primitives and governance, while extensions provide specialised, sandboxed capabilities communicating exclusively via structured remote procedure calls.

Recommended direction

The recommended direction mandates a declarative, capability-driven extension model operating entirely out-of-process. The platform must treat all extensions as untrusted by default, sandboxing execution within isolated bytecode runtimes or rootless containers.[1] Communication between the execution plane and the control plane must strictly occur over an authenticated, stateless remote procedure call protocol using structured data payloads.[2] By leveraging standardised artifact registry specifications for distribution, cryptographic provenance attestations for supply-chain security, and workload identity documents for dynamic credentialing, the architecture achieves a highly decoupled, secure, and governable ecosystem.[4]

Scope and assumptions

This architecture assumes a multi-tenant, multi-project environment where the control plane is the absolute authority over policy, authorisation, identity, and task orchestration. It is assumed that extensions are ephemeral, stateless entities that rely entirely on stable core APIs for persistence, and that they may be potentially hostile, vulnerable, or poorly maintained. The scope of this document encompasses the complete lifecycle management of all extension types, project onboarding frameworks, isolation mechanisms, protocol designs, and supply-chain validations. The architecture explicitly avoids the immediate development of a fully

general public marketplace, focusing first on rigorous internal governance and verified third-party integrations to establish a proven baseline of trust.

Extension taxonomy

To govern extensions safely, they must be rigorously classified into a formal taxonomy. This classification dictates the expected trust level, installation boundaries, allowed APIs, and lifecycle rules for each integration.

1. **Project adapters:** These extensions connect a project to specific repositories, issue trackers, or deployment systems. They require network egress rights but are strictly bound to a single project's execution namespace.
2. **Tool plugins:** These expose bounded, discrete operations, such as file transformation, browser research, or data extraction. They execute in tightly constrained sandboxes and are classified by side-effect severity, ranging from read-only operations to irreversible external actions.
3. **Provider adapters:** These normalise interactions with external language models, embedding services, or search providers. They map generic capability requests to provider-specific payloads without reducing the requested capability to a lowest common denominator.
4. **Coding runtimes:** These define the programming language environments, compilers, and dependencies required to execute generated code. They dictate the execution boundaries and operating-system dependencies for dynamic scripts.
5. **Sandbox and execution adapters:** These broker connections to underlying rootless containers, micro-virtual machines, or ephemeral remote cloud build systems, ensuring that workloads are isolated from the host.[1]
6. **Workflow extensions:** These define state machines, retries, approvals, and compensations for durable executions. They must execute deterministically, recording all non-deterministic external calls to ensure replay safety during version upgrades.[7]
7. **Fleet templates:** These constitute reusable definitions of agent roles, routing rules, memory access parameters, and escalation pathways applied across an entire task force.
8. **Agent-role templates:** These define individual, reusable agent responsibilities and capability bounds without instantiating an entire fleet.
9. **Memory extensions:** These connect the platform to external vector stores, document indexes, or episodic databases, dictating how project ledgers are updated and how semantic knowledge is retrieved.
10. **CI and delivery integrations:** These broker interactions with continuous integration pipelines, requiring highly sensitive dynamic credentials to start jobs, approve deployment gates, and promote environments.
11. **Environment-image extensions:** These provide approved, immutable root filesystems for specific tools or coding runtimes, rigorously verified via cryptographic digests and vulnerability scans.
12. **Policy extensions:** These inject organisation-specific or project-specific compliance rules evaluated securely by the control plane's central policy engine.

13. **Observability extensions:** These export telemetry, traces, audit evidence, and cost records to external organisational logging systems without exposing raw control-plane memory.

Project-onboarding model

The standard project-onboarding process is designed to translate human-readable requirements into a reproducible, immutable project definition. When onboarding a new project, maintainers must declare a comprehensive set of parameters, including the project identity, business purpose, connected repositories, default branches, and protected paths. Furthermore, the onboarding model requires explicit declarations of allowed and prohibited providers, data sensitivity classifications, regulatory constraints, and required memory namespaces. The onboarding process enforces default-deny network rules and zero-trust data access. Instead of accumulating manual dashboard toggles that lead to configuration drift, the system compiles these declarations into a canonical, declarative configuration tree that is versioned, auditable, and statically validated prior to project activation.

Project-onboarding SDK

The project-onboarding SDK empowers maintainers to construct, validate, and dry-run project configurations locally before committing them to the control plane. Designed to support command-line interfaces, API integrations, and declarative manifests, the SDK operates via a language-neutral protocol, avoiding the exposure of unstable control-plane libraries to extension authors. The SDK provides local scaffolding capabilities, automated validation of configuration schemas, and sophisticated policy simulation. Crucially, its dry-run mode evaluates dependency resolution, capability matching, and security policy compliance—verifying secret references and simulating network restrictions—without issuing active credentials, mutating the production project registry, or activating durable workflows. This ensures that a proposed project package containing repository definitions, workflow packages, and CI integrations is structurally sound and secure prior to deployment.

