Software Supply-Chain Security and Release Engineering for Agentic Factories

Executive summary

The introduction of autonomous coding agents into an agentic software factory fundamentally accelerates the rate of code generation, dependency modification, and infrastructure mutation. This velocity invalidates traditional, manual release engineering. When agents can autonomously inspect repositories, modify code, run tests, and propose deployments, the supply chain must shift from implicitly trusting human intent to explicitly verifying cryptographic evidence. This exhaustive DevSecOps architecture report outlines a provider-neutral framework designed to transform source code into verified, reproducible, traceable, promotable, deployable, and recoverable software artefacts. It mandates that deployment environments trust only immutable artefact digests accompanied by signed provenance and verified Software Bill of Materials (SBOM) records, ensuring that no agent—or compromised human operator—can bypass security controls, tamper with build environments, or deploy unverified code.

Recommended direction

Organisations operating an agentic software factory must adopt a zero-trust supply-chain posture based on the Supply-chain Levels for Software Artifacts (SLSA) v1.0 specifications and the Open Container Initiative (OCI) v1.1.0 standard.[1] The foundational recommendation is the strict decoupling of build execution from release decision-making. Artefacts must be built once in ephemeral, isolated environments, identified solely by immutable cryptographic digests, and promoted through release channels without modification. Rebuilding from source for different deployment environments must be strictly prohibited. Furthermore, agent-generated code must be subjected to deterministic security scanning, strict merge-queue serialisation, and human-operator approval boundaries for high-risk changes, ensuring autonomous speed does not compromise production stability.

Scope and assumptions

This architecture covers the end-to-end software supply chain from repository change to production deployment, including source intake, dependency resolution, build execution, SBOM generation, provenance attestation, release candidate creation, and rollback procedures. It encompasses application code, agent-platform services, plugins, runtime container images, language packages, and infrastructure-as-code configurations. The report assumes the organisation operates a small but scaling technical team heavily augmented by

autonomous agent workers. The foundational infrastructure is assumed to include a version control repository provider, a continuous integration (CI) control plane, a key-management framework, and an OCI-compliant artefact registry. It is further assumed that while agents possess the capability to read code, propose changes, and trigger CI, they are strictly prohibited from holding administrative privileges over repository branch protections, signing keys, transparency logs, or production deployment controllers.

Supply-chain threat model

The agentic software factory faces unique supply-chain threats that blend traditional DevSecOps vulnerabilities with autonomous-system risks. Autonomous agents introduce novel threat vectors, such as the rapid introduction of obscure dependencies, the masking of failing tests through automated CI modifications, and the generation of semantically conflicting changes that pass independent checks but fail upon integration.

Threat Category	Specific Threat	Preventive Control	Detective Control	Recovery Control	Residual Risk
Source	Malicious agent pull request	Protected branches, mandatory deterministic review	Static analysis, strict test suites	Revert commit, isolate branch	Logic flaws evading deterministic tests
Source	Force-push rewriting history	Branch protection configuration	Audit logs for forced pushes	Restore branch history from immutable ledger	Administrator account compromise
Dependency	Typosquatting / Confusion	Registry proxy, scoped internal namespaces	Dependency scanning, SBOM diffs	Package quarantine, rollback	Zero-day execution before detection
Dependency	Malicious lifecycle script	Disable scripts globally (ignore-scri	File-system monitoring in CI	Wipe CI environment, block package	Sandbox escape during build

		pts)			
Build	Persistent runner contamination	Ephemeral runners, workload identity	Provenance verification, independent rebuilds	Rotate builder identity, discard cache	Leakage of short-lived tokens
Build	Non-hermetic build	Default-deny network restrictions during compile	Build inputs manifest verification	Discard unverified artefact	Undetected compiler modification
Release	Signing-key compromise	Keyless signing, short-lived certificates	Transparency log monitoring	Revoke identity, re-sign artefacts	Verification delays in offline sites
Release	Mutable release tag overwrite	Immutable digest deployment	Registry tag immutability policies	Rollback to known digest	Misconfigured deployment manifests
Agent	Agent weakens CI checks	Required status checks, code-owner review	CI configuration audit logs	Restore CI files, revoke agent token	Collusion with compromised operator
Agent	Merge-queue race conditions	Serialised or batched merge queue	Post-merge shadow test execution	Revert merge candidate	Semantic conflicts evading tests

Trust boundaries and invariants

Clear cryptographic trust boundaries must separate the execution plane where agents operate from the control plane where trust is anchored. Systems that must remain strictly outside agent execution environments include the git provider's branch-protection configuration, the

merge-queue policy engine, the key-management service, the provenance store, and the release controller. Agents operate on untrusted branches and propose changes. The CI control plane orchestrates isolated build runners, establishing a boundary where the builder identity is verified. The artefact registry and signing services sit behind strict workload-identity boundaries. Agents must never possess unrestricted access to production credentials, provenance roots, artefact-retention controls, or audit-history deletion mechanisms.

Supply-chain maturity model

The organisation must progress through defined maturity stages, mapping directly to operational readiness and risk tolerance.

Stage 0 entails informal builds characterised by manual processes, mutable environments, weak dependency pinning, and the absence of provenance. This stage is fundamentally unsuitable for agentic factories, as the lack of deterministic traceability makes it impossible to audit autonomous actions. Stage 1 introduces controlled CI builds, requiring protected branches, mandatory CI, ephemeral runners, dependency lockfiles, and basic SBOM generation. Stage 2 achieves verifiable builds by mandating isolated builders, signed provenance, signed artefacts, content-addressed promotion, and strict merge queues. Stage 3 establishes a hardened supply chain through stronger SLSA alignment, near-hermetic builds, independent provenance verification, a trusted release controller, and keyless signing procedures. Finally, Stage 4 represents a high-assurance multi-project factory, implementing per-project trust domains, policy-driven dependency approval, independent build verification, multi-party approval for sensitive releases, and external transparency notarisation.

