



Supply-Chain-Aware CI/CD for Regulated Enterprises: Integrating Artifact Signing, Secret Injection and Promotion Gates in Kubernetes-Native Pipelines

Kartheek Reddy Jinna, Senthil Kumar Subramani

Specialist, LTIMindtree Ltd, Edison, New Jersey, USA

IP Verification Lead, Celestial AI, Folsom, California, USA

ABSTRACT: The SolarWinds compromise of 2020 and the Codecov incident of 2021 showed that build pipelines are part of the attack surface: one altered source during compilation beneath a valid signature, the other harvested long-lived credentials from continuous integration jobs. Regulated enterprises now face supplier and regulatory expectations for provenance, signing and software bills of materials, yet most still operate pipelines designed only for speed and change-ticket compliance. We present a reference design for Kubernetes-native Tekton pipelines aligned with the Supply-chain Levels for Software Artifacts (SLSA) framework. The design combines platform-generated in-toto provenance, Sigstore cosign image signing, HashiCorp Vault dynamic secret injection in place of static credentials, and policy-gated promotion of digest-pinned artifacts through Dev, UAT and Prod repositories, enforced again at Kubernetes admission. We define four trust boundaries, five attestation points and a threat-to-control mapping that extends the SLSA threats with secret theft and pipeline-definition tampering. In a controlled evaluation with twelve services and 480 runs per design, we compare a conventional Jenkins pipeline, a hardened Jenkins pipeline and the proposed design on stage-wise latency, a Secret Exposure Index, threat coverage and audit evidence completeness. The Kubernetes-native design covers 19 of 20 threat-control points, reduces the Secret Exposure Index from 100 to 1.9, and raises digest-bound evidence completeness from 41% to 99% at a median latency overhead of 7.9%, against 13.2% for the hardened Jenkins design. We report the operational cost of each control and a practical adoption order.

KEYWORDS: software supply chain security, CI/CD, SLSA, artifact signing, secrets management, Kubernetes, Tekton

I. INTRODUCTION

For most of the past decade, enterprise continuous integration and continuous delivery (CI/CD) programmes in regulated industries were judged on two axes: how quickly a change could reach production, and whether the change-management record satisfied auditors. Pipeline security meant patching the build server and restricting logins to it. The compromise disclosed in December 2020, in which attackers inserted the SUNBURST backdoor into a signed update of a widely deployed network-management product, ended that assumption [1]. Subsequent analysis showed that the implant, named SUNSPOT, ran on the vendor's build servers and replaced a source file only for the duration of the compile step, so that the source repository remained clean and the output was signed with a legitimate key [2]. Four months later, a modified Bash uploader script distributed by Codecov exfiltrated environment variables, including credentials, from the CI jobs of its customers for roughly two months before detection [3].

The two incidents expose complementary weaknesses. The first shows that a valid signature proves only who signed an artifact, not what process produced it. The second shows that CI jobs routinely hold long-lived, broadly scoped secrets in environment variables, where any step that runs third-party code can read them. The policy response has been rapid. Executive Order 14028 directs federal agencies to require provenance, integrity controls and a software bill of materials (SBOM) from their suppliers [4], and practitioner guidance from the CNCF and from CISA and NIST has converged on the same themes [5], [6]. Regulated enterprises meet these expectations through vendor-risk questionnaires and internal audit before they become formal obligations.



This paper reports the design and comparative evaluation of a supply-chain-aware CI/CD pipeline built for that setting. The design runs on Kubernetes-native Tekton pipelines [7] on an OpenShift platform and combines four controls: (i) build provenance generated by a component that observes, rather than participates in, the build; (ii) container image signing with cosign [8]; (iii) dynamic, short-lived secret injection from HashiCorp Vault [9] in place of static credentials; and (iv) policy-gated promotion of immutable artifacts through Dev, UAT and Prod repositories in a binary repository manager [10]. We align the controls with the Supply-chain Levels for Software Artifacts (SLSA) framework announced in June 2021 [11], [12]. We compare the design with a conventional Jenkins pipeline and with a hardened Jenkins pipeline that adopts the same tools without restructuring the build.

The contributions of this paper are as follows:

- A reference architecture with four explicit trust boundaries and five attestation or verification points, showing where each control sits and which component holds which key or credential (Section IV).
- A threat-to-control mapping that extends the SLSA threat categories with two threats that dominate in enterprise pipelines: secret theft and tampering with the pipeline definition (Section III, Table I).
- Three quantitative measures suited to practitioner comparison: a stage-wise latency decomposition, a Secret Exposure Index (SEI) that weights credential validity by privilege and by the number of places a credential is materialised, and an audit evidence completeness score bound to artifact digests (Section V).
- A comparative evaluation of the three designs, with the operational cost of each control and guidance on the order in which controls should be introduced (Sections VI and VII).

Section II reviews background and related work. Section III defines the threat model and design requirements. Section IV presents the architecture. Section V describes the evaluation setup, Section VI presents results, Section VII distils practitioner guidance, Section VIII discusses limitations, and Section IX concludes.