Core-versus-extension boundary

The fundamental architectural principle of this system is that the control plane owns stable governance, while extensions own specialised, volatile capabilities. The core platform strictly controls task identity, project identity, policy decisions, authorisation, workload identity issuance, sandbox lifecycles, and audit events. Conversely, extensions are relegated to implementing provider-specific API calls, executing local tool operations, formatting data, and preparing environments. Extensions are explicitly prohibited from mutating task state directly, disabling audit trails, approving their own capability scopes, lowering their risk classification, or interacting directly with core databases. Any operation that alters canonical authoritative state must traverse stable, rate-limited, and audited APIs exposed by the control plane. This ensures that extensions never evolve into unmanageable alternative control planes.

Extension architecture models

Evaluating architectural models for extension execution requires balancing isolation, latency, portability, and upgrade safety.

Host Model	Isolation Level	Latency	Deployment Complexity	Memory Safety	Best Suited For
In-process Plugins	None	Ultra-low	Low	Very Low (crash risks)	Strictly prohibited
Subprocess Plugins	Low	Low	Medium	Low	Legacy local transitions
Containerised	Medium	Medium	Medium	Medium	Complex provider adapters
Portable Bytecode	High	Low	Low	High	Tool plugins, policy logic
MicroVMs	Very High	High	High	High	Untrusted coding runtimes
Remote Services	Boundary	High	High	High	External managed services

The architecture strictly prohibits in-process plugins due to their capacity to induce control-plane panics and memory corruption. For highly trusted internal logic and tool plugins, portable bytecode modules (such as WebAssembly) provide optimal low-latency execution with strict, capability-based sandboxing.[1] For arbitrary coding runtimes and unverified third-party extensions, hardware-enforced micro-virtual machines or strictly bound rootless containers are mandated to prevent privilege escalation and secure the host environment.

Extension manifest

The extension manifest is a machine-readable document acting as the ultimate, statically verifiable contract between the extension and the control plane. It defines every required capability, network destination, and configuration schema prior to execution.

JSON

```
{
  "manifestVersion": "1.0",
  "extensionId": "generic-coding-runtime",
  "publisher": "organisation-core",
  "version": "2.4.1",
  "trustTier": "organisation-maintained",
  "capabilitiesProvided": ["runtime.execute.python"],
  "capabilitiesRequired": ["secrets.request-dynamic", "network.approved-api"],
  "permissionsRequested": [
    { "scope": "project", "action": "memory.read-project", "risk": "medium" }
  ],
  "networkDestinations": ["api.sample-provider.com"],
  "evidenceContract": { "emits": ["cost", "latency", "side-effect"] },
  "compatibility": { "minimumPlatformVersion": "1.2.0" }
}
```

This manifest prevents hidden dependencies and undocumented runtime behaviours. Any capability, filesystem access, or network destination not explicitly declared within this document is cryptographically blocked at runtime by the extension host's admission controller.

Capability and permission model

The platform employs a granular, capability-based permission system that rejects broad, opaque roles such as full-administrator. Extensions must request exact, finely scoped capabilities (e.g., repository.write-task-branch, model.invoke-text, secrets.request-dynamic). Each capability explicitly defines a resource scope (e.g., restricted to a specific repository), an action scope, a duration limit, and a rigorous risk classification. Access requests must map strictly to these capabilities, ensuring least privilege. If an extension attempts an action beyond its static capability grant, the execution is terminated, and a policy violation event is logged.

Trust tiers and isolation

Trust tiers govern the operational boundaries, monitoring intensity, and isolation requirements of every extension within the ecosystem.

Trust Tier	Allowed Host Model	Network Restrictions	Required Reviews	Incident Response
Core-maintained	Portable Bytecode	Platform defaults	Automated testing	Direct patching
Organisation-maintained	Rootless Containers	Org-internal networks	Internal peer review	Automatic rollback
Verified third party	Isolated Containers	Whitelisted external	Independent security	24hr remediation
Unverified third party	Ephemeral MicroVMs	Default-deny entirely	Explicit per-project	Immediate revocation
Quarantined	Suspended state	Fully disabled	N/A	Forensic preservation

A trust label is merely semantic and must never substitute for technical isolation. Even a core-maintained extension must execute within a resource-constrained, isolated environment to prevent cascading failures.

Extension-host architecture

The extension host operates as a hardened environment situated within the execution plane, physically separated from the authoritative databases of the control plane. When an extension is activated, the host provisions an authenticated remote procedure call (RPC) channel, assigns a unique workload identity document to the specific extension process, and enforces memory, CPU, and execution timeouts.[4] The host transparently intercepts all outbound network requests, validating them in real-time against the manifest's permitted destinations. Additionally, the host continuously monitors process health, handles execution retries, and captures structured standard output to fulfill the audit evidence contract without requiring the extension to implement complex logging libraries.