Progression requires satisfying strict entry and exit criteria. Moving from Stage 1 to Stage 2, for example, demands that 100% of production deployments are triggered by immutable digests rather than tags, with automated verification of signatures at the admission controller.

Practical SLSA target

The architecture aligns with the SLSA v1.0 specification, which separates requirements into distinct tracks to evaluate the trustworthiness of software artefacts.[1] The SLSA Build track focuses on the build process and provenance generation. The organisation should adopt a pragmatic, risk-tiered approach to SLSA compliance. Early internal services and generic-worker images must target SLSA Build Level 2, which requires a hosted build platform generating signed provenance, authenticating the build service.[4] Agent-platform control-plane services, high-risk execution components, and security-critical plugins must target SLSA Build Level 3, which mandates that the build is isolated, parameters are completely externalised, and provenance is non-falsifiable by the build workload.[5] SLSA Build Level 1 controls, establishing that provenance exists and is distributed to consumers, are mandatory immediately across all projects.[4] Requirements for SLSA Build Level 4, involving fully hermetic and reproducible builds, provide exceptional security but demand disproportionate operational complexity for a small team; these should be deferred until managed infrastructure inherently supports them.

Agent-generated-code trust model

Code authored or modified by autonomous agents is inherently untrusted. The trust model must not assume agent-generated code is inherently malicious, nor inherently safer than human code; rather, it is prone to contextual hallucination, semantic logic errors, and unwarranted dependency additions. Agent-generated changes must be validated through rigid repository policy, deterministic static analysis, strict testing, security scanning, and trusted builds. Agent authorship dictates the review model: changes to low-risk application logic may be auto-merged after deterministic validation or independent peer-agent review. Conversely, changes affecting authentication, authorisation, secrets, sandboxing, deployment infrastructure, or CI configuration trigger mandatory human multi-party approval. Agent tasks hold lower merge-queue priority than human-operator emergency hotfixes.

Repository and branch governance

Repositories must be segmented by risk, applying the strictest governance to control-plane, infrastructure, and security-critical extension repositories. Protected branches must mandate required status checks, enforcing that pull requests pass automated testing and security scanning. Stale approvals must be dismissed immediately upon new code being pushed to an open pull request. Code-owner review is strictly required for sensitive directories, ensuring agents cannot unilaterally alter infrastructure or security configurations. Repositories must enforce linear history, prohibiting merge commits that obfuscate task lineage, while strictly prohibiting force-pushes and branch deletions on main branches.

Review and merge-queue architecture

In an agentic factory, multiple agents operate concurrently, submitting pull requests simultaneously. Independent pull-request CI passes become invalid when conflicting changes merge close together, leading to integration failures.[6] The architecture mandates a dependency-aware merge queue to construct candidate integration states.

Merge Queue Model	Description	Security	Speed	Small-Team Suitability
Strict Serial	Processes one PR at a time.	Highest	Slowest	Moderate
Speculative	Tests multiple PRs in parallel assuming previous ones pass.	High	Fast	Low (High complexity)

Batched	Groups compatible PRs into a single integration test.	High	Fast	High (Recommended)
Dependency-Aware	Analyses code graphs to batch non-overlapping changes.	Very High	Optimal	Low (Deferred)

The recommended batched merge queue constructs a candidate state containing the latest protected-branch state and batches compatible changes. It reruns required checks in an ephemeral environment, records the candidate commit, and links the results to the exact candidate digest. If a batch fails, the queue executes a bisect strategy to identify the failing agent-generated change, requeuing the valid changes and attributing the failure accurately.

Dependency governance

Dependencies represent a massive attack surface. The dependency policy must govern direct, transitive, development, build, and test dependencies, alongside operating-system packages and base images. Every dependency source must be identifiable. The factory must route all external package requests through an approved dependency caching proxy. This proxy serves as the enforcement point for allow and deny rules, blocking prohibited packages, prohibited licences, and packages containing unresolved vulnerabilities exceeding maximum severity thresholds. When an agent proposes adding a new dependency, it must compile a review packet displaying the package age, recent maintenance, transitive dependency count, install scripts, and a robust justification for why internal utility modules cannot fulfill the requirement.

Lockfile and lifecycle-script policy

Lockfiles are unequivocally mandatory. Source manifests and lockfiles must remain consistent, ensuring deterministic ordering, checksum validation, and registry URL pinning. Package lifecycle scripts, such as preinstall, install, and postinstall, represent a critical arbitrary-code execution risk during dependency resolution.[7] The package-manager configuration must explicitly disable scripts by default (e.g., using ignore-scripts=true in npm ecosystems).[8] Packages that genuinely require execution of build hooks or native compiler plugins must be explicitly allowlisted and executed within a restricted, network-isolated sandbox that records the process tree and scans generated binaries. Disabling lifecycle scripts neutralises broad classes of supply-chain attacks, such as remote access trojans injected via typosquatting, with

minimal compatibility impact once native bindings are properly allowlisted.[7]

Vulnerability and exception management

Vulnerability management distinguishes between raw vulnerability severity, actual exploitability, runtime reachability, and environmental exposure. The factory must not automatically block all releases for low-relevance vulnerabilities, which induces alert fatigue. The lifecycle involves detection, reachability analysis, deployed exposure determination, and automated remediation task creation. Temporary dependency exceptions must be strictly governed. An exception record must contain the dependency, version, vulnerability issue, affected project, justification, compensating controls, human approver, and a hard expiry date. Expired exceptions fail closed, triggering explicit escalation and blocking subsequent artefact promotion.