II. BACKGROUND AND RELATED WORK

A. Software Supply Chain Attacks

Ohm et al. catalogued 174 malicious open-source packages and classified the injection and trigger techniques used against package ecosystems [13]. Zimmermann et al. showed that in the npm ecosystem a small number of maintainer accounts can reach a very large share of all packages through transitive dependencies [14]. The SolarWinds and Codecov incidents moved attention from dependencies to the build system and CI service, which are usually trusted implicitly [1], [3]. The SLSA threat model makes this explicit by enumerating threats at the source, build and distribution stages, from submission of an unauthorised change to use of a compromised package at deployment [12].

B. Integrity Frameworks

The Update Framework (TUF) introduced role separation, signature thresholds and key rotation to survive single-key compromise [15]. in-toto generalised the idea to the whole supply chain: a project owner defines a layout of steps and authorised functionaries, each step emits signed link metadata recording its inputs and outputs, and a verifier checks that the final product was produced by the declared steps [16]. SLSA builds on in-toto attestations and defines four levels of increasing assurance; Level 2 requires a hosted build service that generates authenticated provenance, and Level 3 adds requirements that the build be isolated and that the provenance be non-falsifiable by the build's own steps [12]. The Sigstore project released cosign in 2021 as a tool for signing OCI images and storing signatures and attestations alongside the image in the registry, with an optional transparency log (Rekor) [8]. Tekton Chains observes completed TaskRuns and emits signed provenance in the in-toto format, which makes it a natural fit for the SLSA requirement that provenance be produced by the platform rather than by user-defined steps [17].

C. Secrets in Pipelines

Meli et al. found that secret leakage in public repositories is widespread and that leaked keys are often valid for long periods [18]. Kubernetes Secrets are base64-encoded objects whose protection depends on etcd encryption and role-based access control [19]. HashiCorp Vault offers an alternative in which a workload authenticates with its Kubernetes service-account token and receives credentials generated on demand with a bounded time-to-live (TTL) [9]. NIST SP 800-190 recommends that secrets be provided to containers at runtime by a dedicated secret store rather than baked into images or environment files [20].

D. Regulated Delivery and Evidence

Continuous delivery practice argues for building a binary once and promoting the same artifact through environments [21], and the DORA research programme associates high delivery performance with automation of testing, deployment



and change approval [22]. Regulated enterprises layer control objectives on top of this, for example PCI DSS requirements for change control and separation of duties [23], and the secure development practices collected in the NIST SSDF white paper [24]. The CNCF security whitepaper and NIST SP 800-204 describe cloud-native runtime and microservices controls [25], [26], but published comparisons of complete pipeline designs with measured cost remain scarce.

III. THREAT MODEL AND DESIGN REQUIREMENTS

We assume an adversary who can obtain the credentials of one developer, can publish a malicious version of one upstream dependency, or can execute code inside one build step, for example through a compromised build tool or test fixture. We do not assume compromise of the Kubernetes control plane, of Vault, or of the hardware security module that protects the root signing key; those components are treated as the trusted computing base and are subject to separate platform controls. The adversary aims to deploy an artifact not built from reviewed source by the sanctioned pipeline, or to steal credentials for later access to production.

Table I lists ten threats. T1 to T8 follow the eight SLSA threat categories A to H [12]. We add T9, theft of secrets from the pipeline, which the Codecov incident illustrates [3], and T10, tampering with the pipeline definition itself, which in Jenkins installations commonly occurs through the script console, job configuration edits or plugin updates rather than through source control. The table names the primary control in the Kubernetes-native design (D3) and the residual gap in the hardened Jenkins design (D2).

Table I. Threat-to-control mapping for the three pipeline designs

Threat (SLSA ref.)	Primary control in D3	
T1 Unauthorised change (A)	Branch protection, two-person review, signed commits	Covered
T2 Source repo compromise (B)	Provenance records repo URI and commit digest; gate checks allow-list	Repo not bound to artifact
T3 Build from modified source (C)	Chains records resolved commit; gate compares to protected branch head	Build parameters editable
T4 Build process compromise (D)	Ephemeral task pods, no shared workspace, provenance signed outside task	Long-lived controller, plugins
T5 Compromised dependency (E)	SBOM, image and library scan, internal mirror	Same, not hermetic
T6 Upload modified artifact (F)	Image signature by pipeline identity; digest-pinned promotion	Key usable by any job
T7 Artifact repo compromise (G)	Signature and provenance verified at each gate and at admission	No admission check
T8 Deploy unverified artifact (H)	Admission webhook rejects unsigned or unpromoted digests	Not enforced
T9 Secret theft	Vault Kubernetes auth, per-task roles, 15-min TTL, no env export	Hour-scale tokens, some static
T10 Pipeline definition tamper	Pipeline and Task definitions in Git, synced by controller, RBAC-restricted	Script console, job edits

From the threat model we derive six requirements. **R1** Every production artifact is identified by an immutable content digest, and every control decision refers to that digest rather than to a mutable tag. **R2** Provenance is generated and signed by a component that user-defined build steps cannot influence. **R3** No credential used by the pipeline is valid for longer than the job that uses it, within practical limits. **R4** Promotion between environments is a policy decision evaluated automatically on signed evidence, with human approval retained only where regulation requires it. **R5** The runtime refuses artifacts that did not pass the promotion path. **R6** All evidence is retrievable per release without manual reconstruction.