RPC and protocol design

Communication between the control plane and extensions relies on a stateless, language-neutral remote procedure call protocol utilising structured JSON payloads. This protocol standardises interactions, directly solving the complexity of integrating diverse tools and model providers without bespoke adapter code.[10] Every RPC request contains an id, a method, and versioned params, supporting schema evolution and asynchronous long-running

operations.[2] The control plane acts as the server, while the extension acts as the client for state mutations. For tool invocation, the control plane sends a request to the extension host, which returns either a structured result payload or a standardized error payload containing retryability flags, operator-facing explanations, and execution metadata.[2]

Registry and package formats

To ensure reproducibility and security, extensions must be distributed via standardised artifact formats rather than mutable source packages.

Package Format	Reproducibility	Portability	Isolation	Target Extension Type
Source Package	Low	Low	None	Unmanaged scripts
Language Package	Medium	Medium	Low	Legacy libraries
Container Image	High	Medium	Medium	Runtimes, Adapters
Artifact Manifest	High	High	High	Workflow dependencies
Portable Bytecode	High	Very High	High	Tool plugins, Policy

The platform relies on Open Container Initiative (OCI) artifact registry protocols to store extensions. Using the OCI artifact layout allows the platform to store portable bytecode modules, configuration packages, and container images under a unified, content-addressed system characterised by specific media types (e.g., application/vnd.wasm.config.v0+json).[9] A private, internal registry must be mandated to prevent dependency confusion vulnerabilities, ensuring the package manager does not silently fall back to a public repository containing a malicious package with a higher version number.[12]

Software-supply-chain security

Supply-chain integrity is a non-negotiable prerequisite for extension execution. Every extension version must be accompanied by a cryptographic provenance attestation—verifying the builder

identity, source repository, and execution pipeline—and a Software Bill of Materials (SBOM) detailing all transitive dependencies.[6] The platform's admission controller rejects any extension lacking a verified signature matching an approved publisher identity. By cryptographically verifying these attestations against an immutable transparency log before installation, the platform actively prevents the ingestion of compromised artifacts, fulfilling the strict requirements of modern supply-chain levels for software artifacts.[6]

Publisher identity and signing

Publishers are identified using keyless signing infrastructure tied to short-lived identity certificates, eliminating the catastrophic risks associated with compromised, long-lived private keys. When an automated build system compiles an extension, it authenticates via an OpenID Connect token, receives a short-lived signing certificate binding the cryptographic signature to the specific pipeline, signs the artifact and its SBOM, and records the signature in an append-only transparency log.[14] The operator interface clearly distinguishes the verified identity of the package signer, exposing whether an extension was produced by an approved organisational pipeline or an unverified developer workstation.

Certification model

The certification framework enforces progressive validation levels before an extension can operate in sensitive environments. Certification is strictly bound to a single, immutable version digest; a certified previous version does not confer automatic trust to a subsequent release.

1. **Syntactically Valid:** The manifest parses correctly, and schema definitions are well-formed. Allowed only for local dry-run testing.
2. **Compatible:** Protocol versions align with the target control plane without unresolvable dependency conflicts.
3. **Functionally Tested:** The extension successfully executes against the automated conformance suite.
4. **Security Reviewed:** The SBOM is free of known vulnerabilities, and permissions request the absolute minimum required scope.
5. **Production Approved:** The extension is permitted for general organisational use.
6. **Sensitive-Data Approved:** Network isolation is fully audited to ensure zero data exfiltration risk, permitting use in highly regulated projects.

Conformance and contract testing

All extensions must successfully navigate a mandatory automated conformance suite prior to certification. This suite enforces rigorous consumer-driven and provider-driven contract testing. Tests validate manifest accuracy, timeout handling, payload size limits, and idempotency guarantees. Consumer-driven contract tests ensure that control-plane updates do not break supported extension APIs by replaying golden requests and asserting expected responses. Provider-driven tests verify that a new extension version adheres to the behavioral boundaries set by the core. Adversarial tests within the suite simulate malicious activity, such as

output-schema violations, excessive memory consumption, and recursive hook loops, verifying the extension host's ability to contain failures gracefully.

Compatibility and dependency management

Compatibility is evaluated through a multi-dimensional matrix encompassing the platform version, protocol schema, dependent workflow versions, and environment images. The system distinguishes strictly between syntactic compatibility (schema matching) and semantic compatibility. A change in default behaviour, an altered cost model, or a shift in a provider's data residency guarantee constitutes a semantic incompatibility, even if the JSON schema remains identical. When extensions share transitive dependencies, the platform resolves conflicts by enforcing strict per-project or per-workflow isolation, allowing multiple side-by-side extension versions to run in distinct namespaces rather than forcing a global convergence that risks breaking durable workflows.