Build-isolation architecture

The build environment guarantees that a compromised dependency or malicious agent-generated script cannot exfiltrate secrets, manipulate the build toolchain, or poison subsequent builds.

Build Environment	Isolation	Startup Cost	Cache Safety	Reproducibility
Persistent Runner	Low	Low	Low (Risk of leakage)	Low
Ephemeral Container	Moderate	Low	Moderate	Moderate
Sandboxed Container	High	Moderate	High	High
MicroVM	Very High	High	Very High	Very High

The architecture mandates ephemeral containers for routine internal builds and untrusted pull requests, providing a clean environment per build with short-lived workload identity and deterministic teardown. High-risk extensions and environment-image builds should graduate to MicroVMs to ensure robust hypervisor-level isolation.

Hermeticity and reproducibility

Achieving a fully hermetic build, where all inputs are entirely self-contained without dynamic network access, is the ultimate goal for verifiable builds. The architecture implements dependency-pinned networked builds, where the build network policy restricts egress strictly

to the approved dependency proxy. The system captures a comprehensive build inputs manifest containing the source commit, repository digest, dependency lockfile digest, base image digest, compiler versions, and target platform.[5] While byte-for-byte reproducibility is technically challenging across diverse language ecosystems, functional equivalence is mandatory. For security-critical components, the architecture executes independent rebuild verification, wherein a separate CI builder in a separate account recompiles the declared inputs without access to the original artefact, subsequently comparing the output digests to validate integrity.

Build-cache and secret security

Shared compiler and dependency caches pose significant cross-project data leakage and poisoning risks. Cache architectures must employ strict namespaces separated by project and trust tier. Release builds executing on protected main branches must utilize read-only verified caches or execute from a completely clean state to prevent cache-key collisions injected by malicious untrusted pull requests. Furthermore, build secrets must never persist into artefact layers. Long-lived credentials must be replaced by workload identity brokered operations, using short-lived tokens mounted securely as temporary files or environment variables that are scrubbed prior to artefact finalisation.

SBOM architecture

The Software Bill of Materials provides exhaustive transparency into the composition of the artefact. The ecosystem currently supports two dominant standards: SPDX and CycloneDX.[10]

SBOM Standard	Focus	Capabilities	Structure	Small-Team Suitability
SPDX 3.0	Licensing, Compliance, Build	Profiles (Core, Build, Security, AI), JSON-LD	Highly detailed, file-level	High for Base Images [12]
CycloneDX 1.6	AppSec, Vulnerabilities	VEX, ML-BOM, CBOM	Component/Service centric	High for Applications [11]

The agentic factory should utilize an internal canonical model during build execution, automatically exporting to both SPDX 3.0 (for infrastructure, operating system packages, and build provenance profiles) and CycloneDX 1.6 (for application vulnerability exploitability exchange and machine learning model tracking).[11] SBOM generation must occur at the final artefact stage to accurately reflect the true composition of the container image, capturing dynamically generated code, statically linked libraries, and base-image layers.

SBOM validation and diffing

Generated SBOMs must not be implicitly trusted. Validation checks must confirm schema validity, component completeness, and ensure the SBOM digest perfectly matches the expected artefact digest. During the pull request and release review phases, agents or operators must be presented with a human-readable SBOM delta showing added, removed, or upgraded components, new lifecycle scripts, and new native binaries. This diffing strategy exposes supply-chain drift before integration.

Provenance attestations

Provenance asserts the verifiable history of software artefacts, detailing where, when, and how they were produced.[5] The architecture implements the in-toto attestation framework. A standard attestation envelope wraps a signed statement containing a subject (the artefact digest) and a predicate.[15] The factory must use the standard SLSA Provenance predicate (https://slsa.dev/provenance/v1), which explicitly defines the buildDefinition (build type, external parameters, and resolved dependencies) and runDetails (builder identity, metadata, and timestamps).[5] Crucially, provenance must be generated autonomously by the trusted build control plane, not supplied by the execution workload, ensuring tamper resistance.[18]

Artefact signing and key governance

Artefact signing provides the cryptographic guarantee of provenance and integrity. Traditional long-lived signing keys require complex key custody, hardware-backed key management, and catastrophic revocation procedures.[20] The factory must instead adopt a keyless signing architecture backed by transparency logs and The Update Framework (TUF).[22]

Signing Model	Complexity	Identity Binding	Compromise Impact	Revocation Speed
Long-lived Key	Moderate	Weak (Key bound)	Catastrophic	Slow
Cloud KMS	High	Strong (IAM bound)	High	Fast
Keyless Signing	Low	Strong (OIDC bound)	Low	Instant (Short-lived)

Under keyless signing, the ephemeral private key exists only in memory during the signing operation and is immediately discarded. A short-lived certificate binds the OpenID Connect

(OIDC) workload identity to the ephemeral public key. The signature and certificate are published to a public or internal transparency log.[20] Verification requires checking that the artefact is recorded in the transparency log, providing a non-repudiable proof chain that withstands the compromise of individual CI accounts.[21]

OCI artefacts and registry design

The OCI-compliant artefact registry serves as the authoritative distribution mechanism for container images, environment images, policy bundles, and generic binary packages.[2] Utilizing the OCI Image Specification v1.1.0, the registry must leverage the Referrers API and the subject field to cryptographically link an image to its associated signatures, SBOMs, and provenance attestations without mutating the original image manifest.[2] The registry design mandates strict namespace project isolation, immutable tag configurations to prevent tag overwriting, and rigorous garbage collection policies that refuse deletion of digests actively referenced by deployments, rollback plans, or legal holds.