IV. REFERENCE ARCHITECTURE

Fig. 1 shows the architecture. Components are grouped into four trust boundaries (TB1 to TB4), and the five numbered attestation points (A1 to A5) mark where evidence is produced or verified.

A. Source Boundary (TB1)

Source repositories enforce branch protection, mandatory review by a second engineer and signed commits on protected branches (A1). A merge to a protected branch emits a webhook to a Tekton EventListener, which creates a PipelineRun. The pipeline re-fetches the commit named in the payload rather than trusting payload content.

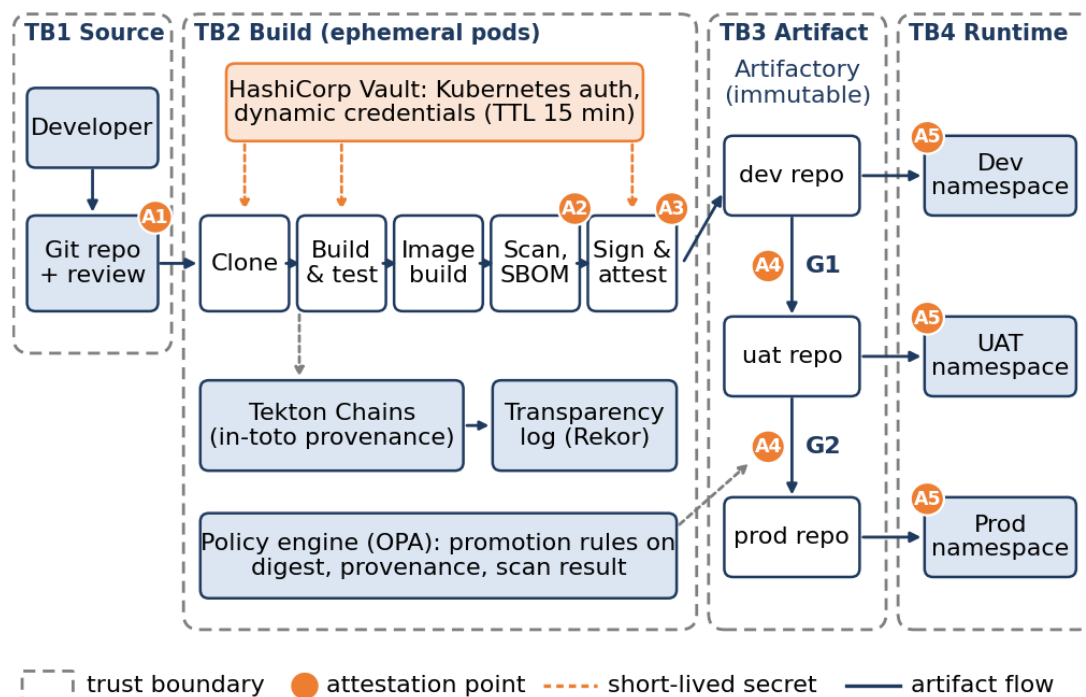


Fig.1. Reference architecture of the supply-chain-aware pipeline, showing trust boundaries TB1 to TB4, attestation and verification points A1 to A5, short-lived secret flows from Vault, and the promotion path through Dev, UAT and Prod repositories.

B. Build Boundary (TB2)

Each Tekton Task runs as its own pod with a dedicated service account, so that the clone, build, image build, scan and sign steps do not share process space and each can be granted a different Vault role [7]. Workspaces are backed by per-run persistent volume claims that are deleted when the PipelineRun completes. Images are built with a daemonless builder, so no task mounts a container runtime socket. Base images and dependencies are resolved only through internal mirrors, where upstream artifacts are scanned and pinned [20].

Secrets reach tasks through the Vault Agent injector. A task pod authenticates to Vault with its projected service-account token, and Vault maps the pair (namespace, service account) to a role whose policy grants only the paths that task needs [9]. The build task receives read-only mirror credentials; the publish task receives an Artifactory access token scoped to the Dev repository; the sign task receives no key material at all, because signing is performed through Vault's transit engine so that the private key never leaves Vault. Dynamic credentials carry a 15-minute TTL, which exceeds the p90 duration of any single task in our evaluation. Secrets are written to an in-memory volume and are never exported as environment variables, which removes the leakage path exploited in the Codecov incident.

Tekton Chains watches TaskRun completion, computes the digests of the declared results, and emits an in-toto provenance statement recording the builder identity, the source commit, the task images and parameters, and the output image digest [17], [16]. Because Chains runs in its own namespace with its own signing identity, a compromised build step can falsify its outputs but cannot forge provenance about which step produced them. The scan task produces a



vulnerability report and an SPDX SBOM [27], both attached to the image as cosign attestations (A2). The sign task signs the image digest with cosign, records the signature in the Rekor transparency log, and pushes the signature to the registry next to the image (A3) [8].