Repository onboarding

When a project connects to a repository, the onboarding process rigorously extracts project identity, default branches, and protected paths. Crucially, the platform treats all repository-provided instructions—such as development container scripts, CI configurations, and package-manager hooks—as inherently untrusted. Repository policy cannot override organisation or platform-level security policies. During onboarding, the repository contents are scanned for exposed secrets and vulnerable dependencies before being mounted as read-only volumes within a sandboxed execution environment, ensuring malicious code cannot persist or alter the control plane.

Tool onboarding

Tools are onboarded by declaring structured input and output schemas, expected side effects, and exact idempotency guarantees. Tools are classified by side-effect severity, ranging from read-only data extraction to irreversible external actions. For high-risk operations involving secret-sensitive access or database mutations, the platform utilises a brokered proxy pattern. Instead of granting the tool extension direct access to underlying credentials, the extension requests a defined operation via a secure API, and a trusted proxy performs the mutation, logging the action and shielding the credential from the extension’s potentially vulnerable memory space.

Provider-adapter onboarding

Provider adapters are onboarded to abstract underlying vendor APIs for language models and embedding services without sacrificing specialised features. The adapter manifest maps core platform capabilities—such as token streaming, multimodal inputs, and structured output formatting—to provider-specific endpoints. To prevent vendor lock-in, the control plane routes tasks based on required capabilities, usage costs, and geographic data-residency policies rather than hard-coded provider identifiers. Adapters must accurately report token usage and

rate limits back to the core through standard observability contracts, ensuring transparent cost governance.

Coding-runtime and environment-image onboarding

Coding runtimes are onboarded as immutable environment images. The extension manifest declares supported processor architectures, base image digests, package managers, and isolated network requirements. To achieve certification, environment images must demonstrate non-root execution capabilities, the complete absence of embedded secrets, and strict adherence to resource limits. Vulnerability scanners continuously evaluate these images against threat intelligence databases. Project-supplied images are treated as potentially hostile and are subjected to rigorous malware scanning and sandbox confinement before execution is permitted.

Workflow and fleet-template onboarding

Durable workflows and fleet templates dictate long-running orchestration logic and multi-agent coordination. Workflows are onboarded as versioned packages declaring discrete activities, state transitions, retry policies, and compensations. Because these workflows can run for months, their code must be strictly deterministic. Any non-deterministic behaviour, such as random number generation or network calls, must be isolated into recorded activities to prevent replay failures during infrastructure restarts or upgrades.[7] Fleet templates instantiate governed agent roles with explicit bounds on concurrency, budget, and parallel task generation, preventing unbounded recursive execution loops.

Memory namespace and adapter governance

Projects require highly isolated memory namespaces to manage episodic history, semantic knowledge, and project ledgers. Memory adapters dictate how this data is indexed, embedded, and retrieved. Governance rules enforce cryptographic segregation of project memory; an extension acting on behalf of example-project is technologically prevented from querying the vector space or knowledge graph of another project. Memory adapters must pass conformance tests proving deterministic retrieval, proper metadata attribution, strict permission filtering, and reliable deletion capabilities to satisfy regulatory data-retention mandates.

CI and delivery integrations

CI adapters act as secure bridges to external continuous integration and delivery systems. Because these operations involve high-risk deployment actions, CI adapters are prohibited from holding persistent, broad-access administrative tokens. Instead, they utilise workload identity federation to request short-lived, finely scoped dynamic tokens capable only of triggering specific pipeline jobs, reading test artifacts, or approving exact deployment gates. Untrusted pull-request builds are strictly confined to network-isolated sandboxes, preventing the exfiltration of environment variables or proprietary source code during automated testing.

Configuration and policy overlays

Configuration precedence resolves hierarchically from platform defaults, up through organisation defaults, project configurations, workflow settings, and finally task-level overrides. A paramount security invariant dictates that security ceilings—such as maximum budgetary limits, denied network regions, or required approval gates—cannot be weakened by a lower-precedence configuration. If an organisation policy mandates local data residency, an extension configuration attempting to route telemetry to an external global region is statically rejected during the dry-run validation phase, generating a clear conflict resolution log.

Installation and staged activation

Extension installation is executed as a distinct, fully auditable cryptographic event. Prior to installation, the control plane generates a detailed installation plan outlining the package digest, resulting capability grants, network egress modifications, and estimated cost impacts. Staged activation follows a progressive rollout sequence: the package is registered and validated, installed in an inactive state, executed against the conformance suite, promoted to a contained sandbox project, and eventually activated globally. Dry-run installation modes allow operators to simulate permission expansions and migration plans safely without generating permanent resources or risking production stability.

Upgrade, canary, and rollback

Upgrading extensions requires precise, deterministic lifecycle management to preserve durable workflows.