Release bundles and release manifests

A release transcends a singular container image; it is a cryptographic bundle. A release bundle encompasses the release manifest, artefact digests, signatures, provenance attestations, SBOMs, vulnerability results, database migration plans, configuration references, and deployment policy. This entire bundle must be content-addressed. The release manifest serves as the orchestrating document, explicitly defining the release ID, source commit, target channel, and rollback target, ensuring the deployment controller possesses all requisite context to execute or revert the release deterministically.

Artefact promotion and release channels

Artefact promotion is defined as moving or approving the same verified artefact across progressive environments (development, staging, canary, production) rather than rebuilding the source separately for each environment. Rebuilding source independently for staging and production inevitably introduces supply-chain divergence, as transient variables such as timestamps, dynamic dependency resolution, or compiler cache states shift between builds. Promotion records must contain the artefact digest, the source and destination environments, test evidence, the policy evaluation result, and the human or agent approver. Release channels function as logical pointers to immutable digests, governed by distinct entry criteria, provenance requirements, and vulnerability policies.

Progressive delivery and canary releases

High-velocity agentic deployments require progressive delivery mechanisms to limit blast radius. A canary release strategy must route a small percentage of traffic or specific low-risk tenant tasks to the newly promoted digest. Canary metrics must extend beyond rudimentary process uptime to encompass error rates, latency, task completion success, policy violations, and evidence completeness. The canary evaluator mandates a strict observation window;

failure to maintain success thresholds triggers automatic rollback to the prior known-good digest, alerting the operator console.

Rollback and roll-forward

Rollback safety guarantees resilience against defective agent-generated changes. The deployment controller must rollback to an immutable, known-good artefact digest, automatically verifying its signature, provenance, and database schema compatibility before execution. However, roll-forward is functionally mandated when a release involves irreversible database schema migrations, external API side-effects, or security-key rotations. The architecture necessitates a decision matrix to evaluate state compatibility, enforcing that database migration code is versioned, rigorously tested in the merge queue, and securely packaged within the release bundle.

Database, infrastructure, configuration, and policy releases

Infrastructure-as-code and configuration modifications must be treated as versioned release inputs. Autonomous agents may prepare terraform plans or configuration bundles, but applying these changes requires strict approval binding the plan digest to the release manifest. Policy bundles governing security constraints must undergo tests, simulation, approval, and staged promotion. A code release must never be permitted to silently weaken release policy or infrastructure configurations.

Emergency-hotfix engineering

Emergency hotfixes necessitate a rapid path that reduces canary durations and review counts, but they must never become an evidence-free bypass mechanism. An emergency hotfix may accelerate the release window, but it is strictly prohibited from omitting source identity, commit identity, build isolation, artefact digests, provenance generation, or keyless signing. The break-glass release authority requires step-up authentication and comprehensive audit logging. Following the deployment of an emergency hotfix, the system mandates a post-hotfix reconciliation process—enforcing a full test suite run, SBOM validation, main-branch reconciliation, and incident review within a stringent time limit.

Release evidence and audit timeline

To guarantee verifiability against insider threats and compromised infrastructure, the release evidence chain must be anchored cryptographically. This chain links the task, branch, commit, pull request, test evidence, SBOM, provenance, signature, release, promotion, and deployment. The evidence chain is anchored using Merkle roots submitted to an internal transparency log or an external timestamp authority, ensuring that audit trails are tamper-evident and historically immutable.[25] The operator console must surface a chronological timeline answering precisely which agent task created the change, which builder created the artefact, which dependencies

were integrated, and which immutable digest was deployed.

Release policy and deployment admission

The deployment environment must enforce strict admission control. An admission controller validates the release bundle prior to execution. The controller must deterministically reject any unsigned artefact, invalid signature, unknown builder identity, missing provenance, unapproved commit, prohibited dependency, revoked key, or mutable tag lacking a resolved digest.[21] Agents may investigate and explain policy failures, but they possess absolutely zero authority to override deterministic policy checks.

Incident and compromise response

Robust procedures must govern the response to a compromised dependency, builder identity, or signing key. If a dependency is compromised, the SBOM store executes a global query to identify all affected deployed artefacts, triggering immediate quarantine, temporary blocking, and orchestrated rebuilds using secure alternatives. If a signing key or builder identity is compromised, the incident response involves immediate trust revocation via the transparency log, generating an inventory of affected signatures, executing re-verification utilizing historical provenance, establishing new key material, and communicating the revocation to the operator console.[22]

Disaster recovery for release infrastructure

Release infrastructure is as critical as production application infrastructure. Disaster recovery protocols must ensure secure, resilient backups of the artefact registry, SBOM store, provenance store, signing metadata, and policy bundles. Recovery exercises must validate that restored artefact registries map seamlessly to restored provenance records, ensuring that no cryptographic links are broken during the restoration process, thereby preserving the integrity of the supply-chain evidence chain.

Reference architecture

The reference architecture defines the flow of source code to deployed artefact through strictly governed trust boundaries.