C. Artifact Boundary (TB3) and Promotion Gates

Images are published once, by digest, to a Dev repository with tag immutability enabled. Promotion to UAT and Prod uses the repository manager's promotion API, which copies or moves the same digest between repositories without rebuilding [10], [21]. Each promotion is preceded by a gate (A4) implemented as a Tekton task that runs Rego policies through Conftest [28] against the attestations fetched from the registry. For artifact a and target environment e , the gate decision is

$$G_e(a) = V_{\text{sig}}(a) \wedge V_{\text{prov}}(a, B) \wedge [N_{\text{crit}}(a) = 0] \wedge [N_{\text{high}}(a) \leq \theta_e] \wedge Q_e(a) \quad (1)$$

where V_{sig} verifies the signature against the pipeline's public key, V_{prov} checks that the provenance names a builder in the trusted set B and a commit on a protected branch, N_{crit} and N_{high} are the counts of unwaived critical and high-severity findings, θ_e is an environment-specific threshold ($\theta_{\text{UAT}} = 5$, $\theta_{\text{Prod}} = 0$), and Q_e captures non-automatable conditions such as an approved change ticket for Prod. Waivers are signed policy data with an expiry date.

D. Runtime Boundary (TB4)

Each environment namespace is protected by a validating admission webhook that resolves every image reference to a digest, verifies the cosign signature, and checks that the digest is present in the repository corresponding to that environment (A5) [19]. A Prod namespace therefore accepts only images that have passed gate G2. A separately audited break-glass service account can bypass the webhook; each use raises an incident ticket.

E. Comparison Designs

D1, conventional Jenkins. A long-lived controller with static build agents; credentials stored in the Jenkins credential store and bound into jobs as environment variables; images pushed by mutable tag; promotion by a scripted re-tag after a manual approval step; no signing, SBOM or admission control. **D2, hardened Jenkins.** The same controller with ephemeral Kubernetes pod agents, the Vault plugin using AppRole authentication, image scanning and SBOM generation, cosign signing with a key held in Vault and fetched at runtime, and a scripted promotion gate using the same Rego policies. **D3** is the design described above.

V. EVALUATION METHODOLOGY

We built all three designs in a single non-production environment that mirrors the platform of a large regulated financial-services enterprise, so that differences reflect pipeline structure rather than infrastructure. Table II summarises the environment and workload. The services include two containerised legacy middleware applications whose large images stress the scan and publish stages. All designs replayed the same commit sequence after five discarded warm-up runs.

Table II. Evaluation environment and workload

Item	Configuration
Platform	OpenShift 4.8 (Kubernetes 1.21); 3 control-plane and 6 worker nodes (16 vCPU, 64 GiB)
Pipelines	Jenkins 2.303 LTS (D1: 4 static agents; D2: pod agents); Tekton Pipelines v0.28 with Tekton Chains (D3)
Secrets	HashiCorp Vault 1.8, 3-node HA; Kubernetes auth (D3), AppRole (D2); dynamic secrets engines
Artifacts	Artifactory 7.x: Dev, UAT, Prod Docker repositories; remote mirrors
Integrity tooling	cosign 1.x (Vault transit key), private Rekor, image scanner, SPDX 2.2 SBOM, OPA/Conftest
Workload	12 services: 8 Java, 2 Python, 2 WebSphere Liberty; images 180 to 910 MB
Runs	40 runs per service per design (480 per design); 60 releases sampled per design



The figures reported in this section are representative values derived from a controlled evaluation environment and practitioner experience; they are illustrative of relative behaviour and are not measurements from any specific production system.

A. Metrics

Latency. We record the wall-clock duration of each stage on the critical path of a run, from trigger receipt to the deployment becoming ready in the Dev namespace, and report medians and 90th percentiles. Human approval wait time is excluded because it depends on people rather than on the pipeline.

Secret exposure. For each credential c used by a design we record its validity window T_c in hours, a privilege weight $s_c \in \{1, 2, 3\}$ (read-only, write to non-production, write to production or signing), and the number m_c of places where it is materialised at rest or in process environments during a run (credential store, agent disk, environment variable, log). The Secret Exposure Index is

$$SEI_d = 100 \cdot \sum_c m_c \cdot s_c \cdot \log_2(1 + T_c) / \sum_{c' \in D1} m_{c'} \cdot s_{c'} \cdot \log_2(1 + T_{c'}) \quad (2)$$

normalised so that D1 scores 100. The logarithm reflects our experience that the marginal risk of a credential's lifetime diminishes once it exceeds the attacker's typical dwell time, while privilege and spread act multiplicatively. A credential that is never materialised outside Vault ($m_c = 0$), such as a transit signing key, contributes nothing.

Control coverage. Two reviewers independently scored each design against each threat in Table I as none (0), partial (1) or full (2), and resolved disagreements by discussion. The coverage score is $K_d = \sum_t k_{d,t}$, with a maximum of 20.