Upgrade Strategy	Description	Impact Profile	Target Scenario
Replace in place	Overwrites the active version globally.	High	Minor stateless configurations.
Side-by-side	Both versions run concurrently in isolated spaces.	Low	Durable workflows requiring old APIs.
Canary	Routes fractional traffic to the new version.	Low	High-risk provider adapters.
Project-pinned	Explicit version tied directly to project	Zero	Strictly regulated environments.

	manifest.		
Forced Security	Immediate overwrite ignoring pinned dependencies.	Disruptive	Critical vulnerability mitigation.

Upgrades must not implicitly expand an extension’s permissions; any request for new capabilities requires explicit human review. Side-by-side versioning is critical for durable workflows that remain pinned to older adapter versions to maintain determinism. Data migrations accompanying an upgrade must declare checkpoint strategies and forward-recovery mechanisms, executing via restricted APIs rather than arbitrary database schema alterations. Rollback procedures restore previous protocol states and routing rules without silently reversing completed, valid external user actions.

Health and behavioural monitoring

Extensions must expose standard health endpoints, yet a simple process heartbeat is entirely insufficient for security governance. Behavioural monitoring continuously analyses runtime execution for anomalies, tracking signals such as unexpected process spawning, unexplained latency spikes, output schema violations, and abnormal credential requests. The control plane measures these signals against historical operational baselines. If an extension begins attempting cross-project memory access or requests undeclared external network destinations, behavioural monitoring systems instantly downgrade the extension's trust tier, triggering automatic throttling, suspension, or full quarantine protocols.

Vulnerability response

When a new software vulnerability is disclosed affecting an extension's dependencies, the platform's reconciliation service utilises the extension's SBOM to instantly identify all impacted project installations. A rigorous severity assessment dictates the automated response: low-risk vulnerabilities issue operator warnings and deprecation notices, while critical, exploitable vulnerabilities trigger immediate, automated quarantine. A patched release addressing the vulnerability must undergo full recertification, proving the issue is resolved via updated cryptographic attestations and vulnerability scans before it is permitted to replace the quarantined version.

Quarantine and emergency revocation

Quarantine semantics strictly freeze an extension's execution state without destroying forensic evidence. The control plane ceases routing new tasks to the quarantined extension, actively revokes its workload identity credentials, and severs its network access, while preserving memory states and logs for investigation. Emergency revocation serves as a broader kill

mechanism, globally invalidating all versions of a compromised publisher's extensions, dropping active tasks, and alerting all affected project owners. Workflow engines detect the revocation and automatically initiate compensation logic or degrade gracefully to fallback providers to maintain core service availability.

Deprecation and removal

Extensions undergo a formal, staged deprecation lifecycle before removal. The sequence involves marking the extension as deprecated, explaining the rationale, identifying all affected projects, providing migration tooling, and establishing a firm end-of-support date. Following this window, new tasks are blocked. Uninstalling an extension is not a simple deletion; it requires verifying that all temporary credentials, dynamic network rules, and ephemeral caches are demonstrably destroyed. Crucially, the removal process explicitly preserves historical audit evidence, task ledgers, and operational logs. An extension cannot unilaterally erase its historical footprint upon removal.

Project decommissioning

Decommissioning a project executes a comprehensive cryptographic destruction sequence. The platform archives project memory namespaces according to legal retention policies, revokes all repository access tokens, shreds dynamic secrets, and terminates all related active fleet templates. The system generates a final, verifiable decommissioning record detailing the exact deletion of artifacts, the termination of CI integrations, and the retention of compliance-mandated audit logs. This exhaustive procedure ensures no orphaned resources, latent access permissions, or unmonitored backdoors remain within the platform infrastructure following project closure.

Operator experience

The operator console must abstract the complexity of cryptographic attestations and JSON manifests into actionable, plain-language operational consequences. Instead of displaying a raw technical identifier like repository.write-task-branch, the console provides explicit context: "This extension may create or modify files on task-specific branches in the selected repositories, but cannot merge into protected branches without separate approval." Operators are provided with a visual dependency graph, allowing them to trace which projects depend on specific extension versions and instantly assess the blast radius of a proposed deprecation, upgrade, or security quarantine, drastically lowering the extension-specific operational burden.

Data, secrets, network, and cost governance

Robust governance frameworks constrain extension behavior across four critical dimensions:

- **Data Residency:** Extensions must declare regional processing capabilities. The control plane enforces these constraints via geographic routing tags, ensuring compliance with sovereign cloud and data localization regulations.[16]
- **Secrets Governance:** Extensions never possess static, embedded secrets. They request

a dynamic, short-lived lease from a central secrets broker via their workload identity document, ensuring access expires automatically.
- **Network Governance:** The platform enforces a default-deny network egress policy. All expected external destinations must be statically declared in the manifest and approved at installation; any deviation results in packet drops and immediate behavioral alerts.
- **Cost Governance:** Extensions emit structured cost evidence per invocation (e.g., £0.05 per task). The control plane aggregates this telemetry against project-level budgets, automatically terminating task execution if financial ceilings are breached.