Code snippet

```
sequenceDiagram
  participant Agent as Agent Worker
  participant Repo as Source Repository
  participant MQ as Merge Queue
  participant CI as CI Control Plane
  participant Reg as OCI Registry & Logs
```

```
    participant Deploy as Deployment Controller

    Agent->>Repo: Open PR (Task-linked Branch)
    Repo->>MQ: Trigger Deterministic Tests
    MQ->>Repo: Enforce Status Checks & Approvals
    Repo->>CI: Merge to Main & Trigger Release Build
    Note over CI: Ephemeral Builder Provisioned
    CI->>CI: Fetch Dependencies via Proxy
    CI->>CI: Compile & Generate SBOM
    CI->>CI: Generate SLSA Provenance
    CI->>Reg: Keyless Signing & Write to Transparency Log
    CI->>Reg: Upload Artefact, SBOM, Provenance (OCI Referrers)
    Reg->>Deploy: Promote Release Candidate
    Deploy->>Deploy: Verify Signatures & Provenance
    Deploy->>Deploy: Execute Canary Release
```

The architecture segregates the execution plane (Agent, Repo) from the cryptographic control plane (CI, Registry) and the operational control plane (Deployment Controller). The source flows through the merge queue, triggering the CI orchestrator to spawn ephemeral builders. Artefacts flow into the OCI registry, attached to provenance and signatures, which are subsequently validated by the deployment controller prior to production execution.

Schemas and contracts

To ensure provider independence, systems communicate via standardised schemas.
Provenance Attestation (in-toto SLSA v1.0 Model) [5]:

JSON

```
{
  "_type": "https://in-toto.io/Statement/v1",
  "subject": [
    {
      "name": "example-service",
      "digest": { "sha256":
"e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855" }
    }
  ],
  "predicateType": "https://slsa.dev/provenance/v1",
  "predicate": {
    "buildDefinition": {
      "buildType": "https://agent-platform.internal/build-profile/v1",
```

```
    "externalParameters": { "source":
"git+https://git.internal/project/example-service@refs/heads/main" },
    "resolvedDependencies": [
     {
      "uri": "git+https://git.internal/project/example-service@a1b2c3d4",
      "digest": { "gitCommit": "a1b2c3d4e5f6g7h8i9j0" }
     }
    ]
   },
   "runDetails": {
    "builder": {
     "id": "https://ci.internal/ephemeral-builder/v1"
    },
    "metadata": {
     "invocationId": "run-987654321",
     "startedOn": "2026-07-27T04:35:00Z",
     "finishedOn": "2026-07-27T04:38:00Z"
    }
   }
  }
}
```

Release Manifest Schema:

```
JSON
{
 "releaseId": "rel-20260727-alpha",
 "project": "example-service",
 "version": "1.4.2",
 "sourceCommit": "a1b2c3d4e5f6g7h8i9j0",
 "artefacts":,
 "databaseMigrations": ["migration-0042.sql"],
 "targetChannel": "production",
 "rollbackTarget": "rel-20260720-stable"
}
```

Security invariants

The supply chain is governed by immutable security invariants. Production environments execute only immutable artefact digests; mutable tags are inherently untrusted. Release builds

are isolated, ephemeral, and strictly prohibited from modifying signing keys. Provenance is exclusively generated by the trusted builder identity, ensuring the build workload cannot falsify its own origins. Every production artefact must possess a cryptographically valid signature. Tested artefacts are promoted across environments; rebuilding code independently per environment is forbidden. Untrusted pull requests generated by agents receive absolutely no production secrets. Rollbacks utilise previously verified, known-good artefacts. Quarantined artefacts cannot be promoted, and revoked signing identities are deterministicly rejected by admission controllers.

Conformance and adversarial tests

Automated adversarial tests must routinely validate the integrity of the supply-chain architecture. Conformance tests verify protected-branch enforcement, merge-queue batching, lockfile consistency, lifecycle-script blocking, and build cache separation. Adversarial scenarios must be simulated to assess resilience. If an agent attempts to change CI configurations to skip security scans, the code-owner review invariant blocks the pull request. If a new transitive dependency introduces a malicious lifecycle script, the global ignore-scripts policy neutralises the threat during dependency resolution. If an attacker replaces an artefact behind a mutable tag, the deployment admission controller rejects the payload because the referenced immutable digest fails cryptographic verification. If a signing key is compromised, transparency log monitors detect anomalous issuance, triggering immediate revocation and artefact quarantine.

Metrics and operational objectives

Agentic factory metrics must meticulously balance velocity against verifiable security. Standard operational metrics encompass lead time, deployment frequency, change failure rate, and time to recovery. However, agent-specific metrics are vital for monitoring autonomous behavior, including the volume of agent-created changes per release, agent dependency-addition rates, autonomous test-fix success, merge conflicts between parallel agents, and the frequency of agent-caused release failures. Operational objectives must prioritise release-evidence completeness and policy-block accuracy. These metrics must not be gamified to encourage unsafe speed; policy blocks generated by untrusted agent code represent a successful security intervention, not operational friction.

Cost and build-versus-buy analysis

Optimising controls for a small organisation demands strategic procurement to minimise operational burden without compromising security criticality.

Component	Strategy	Justification
Source Hosting & CI	Buy (Managed)	Commoditised; high operational burden to

		self-host.
Merge Queue	Buy	Complex state management and speculative execution logic.
Dependency Proxy	Buy (Managed)	Requires high uptime, global caching, and vulnerability DB integration.
SBOM & Provenance Gen.	Build (OSS Tooling)	Standard CLI tools easily integrate into existing CI pipelines.
Signing Infrastructure	Adopt (Keyless OSS)	Eliminates KMS costs, key-rotation overhead, and complex custody.
OCI Artefact Registry	Buy (Managed)	OCI v1.1.0 support and multi-region replication are critical for resilience.
Policy/Admission Engine	Build (OSS Tooling)	Standard Kubernetes admission controllers provide high portability.