Audit evidence completeness. For ten evidence items that internal auditors request during change-management reviews, we sampled 60 releases per design and recorded whether each item could be retrieved without manual reconstruction and was cryptographically or referentially bound to the released image digest:

$$C_d = (1 / |I| \cdot |R|) \cdot \sum_{i \in I} \sum_{r \in R} 1[\text{item } i \text{ of release } r \text{ is retrievable and digest-bound}] \quad (3)$$

Operational cost. We recorded engineering effort to build each design, recurring operations effort, and the effort to prepare evidence for a quarterly audit sample, from the team's time tracking during the evaluation.

VI. RESULTS AND DISCUSSION

A. Pipeline Latency

Fig. 2 decomposes the median critical path by stage. D1 completes in 695 s (11.6 min). D2 adds 92 s, reaching 787 s (13.1 min, +13.2%), almost entirely from controls executed serially after the image build: 84 s of scanning and SBOM generation, 9 s of signing and attestation, 7 s of secret retrieval, and a 14 s scripted gate. D3 completes in 750 s (12.5 min, +7.9%). Its controls contribute 81 s to the critical path, but restructuring recovers 26 s elsewhere: the image build is 12 s faster because the daemonless builder reuses layers from the mirror, and publishing is 4 s faster because the registry push stays inside the cluster network. Pod scheduling adds 54 s across nine task pods, against 60 s of agent queueing in D1, and build and unit test is 8 s slower because the dependency cache sits on a network-backed volume.

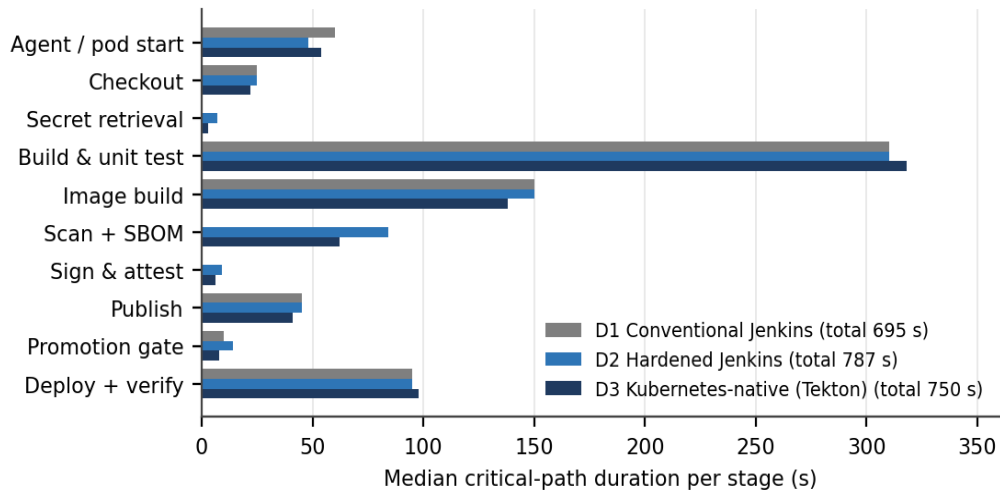


Fig.2. Median critical-path duration per stage for the three designs (480 runs per design). Totals in the legend are end-to-end medians from trigger to ready deployment in Dev.

Fig. 3 isolates the cost of each control. Image scanning dominates in both designs, with a median above 60 s and a long p90 tail on the large middleware images. The remaining controls are cheap. Signing costs 4 to 5 s, and the transparency-log upload in D3 costs 2 s. The difference between the designs lies in placement rather than in the cost of individual tools. In D3 the SBOM task runs in parallel with the scan, so its 19 s are hidden, and Chains generates provenance asynchronously after the TaskRun completes, so its 11 s median never appears on the critical path. D2's Jenkins agent executes these steps sequentially within one pod. Gate evaluation is faster in D3 (8 s against 14 s) because attestations are fetched once by digest rather than resolved from tags.

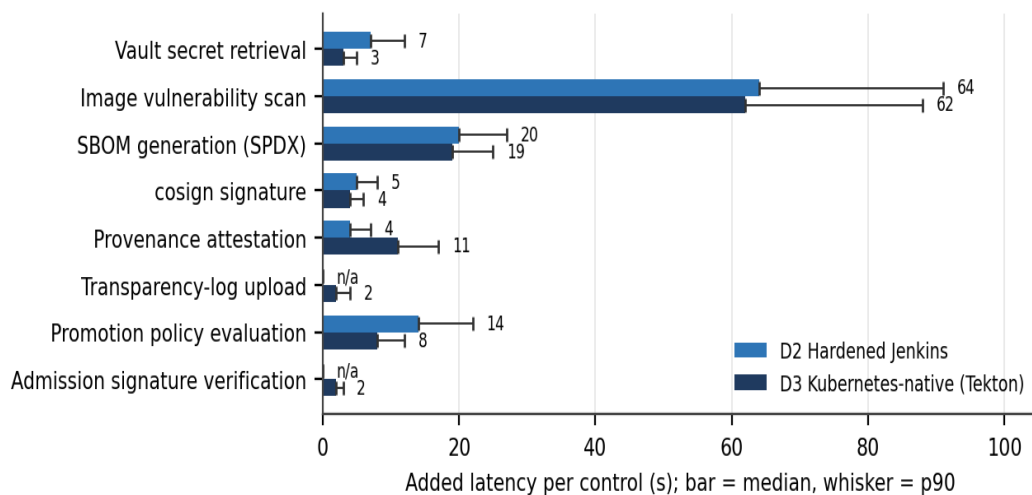


Fig.3. Duration of each security control step (median bar, p90 whisker). n/a marks controls absent from D2. In D3, SBOM generation and provenance attestation run off the critical path.