Reference architecture

The reference architecture enforces a strict physical and logical separation between the authoritative control plane and the volatile execution plane.

Code snippet
sequenceDiagram
participant Scheduler as Control Plane (Scheduler)
participant Policy as Policy Engine
participant Broker as Credential Broker
participant ExtHost as Execution Plane (Extension Host)
participant Provider as External Provider

Scheduler->>Policy: Request capability: model.invoke-text
Policy-->>Scheduler: Authorised. Route to: sample-provider-adapter
Scheduler->>ExtHost: JSON-RPC Call (method, params, id)
ExtHost->>Broker: Request dynamic token (via Workload Identity)
Broker-->>ExtHost: Issue short-lived credential
ExtHost->>Provider: Invoke API with token
Provider-->>ExtHost: Return payload
ExtHost-->>Scheduler: JSON-RPC Result (structured output, cost evidence)

Capability-Resolution Flow:

1. A task requests a specific capability (e.g., model.invoke-text).
2. The Capability Directory identifies installed, certified extensions providing this capability.
3. The Policy Engine filters this list by project residency rules, available cost budgets, and trust tiers.
4. The Protocol Gateway routes a structured JSON-RPC payload to the selected extension host.
5. The Extension Host requests a dynamic token from the Credential Broker using its workload identity.[4]
6. The extension executes the logic, returning a structured result and emitting audit evidence

back to the control plane.

Schemas and safety invariants

Installation Record Schema: To guarantee reproducible installations, the platform generates an immutable record for every activated extension.

JSON

```
{
  "recordId": "inst-8f92a",
  "extensionId": "sample-provider-adapter",
  "version": "1.1.0",
  "packageDigest": "sha256:d3b...8f",
  "projectScope": "example-project",
  "grantedCapabilities": ["network.approved-api"],
  "networkEgress": ["api.sample-provider.com"],
  "installationTime": "2026-07-27T04:06:00Z"
}
```

Safety Invariants:

- Extensions cannot access core databases directly by default.
- Extensions cannot approve their own permissions or lower their trust tier.
- Upgrades cannot silently expand permissions without explicit human review.
- Network access is strictly default-deny.
- Uninstallations must not erase audit evidence or historical ledgers.
- Quarantined extensions receive no new tasks and lose all credential access immediately.
- Extension-created external side effects remain idempotent or are continuously reconciled.

Adversarial scenarios

To prove the resilience of the architecture, it must withstand sophisticated adversarial attacks.

Scenario	Prevention Mechanism	Detection Signal	Containment & Recovery
Extension attempts to read another project's memory	Capability boundary scoping tied to unique memory namespaces.	Policy engine explicitly blocks the RPC request.	Request dropped; behavioral alert generated; no residual risk.
Provider adapter	Hardware-enforced	Firewall rule	Packet dropped;

sends data to undeclared network	network namespace egress filtering.	violation triggered at host level.	host quarantined; forensic evidence captured.
Compromised publisher signs malicious update	Keyless signature threshold policies and transparency log checks.	Transparency log mismatch during admission control.	Signature rejected during staging; installation aborted entirely.
Namespace/Dependency confusion attack	Internal registry maps strictly to private namespaces; public fallback prohibited.	Package resolution mismatch against internal scope.[12]	Build fails; artifact remains unsigned; system alerts operator.
Workflow creates infinite recursive child tasks	Task-tree depth parameters and strict budget exhaustion limits.	Concurrency monitor threshold exceeded.	Workflow paused automatically; human approval required to resume.
Malicious uninstall hook attempts to delete project data	Uninstall routines run in highly restricted, read-only data contexts.	Unauthorized filesystem or API write attempt.	Operation blocked; process forcefully killed; retained data secure.

Metrics and operational objectives

Operational maturity is tracked via precise, quantifiable metrics to ensure the system remains resilient as extension volume scales.

- **Validation Failure Rate:** Target < 2% for project onboarding manifests, indicating clear SDK usability.
- **Extension Startup Latency:** Target < 100ms for portable bytecode sandboxes, ensuring real-time responsiveness for tool plugins.
- **Orphaned-Resource Rate:** Target 0% post-uninstall verification, measured by automated reconciliation scans.
- **Quarantine Propagation Time:** Target < 500ms across all global execution nodes following a critical vulnerability disclosure.
- **Cross-Project Violation Attempts:** Target strictly 0; any non-zero value initiates an immediate forensic security audit and incident response protocol.

Build-versus-buy analysis

Building a secure extension architecture requires strategic delegation of non-differentiating components to proven standards.