Architecture decision records

- **ADR 001: Immutable Artefact Digests as Deployment Identity**
  - *Context*: Mutable tags lead to environment drift, rollback ambiguity, and susceptibility to tag-overwriting attacks.
  - *Decision*: All deployments, release records, and promotion channels must reference exact cryptographic digests.
  - *Consequences*: Requires tooling to resolve tags to digests prior to generating release manifests.
- **ADR 002: Artefact Promotion Instead of Rebuild Per Environment**
  - *Context*: Rebuilding code for staging, then rebuilding again for production introduces variables such as new timestamps or dynamic transitive dependencies, invalidating prior test evidence.
  - *Decision*: Compile once. Promote the verified OCI artefact digest across channels.

    - *Consequences*: Application configuration must be fully externalised from the compiled binary.
- **ADR 003: Mandatory Lockfile Governance and Restricted Lifecycle Scripts**
    - *Context*: Supply-chain attacks routinely exploit unpinned dependencies and execute remote access trojans via malicious postinstall scripts.
    - *Decision*: Lockfiles are mandatory. Global ignore-scripts enabled across all package managers.
    - *Consequences*: Legitimate native bindings must be explicitly allowlisted and built via controlled, sandboxed tasks.
- **ADR 004: Keyless Signing through Trusted Workload Identity**
    - *Context*: Long-lived signing keys pose extreme compromise and revocation risks.
    - *Decision*: Adopt keyless signing utilizing OIDC workload identities and transparency logs.
    - *Consequences*: Requires deployment admission controllers capable of verifying transparency log inclusion proofs.

Phased implementation roadmap

To prevent operational gridlock, controls must be implemented incrementally, targeting the highest-risk vectors first.

Phase A — Source and dependency governance

- Objective: Secure the repository and control dependency ingress.
- Components: Enforce protected branches, mandatory reviews, merge-queue foundations, lockfile policies, global lifecycle-script blocking, and untrusted pull-request isolation.

Phase B — Trusted build foundation

- Objective: Establish reliable, isolated artefact generation.
- Components: Deploy ephemeral container builders, immutable builder images, workload identities, isolated build caches, and build-input manifests.

Phase C — SBOM, provenance, and signing

- Objective: Cryptographically attest to the build process.
- Components: Implement artefact-level SBOM generation, CI-generated SLSA provenance, keyless artefact signing, and signature verification logic.

Phase D — Artefact promotion and release channels

- Objective: Standardise the deployment path.
- Components: Configure the OCI registry Referrers API, define release bundles, implement environment admission controllers, and automate canary release and rollback verification.

Phase E — Hardened supply chain

- Objective: Address advanced threats and insider risks.
- Components: Enforce near-hermetic build networks, implement independent rebuilds for security-critical components, integrate transparency logging, and conduct key-compromise drills.

Phase F — Mature multi-project factory

- Objective: Scale across the organisation.
- Components: Establish project-specific trust domains, advanced dependency risk scoring, multi-party approval structures, and cross-provider build verification.

Ordered engineering backlog

The following provides the initial backlog to instantiate the architecture, designed to convert directly into engineering tickets.

Source control & Merge queue

1. Enforce branch protection on all main branches, prohibiting direct pushes and force-pushes.
2. Require cryptographically signed commits for human operators.
3. Configure CODEOWNERS for security-sensitive directories (e.g., CI, infrastructure).
4. Deploy a batched merge queue requiring status checks before final merge.
5. Isolate untrusted agent pull-request runs from production CI secrets.

Dependency governance 6. Deploy an internal dependency caching proxy. 7. Configure proxy rules to block critical vulnerabilities and prohibited licences. 8. Enforce global ignore-scripts=true across all package manager configurations. 9. Create an explicit allowlist process for packages requiring native build scripts. 10. Implement mandatory lockfile drift-detection within CI workflows.

Build isolation 11. Migrate all release builds to ephemeral, single-use container runners. 12. Implement OIDC workload identities for build runners. 13. Namespace build caches by project and branch; enforce read-only cache on main branches. 14. Restrict CI network egress solely to the dependency proxy and allowed external APIs.

SBOM & Provenance 15. Integrate tooling to generate SPDX 3.0/CycloneDX 1.6 SBOMs at the final artefact stage. 16. Implement CI-level SLSA v1.0 provenance generation (https://slsa.dev/provenance/v1). 17. Bind SBOMs and provenance to the artefact digest via the OCI Referrers API. 18. Implement human-readable SBOM diffing in pull-request views.

Signing & Registry 19. Implement keyless signing utilizing short-lived OIDC certificates and transparency logs. 20. Configure OCI registry namespace retention and tag immutability policies. 21. Setup garbage collection rules based on digest deployment reachability.

Release & Promotion 22. Define standard JSON schemas for release manifests and release bundles. 23. Automate artefact promotion scripts to tag digests across staging and production. 24. Implement OCI admission controllers to reject unsigned or unverified digests. 25. Configure progressive delivery (canary) metrics for automated promotion. 26. Standardise automated binary rollback workflows based on prior known-good digests. 27. Formalise the emergency hotfix pipeline with mandatory post-hotfix reconciliation.

Final recommendations

Recommended SLSA target

Implement SLSA Build Level 2 immediately across all projects to ensure signed provenance and

hosted builds. SLSA Build Level 3 must be targeted for control-plane services, requiring ephemeral environments and complete externalisation of build parameters to guarantee non-falsifiability.

Recommended source-control policy

Enforce protected branches universally. Require at least one independent review (human or dedicated reviewer-agent) and mandate that all main-branch integration passes through a serialised or batched merge queue to prevent semantic collisions.