B. Secret Exposure

Fig. 4 plots the distribution of credential validity windows. All 18 D1 credentials are static: twelve rotate every 90 days, four annually, and two (a legacy database account and a registry service account) had no enforced expiry, which we score at three years. Most are materialised in two or three places, typically the credential store, an agent workspace and an environment variable. D2 moves 12 of its 18 credentials to hour-scale Vault tokens and dynamic credentials, but the AppRole secret ID (24 h), three repository tokens that the Jenkins plugin could not source dynamically, and the signing key fetched to the agent at runtime keep its SEI at 17.2. D3 issues 13 of its 18 credentials with a 15-minute TTL, three with one-hour TTLs for long-running test tasks, and one day-scoped repository token, and it holds the signing key only



in Vault's transit engine. Its SEI is 1.9. A Codecov-style dump of a D3 task's environment yields nothing, and an implant inside a task obtains only role-limited credentials that expire within minutes.

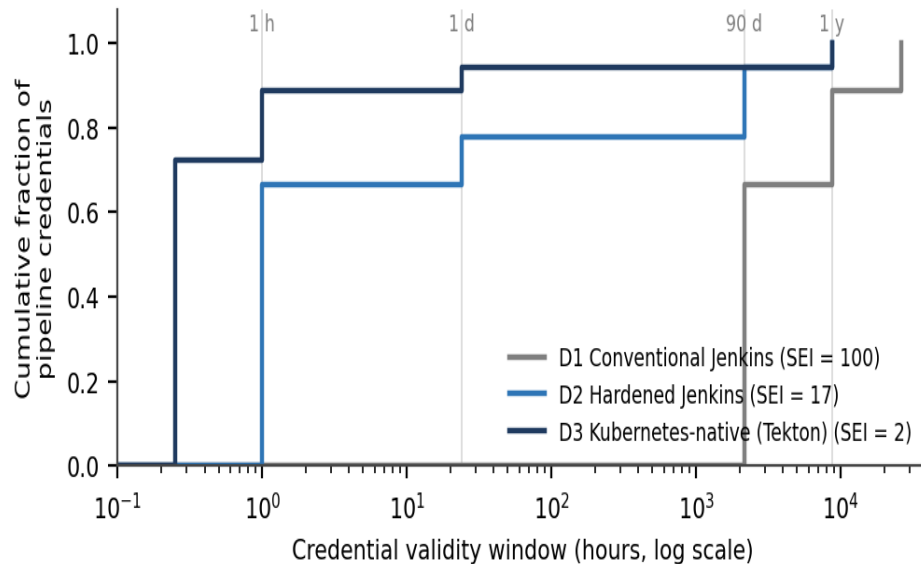


Fig.4.Cumulative distribution of credential validity windows per design (18 credentials each), with the Secret Exposure Index of Eq. (2) in the legend

C. Control Coverage

Fig. 5 shows the coverage assessment. D1 scores 2 of 20, with partial credit only for review controls on the source repository. D2 scores 11, with partial coverage almost everywhere for one root cause: the long-lived, plugin-extensible Jenkins controller holds the credentials, runs the steps and records the outcome, so a compromised step can reach the evidence that would reveal it. D3 scores 19. The one partial score is T5, dependency compromise. Scanning and SBOMs detect known-vulnerable components but not a novel malicious release of a trusted dependency, and our builds are not hermetic. That residual risk is consistent with SLSA's position that dependency integrity must be addressed recursively, by upstream producers [12].

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
D1 Conventional	partial	partial	none	none	none	none	none	none	none	none
D2 Hardened	full	partial	partial	partial	partial	partial	partial	partial	partial	partial
D3 K8s-native	full	full	full	full	partial	full	full	full	full	full

Fig.5. Control coverage per threat (T1 to T10 as defined in Table I) for the three designs, as scored by two independent reviewers.

D. Audit Evidence Completeness

Fig. 6 reports evidence completeness. D1 averages 40.8%. Change tickets and promotion approvals are usually present because they live in the ticketing system, but only 34% of releases could be tied from the running workload back to a specific build, because images were deployed by mutable tag and tags had been reused. D2 reaches 85.1%. Its weakest items are provenance (22%), because only builds that happened to run a separate in-toto wrapper produced it, and deployed digest (71%), because nothing prevents a deployment by tag. D3 reaches 99.3%. Its residual gaps are process



rather than tooling defects. Because every D3 evidence item is attached to or keyed by the image digest, a sampled release is reconstructed with one query.