- **Extension Registry:** *Adopt.* Utilize open-source, OCI-compliant artifact registries to leverage existing ecosystem tooling for content-addressed storage and distribution.[5]
- **Signature and Provenance Service:** *Adopt.* Integrate established keyless signing frameworks and transparency logs rather than constructing bespoke public key infrastructure, reducing operational burden.[14]
- **Policy Engine:** *Adopt.* Implement standard policy-as-code engines to evaluate complex configuration overlays and residency requirements.
- **Control Plane API and Orchestration:** *Build.* The specific capability routing, project configuration overlays, and zero-trust lifecycle state machine represent the platform's core differentiator and demand a bespoke, highly optimized internal implementation.

Architecture decision records

To maintain clear traceability of architectural evolution, key decisions are formalized.

- **ADR 001: Out-of-Process Execution:** *Decision:* All extensions, regardless of trust tier, run out-of-process. *Rationale:* Prevents core control plane crashes resulting from extension memory leaks or panics, ensuring systemic reliability.
- **ADR 002: Capability-Based Routing:** *Decision:* Route tasks by required capabilities, not hard-coded provider names. *Rationale:* Eliminates vendor lock-in and allows policy-driven fallback routing during provider outages.
- **ADR 003: Default-Deny Network Access:** *Decision:* Network egress is blocked by default at the host namespace level. *Rationale:* Immediately mitigates data exfiltration risks from compromised deep dependencies.
- **ADR 004: Content-Addressed Packaging:** *Decision:* Extensions are immutable and fetched exclusively by cryptographic digest. *Rationale:* Guarantees reproducible installations and entirely eliminates the risk of silent upstream mutations.

Phased implementation roadmap

A pragmatic, phased rollout mitigates risk while continuously delivering developer value.

- **Phase A — Extension contracts:** Design the foundational taxonomy, JSON project and extension manifests, capability models, and strict security invariants.
- **Phase B — Project onboarding foundation:** Implement the core project registry, local onboarding SDK, dry-run validation engine, and repository connection logic.
- **Phase C — Internal extension framework:** Construct the first extension host using portable bytecode, deploy the JSON-RPC protocol gateway, and implement capability discovery routing.
- **Phase D — Certification and supply chain:** Integrate keyless signing processes, SBOM validation pipelines, and transparency log admission controllers.
- **Phase E — Upgrade and lifecycle governance:** Implement side-by-side version routing, canary upgrade policies, automated quarantine protocols, and verified uninstall routines.
- **Phase F — Broader extension ecosystem:** Open the internal registry to verified

third-party publishers, implementing advanced automated conformance suites and multi-tenant publisher controls only when internal governance is incontrovertibly proven.

Ordered engineering backlog

To immediately commence development, the following backlog provides highly actionable engineering tasks.

Project Onboarding & Manifests:

1. Define the formal JSON schema for the canonical project manifest.
2. Implement the static manifest validator within the control plane.
3. Construct the CLI SDK for local project scaffolding and manifest generation.
4. Implement the SDK dry-run engine for network and policy simulation.
5. Create the repository connection adapter, enforcing read-only volume mounts.

Registry & Supply Chain: 6. Deploy an internal, OCI-compliant artifact registry for all extensions. 7. Implement the keyless signature verification admission controller. 8. Build the SBOM parser and dependency conflict resolver. 9. Implement internal namespace scoping rules to prevent dependency substitution attacks.[12]

Extension Host & Protocols: 10. Build the portable bytecode extension host sandboxing environment. 11. Implement the structured JSON-RPC protocol gateway for control-plane communication. 12. Design the dynamic capability directory and routing engine. 13. Integrate workload identity document issuance for extension hosts.[4]

Permissions & Governance: 14. Implement the capability-request approval intent state machine. 15. Build the credential broker to issue short-lived, dynamic secrets. 16. Enforce default-deny network egress policies dynamically on host sandboxes. 17. Implement project-scoped cryptographic isolation for memory namespaces.

Lifecycle & Upgrades: 18. Construct the installation plan generator and impact previewer. 19. Implement side-by-side version routing to support durable workflows. 20. Create the automated quarantine and emergency revocation controllers. 21. Develop verified uninstall routines ensuring complete audit preservation.

Final recommendations

The following dictates the absolute architectural posture required to safely operate the extensible agent platform:

- **Recommended Extension Architecture:** Utilise portable bytecode sandboxes for trusted internal tools and policy modules to ensure low latency. Deploy hardware-enforced microVMs or strictly bounded rootless containers for arbitrary, unverified coding runtimes and complex provider adapters.
- **Recommended Project-Onboarding Architecture:** Projects must be defined purely via declarative manifests, validated statically by an SDK, and simulated locally for policy compliance before control-plane activation.
- **Recommended Capability and Permission Model:** Permissions must be requested via granular capabilities rather than broad roles. Upgrades must never silently expand permissions; any new capability request mandates explicit human review.