Recommended dependency policy

Deny all direct network fetches. Route dependencies exclusively through a governed caching proxy. Mandate lockfiles and globally disable package lifecycle scripts (ignore-scripts), requiring explicit whitelisting and sandboxing for native compilations.

Recommended build architecture

Execute all release builds in ephemeral, isolated container environments driven by short-lived workload identities. Build networks must be severely restricted, and build caches must be strictly namespaced and read-only for release branches to prevent cache poisoning.

Recommended SBOM architecture

Generate SBOMs at the final artefact stage to capture actual composition. Standardise on SPDX 3.0 for base images and infrastructure compliance, and CycloneDX 1.6 for application vulnerabilities. Store and cryptographically bind them to the artefact digest.

Recommended provenance architecture

Provenance must be generated autonomously by the trusted CI control plane, not the build script itself. It must comply with in-toto SLSA v1.0 specifications, detailing the buildDefinition and runDetails, and be cryptographically signed.

Recommended signing architecture

Abandon long-lived signing keys. Adopt keyless signing utilizing OIDC workload identities, short-lived certificates, and transparency logs to anchor signatures and ensure immediate non-repudiation and rapid revocation.

Recommended OCI artefact model

Utilise OCI v1.1.0 compliant registries as the universal distribution mechanism. Attach SBOMs, provenance, and signatures directly to the container image digest using the subject reference mechanism via the Referrers API.

Recommended artefact-promotion model

Never rebuild source code for different deployment environments. Promote verified software strictly by migrating or tagging the exact immutable cryptographic digest across progressive

release channels.

Recommended merge queue

Adopt a batched merge queue with shadow execution capabilities. Ensure agents cannot bypass the queue and that batch failures gracefully bisect to isolate malfunctioning agent-generated commits without stalling the pipeline.

Recommended canary and rollback policy

Gate production promotion behind metric-driven canary releases observing error rates and task successes. Ensure rollback mechanisms revert exclusively to previously verified, signed immutable digests rather than attempting to rebuild older source code.

Recommended emergency hotfix process

Allow expedited review and abbreviated canary periods for hotfixes, but strictly prohibit bypassing isolated builds, provenance generation, or keyless signing. Mandate full post-incident reconciliation and testing within 24 hours.

Minimum safe baseline

Before autonomous agents can propose production candidates, the factory must enforce: protected branches, lockfiles, ephemeral builds, immutable artefact digests, SBOM generation, keyless signing, and signature-verifying admission controllers in production.

Controls that may be deferred

Byte-for-byte reproducibility across all language artefacts, multi-cloud independent rebuild verification, and external public notarisation can be deferred until the organisation reaches a high-assurance enterprise scale.

Top fifteen implementation risks

1. **Compromised CI credentials** (High impact, high urgency, mitigate via workload identity).
2. **Malicious lifecycle scripts** (High probability, mitigate via ignore-scripts).
3. **Merge-queue semantic conflicts** (High probability with agents, mitigate via batched queue).
4. **Mutable tag overwrites** (Moderate detectability, mitigate via digest deployment).
5. **Dependency typosquatting** (High probability, mitigate via proxy allowlists).
6. **Loss of signing keys** (High blast radius, mitigate via keyless signing).
7. **Cache poisoning** (Moderate probability, mitigate via cache namespaces).
8. **Bypassed admission control** (High impact, mitigate via cluster policy engine).
9. **Incomplete SBOMs** (High probability, mitigate via final-stage generation).
10. **Agent logic hallucinations** (High probability, mitigate via deterministic testing).
11. **Stale vulnerability exceptions** (Moderate probability, mitigate via hard expiry).
12. **Unverified base images** (Moderate impact, mitigate via base-image provenance).

13. **Rollback failure due to database mismatch** (Moderate probability, mitigate via roll-forward workflows).
14. **Lack of transparency log redundancy** (Low probability, mitigate via internal backups).
15. **Over-privileged CI tokens** (High blast radius, mitigate via scoped OIDC tokens).

Research gaps

Current supply-chain security standards exhibit critical gaps when applied to dynamic agentic systems. While CycloneDX 1.6 and SPDX 3.0 introduce AI and ML-BOM profiles, mapping autonomous agent models—which may shift contextually—to static artefact attestations remains theoretically immature. Furthermore, standard version control providers lack native "shadow execution" for merge queues, where the systemic side-effects of an agent's code are evaluated transactionally across microservices before commit, necessitating custom orchestration. Finally, achieving byte-for-byte reproducibility in highly dynamic environments (such as Node.js or Python) often requires extreme friction that fundamentally degrades the velocity advantages promised by an agentic software factory.

Sources

The architectural guidelines, schemas, and policy recommendations synthesised in this report rely heavily on primary security research and official standardisation documentation. The **Supply-chain Levels for Software Artifacts (SLSA) v1.0** specification provides the foundational build-track requirements and provenance schemas.[1] Provenance attestation structures are derived from the official **in-toto Attestation Framework**, specifically leveraging the https://slsa.dev/provenance/v1 predicate type and DSSE envelopes to ensure non-falsifiability.[5] For Software Bill of Materials architecture, analysis integrates capabilities defined in **SPDX 3.0** (including its novel profile architecture for Build and Security) and **CycloneDX 1.6** (for vulnerability exploitability exchange).[11] Artefact distribution and identity linking models are based strictly on the **Open Container Initiative (OCI) Image Specification v1.1.0** and its Referrers API.[2] Package manager security policies regarding lifecycle scripts integrate official **npm configuration guidance** and recognized AppSec incident reports demonstrating the severity of postinstall exploits.[7] Cryptographic signing paradigms utilize established patterns from keyless signing models, OIDC identities, and transparency log anchoring methodologies via Merkle trees, ensuring robust evidence retention without key-custody burdens.[21]

Final synthesis question

What is the leanest software-supply-chain and release-engineering architecture that allows a small agentic factory to produce trusted software rapidly while ensuring that every deployed artefact is linked to reviewed source, controlled dependencies, an isolated build, a valid SBOM, verifiable provenance, an authorised signature, a governed release channel, and a tested rollback path?