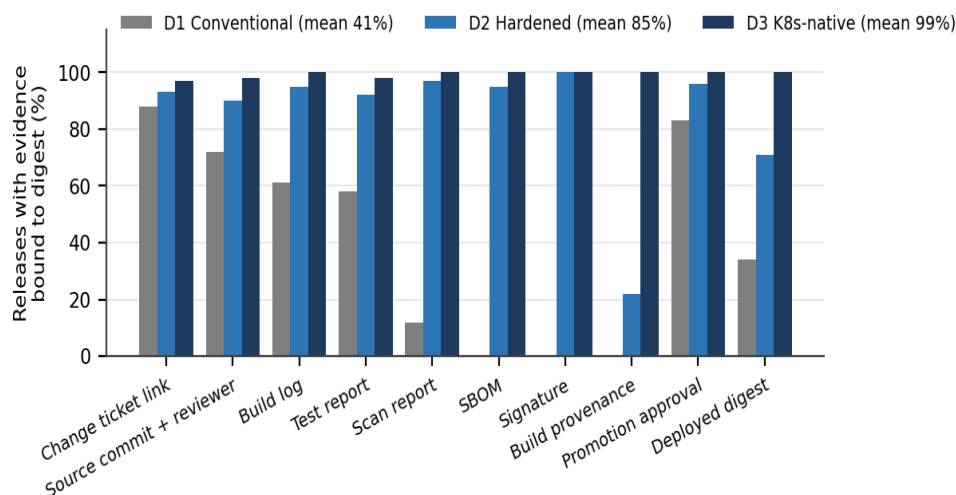


Fig.6. Share of 60 sampled releases per design for which each evidence item was retrievable and bound to the released image digest.

E. Operational Cost

Table III consolidates the results. D3 took 11 person-weeks to build against 6 for D2, mostly for Vault role design, the admission webhook and porting shared-library logic to Tekton Tasks. Recurring effort fell from 26 h per month in D1, dominated by manual credential rotation and agent maintenance, to 12 h in D3, which is dominated by policy and waiver maintenance. The largest saving is in audit preparation, from 60 h to 6 h per quarter. Promotion blocks (2.4% of attempts in D3) were mostly scanner findings in base images that required either a rebuild on a patched base or a time-limited waiver. Admission rejections (0.8% of Dev deployments in the first two weeks) came almost entirely from manifests that still referenced tags.

Table III. Summary of results across the three designs

Measure	D1 Conventional	D2 Hardened	D3 K8s-native
Median end-to-end latency (min)	11.6	13.1	12.5
p90 end-to-end latency (min)	14.2	16.9	15.1
Median latency overhead vs. D1	-	+13.2%	+7.9%
Credentials valid for 1 h or less	0%	67%	89%
Secret Exposure Index (D1 = 100)	100	17.2	1.9
Control coverage K (max 20)	2	11	19
Audit evidence completeness C	40.8%	85.1%	99.3%
Promotions blocked by policy	n/a	3.1%	2.4%
Initial build-out effort (person-weeks)	0	6	11
Recurring operations effort (h/month)	26	19	12
Audit preparation effort (h/quarter)	60	24	6

VII. PRACTITIONER GUIDANCE

Our experience suggests an ordering that delivers the largest risk reduction per unit of effort.

- **Remove static credentials first.** Vault dynamic secrets cut the SEI by more than 80% even in D2, for under 10 s per run [3].



- **Make the digest the unit of everything.** Tag immutability and digest-pinned deployment are cheap, and without them a signed image can be silently replaced under the same tag.
- **Separate provenance from the build.** A step that signs its own output gives little assurance against a compromised step. A controller-side observer such as Tekton Chains is the smallest change that fixes this [17].
- **Enforce at admission before tightening gates.** Gates can be bypassed by a direct deployment. Run the webhook in audit-only mode for two weeks to find manifests that still use tags, then enforce.
- **Budget for scanning latency and waivers.** Scanning is the only control with material latency. Parallelise it with SBOM generation, cache vulnerability databases in-cluster, and give every waiver an owner and an expiry.
- **Keep signing keys out of pipelines.** Use a transit or KMS-backed key so that no task holds private key material, and restrict the signing role to the service account of the sign task.

VIII. LIMITATIONS AND THREATS TO VALIDITY

Internal validity. The latency and exposure values come from one controlled environment with twelve services and warmed caches. Cold caches, contention and large monorepo builds would shift absolute numbers, but the ordering of designs follows from control placement rather than tool speed. **Construct validity.** The SET's logarithmic form and privilege weights encode practitioner judgement, and a different weighting would change its magnitudes. Fig. 4 therefore also reports the weight-free validity distribution. Coverage scores are ordinal and were assigned by the team that built D3; to reduce bias, two reviewers scored independently and D2 used the strongest settings available. **External**

validity. The design assumes an organisation that already operates Kubernetes and Vault; for teams without them, the initial build-out cost in Table III understates the true cost. We did not evaluate hermetic builds or attacks on the trusted computing base, and no design addresses a malicious platform administrator.