- **Recommended Compatibility Policy:** The platform must support side-by-side versioning natively to guarantee that durable workflows do not break upon adapter upgrades.
- **Recommended Supply-Chain Policy:** The admission controller must reject any artifact lacking a verified cryptographic provenance attestation and an exhaustive SBOM.[6] Internal registries must be strictly mapped to prevent namespace confusion attacks.[12]
- **Minimum Safe Baseline:** No extension may execute without an immutable version digest, complete out-of-process isolation, default-deny networking, and short-lived credentials strictly bounded by a workload identity document.

Research gaps

While this architecture provides a robust framework for secure extensibility, further prototyping and validation are required in two specific domains:

1. The exact latency overhead of enforcing strict microVM isolation for high-frequency, read-only tool plugins, and whether lightweight bytecode environments can fully replicate required OS-level dependencies for complex external integrations.
2. The optimal mechanism for performing highly complex, multi-stage data migrations of project memory namespaces when an underlying vector-store provider adapter is deprecated and replaced, ensuring zero downtime for semantic retrieval tasks.

Final synthesis question

What is the leanest secure extension architecture that allows a provider-neutral agent platform to onboard new projects, repositories, tools, provider adapters, runtimes, workflows, fleet templates, memory namespaces, CI systems, and environment images without changing core platform code, while preserving isolation, compatibility, supply-chain integrity, upgrade safety, revocability, and long-term lifecycle governance? The leanest secure extension architecture leverages a strictly **declarative, capability-based control plane coupled with a zero-trust, out-of-process execution plane.** By combining standardised artifact registries for content-addressed distribution, cryptographic supply-chain attestations for unforgeable provenance, workload identity documents for dynamic, short-lived credentialing, and a structured, stateless remote procedure call protocol for all communication, the platform achieves total decoupling. This guarantees that arbitrary third-party extensions can be safely ingested, sandboxed, monitored, and deprecated, ensuring the core platform code remains untouched while stringent isolation and lifecycle governance are seamlessly enforced across the entire multi-tenant ecosystem.

Works cited

1. Let's discuss sandbox isolation - Shayon Mukherjee, accessed July 27, 2026, https://www.shayon.dev/post/2026/52/lets-discuss-sandbox-isolation/
2. How is JSON-RPC used in the Model Context Protocol? - Milvus, accessed July 27, 2026, https://milvus.io/ai-quick-reference/how-is-jsonrpc-used-in-the-model-context-proto

col
3. Overview - Model Context Protocol, accessed July 27, 2026, https://modelcontextprotocol.io/specification/2025-03-26/basic
4. SPIRE Concepts | SPIFFE, accessed July 27, 2026, https://spiffe.io/docs/latest/spire-about/spire-concepts/
5. Packaging | wasmCloud, accessed July 27, 2026, https://wasmcloud.com/docs/v1/concepts/packaging/
6. What Is In-toto Attestation? A Practical Guide - Safeguard, accessed July 27, 2026, https://safeguard.sh/resources/blog/what-is-in-toto-attestation
7. Temporal in Production: Sharp Edges & Good Practices - Nejc Korasa, accessed July 27, 2026, https://nejckorasa.github.io/posts/temporal-in-production/
8. Versioning - Durable Workflow, accessed July 27, 2026, https://durable-workflow.com/docs/features/versioning/
9. Wasm OCI Artifact layout - CNCF TAG Runtime, accessed July 27, 2026, https://tag-runtime.cncf.io/wgs/wasm/deliverables/wasm-oci-artifact/
10. Model Context Protocol (MCP) explained: A practical technical overview for developers and architects - CodiLime, accessed July 27, 2026, https://codilime.com/blog/model-context-protocol-explained/
11. Packaging | wasmCloud, accessed July 27, 2026, https://wasmcloud.com/docs/overview/packaging/
12. Dependency Confusion - HackTricks, accessed July 27, 2026, https://hacktricks.wiki/en/pentesting-web/dependency-confusion.html
13. What is Dependency Confusion | Risks & Prevention - Imperva, accessed July 27, 2026, https://www.imperva.com/learn/application-security/dependency-confusion/
14. Supply-Chain Security for Java. SBOMs, Sigstore signing, SLSA… | by Rahul Kumar | Medium, accessed July 27, 2026, https://medium.com/@27.rahul.k/supply-chain-security-for-java-0651c1e21976
15. What Is in-toto? Securing the Software Supply Chain End to End | Sbomify, accessed July 27, 2026, https://sbomify.com/2024/08/14/what-is-in-toto/
16. GDPR Data Residency: Where EU Personal Data Can Live - Opsio, accessed July 27, 2026, https://opsiocloud.com/blogs/gdpr-data-residency-where-european-personal-data-can-live/