The leanest highly secure architecture relies on outsourcing complex cryptographic state

management while enforcing draconian CI/CD discipline. It mandates **immutable OCI artefact digests** as the single source of truth for deployments, utterly eliminating mutable tags. It utilizes **ephemeral container builders** authenticated via **short-lived OIDC workload identities**, removing the need for long-lived secret management. Builds pull dependencies exclusively through an **internal proxy** with global **lifecycle scripts disabled**. Upon build completion, the CI control plane autonomously generates **SPDX/CycloneDX SBOMs** and **in-toto SLSA v1.0 provenance**, signing the package using a **keyless signing infrastructure** anchored to a transparency log. These cryptographic artifacts are attached to the image digest via the **OCI v1.1.0 Referrers API**. Finally, environments are governed by an **admission controller** that strictly rejects any digest lacking these verified signatures, enabling rapid, agent-driven artefact promotion and rollback without ever compromising cryptographic trust or requiring a rebuild.

Works cited

1. SLSA 1.0: Improving Software Supply Chain Security - Cycode, accessed July 27, 2026, https://cycode.com/blog/slsa-1-0-improving-software-supply-chain-security/
2. OCI Image and Distribution Specs v1.1 Releases - Open Container Initiative, accessed July 27, 2026, https://opencontainers.org/posts/blog/2024-03-13-image-and-distribution-1-1/
3. About SLSA, accessed July 27, 2026, https://slsa.dev/spec/v1.0-rc2/about
4. slsa-levels.md - supply-chain-security - GitHub, accessed July 27, 2026, https://github.com/microsoft/hve-core/blob/main/.github/skills/security/supply-chain-security/references/slsa-levels.md
5. SLSA • Provenance, accessed July 27, 2026, https://slsa.dev/spec/v1.0/provenance
6. Grok × ODEI Architecture Exchange, accessed July 27, 2026, https://api.odei.ai/grok/
7. feat(npm): add `ignore-scripts=true` to `.npmrc` for security hardening · Issue #156 - GitHub, accessed July 27, 2026, https://github.com/beatrizsmerino/beatrizsmerino/issues/156
8. Collection of npm package manager Security Best Practices - GitHub, accessed July 27, 2026, https://github.com/lirantal/npm-security-best-practices
9. NPM Security best practices - OWASP Cheat Sheet Series, accessed July 27, 2026, https://cheatsheetseries.owasp.org/cheatsheets/NPM_Security_Cheat_Sheet.html
10. SPDX vs CycloneDX: Choosing the Right SBOM Format for Your Software Supply Chain, accessed July 27, 2026, https://www.herodevs.com/blog-posts/spdx-vs-cyclonedx-choosing-the-right-sbom-format-for-your-software-supply-chain
11. SPDX vs CycloneDX: Comparing SBOM Formats - Safeguard.sh, accessed July 27, 2026, https://safeguard.sh/resources/blog/spdx-vs-cyclonedx-comparing-sbom-formats
12. SPDX 3.0 in Yocto: What Changed and Why It Matters | Sbomify, accessed July 27,

2026, https://sbomify.com/2026/05/19/yocto-spdx-3-0-overview/
13. All about SPDX 3.0 - Interlynk, accessed July 27, 2026, https://www.interlynk.io/resources/all-about-spdx-3-0
14. AI Bill of Materials and Beyond: Systematizing Security Assurance through the AI Risk Scanning (AIRS) Framework - arXiv, accessed July 27, 2026, https://arxiv.org/html/2511.12668v1
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. in-toto attestations - Trustification, accessed July 27, 2026, https://trustification.io/blog/2023/03/13/in-toto-attestations/
17. SLSA, it's all about provenance attestation | by Rémi Rey - Medium, accessed July 27, 2026, https://medium.com/@rrey94/slsa-its-all-about-provenance-attestation-09a83b7b9de7
18. in-toto and SLSA, accessed July 27, 2026, https://slsa.dev/blog/2023/05/in-toto-and-slsa
19. What Is In-toto Attestation? A Practical Guide - Safeguard, accessed July 27, 2026, https://safeguard.sh/resources/blog/what-is-in-toto-attestation
20. Code Signing Infrastructure & Supply Chain Security - SecurePeak, accessed July 27, 2026, https://securepeak.com/crypto/codesign.php.html
21. Sigstore Cosign Keyless Signing with GitHub Actions OIDC: Complete Guide - QCecuring, accessed July 27, 2026, https://www.qcecuring.com/blog/sigstore-cosign-keyless-github-actions
22. Threat Model - Sigstore, accessed July 27, 2026, https://docs.sigstore.dev/about/threat-model/
23. The Sigstore Trust Model - Dan Lorenc - Medium, accessed July 27, 2026, https://dlorenc.medium.com/the-sigstore-trust-model-4b146b2ecf2c
24. OCI Repositories - JFrog Docs, accessed July 27, 2026, https://docs.jfrog.com/artifactory/docs/oci-repositories
25. Trust Architecture - PrivacyAutomated.ai, accessed July 27, 2026, https://privacyautomated.ai/trust-architecture.html
26. anvildevxyz/craft-trails - GitHub, accessed July 27, 2026, https://github.com/anvildevxyz/craft-trails