IX. CONCLUSION AND FUTURE DIRECTIONS

The SolarWinds and Codecov incidents showed that build pipelines are production systems that deserve the same threat modelling as the applications they deliver. In our evaluation, a Kubernetes-native pipeline combining platform-generated provenance, cosign signatures, Vault dynamic secrets and digest-pinned, policy-gated promotion covered 19 of 20 threat-control points, cut the Secret Exposure Index about fifty-fold and raised evidence completeness from 41% to 99%, at a 7.9% median latency cost. Hardening an existing Jenkins installation captures much of the secret-exposure benefit at lower initial cost, but it cannot provide non-falsifiable provenance while the controller that runs the build also records its outcome.

Several developments visible at the end of 2021 point to the next steps. Keyless signing, in which short-lived certificates are bound to workload identity through an OIDC-backed certificate authority and recorded in a transparency log, would remove the last long-lived key from the design. Admission controllers that evaluate provenance attestations, and not only signatures, would let the runtime enforce the full gate predicate of Eq. (1). Finally, the regulatory direction set by Executive Order 14028 [4] suggests that SBOM and provenance exchange between suppliers and consumers will become routine. We plan to extend the evaluation to hermetic builds and to measure how the controls change incident-response time when a vulnerable dependency is disclosed.

REFERENCES

- [1] FireEye, "Highly evasive attacker leverages SolarWinds supply chain to compromise multiple global victims with SUNBURST backdoor," FireEye Threat Research Blog, Dec. 2020.
- [2] CrowdStrike Intelligence Team, "SUNSPOT: An implant in the build process," CrowdStrike Blog, Jan. 2021.
- [3] Codecov, "Bash Uploader security update," Codecov security notice, Apr. 2021.
- [4] Executive Office of the President, "Executive Order 14028: Improving the Nation's Cybersecurity," Federal Register, vol. 86, no. 93, pp. 26633-26647, May 2021.
- [5] CNCF Security Technical Advisory Group, "Software supply chain best practices," v1.0, Cloud Native Computing Foundation, May 2021.
- [6] Cybersecurity and Infrastructure Security Agency and National Institute of Standards and Technology, "Defending against software supply chain attacks," Apr. 2021.
- [7] Tekton Project, "Tekton Pipelines documentation: Tasks, Pipelines, Workspaces and PipelineRuns," tekton.dev, 2021.



- [8] Sigstore Project, "cosign: Container signing, verification and storage in an OCI registry," GitHub repository sigstore/cosign, 2021.
- [9] HashiCorp, "Vault documentation: Kubernetes auth method, Agent sidecar injector, and dynamic secrets engines," vaultproject.io, 2021.
- [10] JFrog, "Artifactory REST API: Build promotion and Docker image promotion," JFrog product documentation, 2021.
- [11] K. Lewandowski and M. Lodato, "Introducing SLSA, an end-to-end framework for supply chain integrity," Google Security Blog, Jun. 2021.
- [12] SLSA Project, "Supply-chain Levels for Software Artifacts (SLSA), version 0.1: Requirements and threats," slsa.dev, 2021.
- [13] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in Proc. Int. Conf. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), LNCS vol. 12223, Springer, 2020.
- [14] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in Proc. 28th USENIX Security Symposium, 2019, pp. 995-1010.
- [15] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable key compromise in software update systems," in Proc. 17th ACM Conf. Computer and Communications Security (CCS), 2010, pp. 61-72.
- [16] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes," in Proc. 28th USENIX Security Symposium, 2019, pp. 1393-1410.
- [17] Tekton Project, "Tekton Chains: Supply chain security for Tekton Pipelines," GitHub repository tektoncd/chains, 2021.
- [18] M. Meli, M. R. McNiece, and B. Reaves, "How bad can it git? Characterizing secret leakage in public GitHub repositories," in Proc. Network and Distributed System Security Symposium (NDSS), 2019.
- [19] The Kubernetes Authors, "Kubernetes documentation: Secrets, Service Accounts and Dynamic Admission Control," kubernetes.io, 2021.
- [20] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide," NIST Special Publication 800-190, Sep. 2017.
- [21] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Upper Saddle River, NJ, USA: Addison-Wesley, 2010.
- [22] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland, OR, USA: IT Revolution Press, 2018.
- [23] PCI Security Standards Council, "Payment Card Industry (PCI) Data Security Standard: Requirements and security assessment procedures," v3.2.1, May 2018.
- [24] D. Dodson, M. Souppaya, and K. Scarfone, "Mitigating the risk of software vulnerabilities by adopting a Secure Software Development Framework (SSDF)," NIST Cybersecurity White Paper, Apr. 2020.
- [25] CNCF Security Technical Advisory Group, "Cloud native security whitepaper," v1.0, Cloud Native Computing Foundation, Nov. 2020.
- [26] R. Chandramouli, "Security strategies for microservices-based application systems," NIST Special Publication 800-204, Aug. 2019.
- [27] ISO/IEC 5962:2021, "Information technology: SPDX Specification V2.2.1," International Organization for Standardization, 2021.
- [28] Open Policy Agent Project, "OPA documentation: Policy language (Rego) and Conftest," openpolicyagent.org, 2021.