software

MDPI

Article

Integrating Continuous Compliance into DevSecOps Pipelines: A Data Engineering Perspective

Aleksandr Zakharchenko

Independent Researcher, Union, NJ 07083, USA; az68@njit.edu

Abstract

Modern DevSecOps environments face a persistent tension between accelerating deployment velocity and maintaining verifiable compliance with regulatory, security, and internal governance standards. Traditional snapshot-in-time audits and fragmented compliance tooling struggle to capture the dynamic nature of containerized, continuous delivery, often resulting in compliance drift and delayed remediation. This paper introduces the Continuous Compliance Framework (CCF), a data-centric reference architecture that embeds compliance validation directly into CI/CD pipelines. The framework treats compliance as a first-class, computable system property by combining declarative policies-as-code, standardized evidence collection, and cryptographically verifiable attestations. Central to the approach is a Compliance Data Lakehouse that transforms heterogeneous pipeline artifacts into a queryable, time-indexed compliance data product, enabling audit-ready evidence generation and continuous assurance. The proposed architecture is validated through an end-to-end synthetic microservice implementation. Experimental results demonstrate full policy lifecycle enforcement with a minimal pipeline overhead and sub-second policy evaluation latency. These findings indicate that compliance can be shifted from a post hoc audit activity to an intrinsic, verifiable property of the software delivery process without materially degrading deployment velocity.

Keywords: DevSecOps; continuous compliance; data engineering; Policy-as-Code (PaC); Open Policy Agent (OPA); supply chain security; SLSA; Sigstore; Data Governance; Compliance-as-Code; Data Lakehouse; Regulatory Technology (RegTech)

1. Introduction

The acceleration of digital transformation has fundamentally altered how organizations manage risk, data, and decision-making. The continuous delivery (CD) paradigm has largely solved the challenge of software deployment velocity. However, this progress has frequently come at the expense of traceable, continuous compliance. Driven by automation and data, once-discrete domains—infrastructure engineering, software development, data management, and compliance—have converged into a single operational continuum, redefining enterprise control systems and regulatory exposure and turning legacy mechanisms for ensuring governance into a critical bottleneck.

1.1. Problem Statement: The Audit Velocity Mismatch

A core tension exists in the modern enterprise: the mismatch between deployment velocity and audit velocity. While DevSecOps practices enable organizations to deploy code changes hundreds of times per day, the compliance processes designed to govern these changes often remain anchored in snapshot-in-time methodologies.

Academic Editors: Tadashi Dohi, Junjun Zheng and Xiao-Yi Zhang

Received: 17 November 2025
Revised: 18 January 2026
Accepted: 5 February 2026
Published: 10 February 2026

Historically, compliance has been treated as a post hoc, manual audit activity. Evidence collection is frequently reactive, occurring weeks or months after a deployment, which leads to accumulated "compliance debt" and increased regulatory exposure. In regulated sectors such as finance, healthcare, and government, this gap is not merely an operational inefficiency; it represents a systemic governance risk. Traditional governance frameworks emphasize organizational policy and procedural rigor, but lack the operational granularity required for modern, distributed architectures where security and compliance decisions occur at millisecond latency.

1.2. State-of-the-Art and Research Gaps

Recent research has explored multiple dimensions of DevSecOps compliance, including auditability, governance, and security automation. Qureshi and Farooq [1] investigate integrating blockchain technologies into DevOps workflows to improve the immutability and transparency of operational records, demonstrating how distributed ledgers can strengthen audit trails and accountability. Similarly, Pourmajidi et al. [2] propose a reference architecture for governance of cloud-native applications, emphasizing policy enforcement and control mechanisms across distributed systems.

Complementary efforts have examined the semantic and conceptual modeling of security knowledge. Lourenço et al. [3] provide a comprehensive survey of cybersecurity ontologies, semantic log processing techniques, and emerging applications of large language models, highlighting the role of structured knowledge representations in security analysis and governance. From a practitioner perspective, Chittala [4] discusses automating security checks within DevSecOps pipelines, focusing on toolchains and best practices for integrating security scanning into CI/CD workflows.

While these contributions address important aspects of DevSecOps compliance, they largely treat governance, auditability, and security validation as loosely coupled concerns. Existing approaches typically emphasize either immutable logging, high-level governance architectures, semantic classification, or automated security checks, but stop short of defining a unified, end-to-end model for managing compliance evidence across the full software lifecycle.

Three gaps emerge across the current literature:

1. **Lifecycle-Wide Evidence Correlation:** Prior work records or governs individual events or artifacts, but provides limited architectural guidance for correlating source code changes, build outputs, security scans, policy decisions, and deployment state into a single, continuous compliance record.
2. **Queryable, Time-Indexed Compliance State:** Although audit logs and governance controls are well studied, most approaches rely on static records or retrospective analysis. There is a lack of architectures that support real-time or historical querying of compliance state across pipeline stages and runtime environments.
3. **Unified Treatment of Heterogeneous Artifacts:** SBOMs, static analysis results, infrastructure scans, and attestations are typically generated and stored independently. The literature does not yet converge on a canonical data abstraction that models these outputs as a cohesive, versioned data stream suitable for systematic compliance analytics.

As a result, establishing a persistent "Golden Thread" that links compliance evidence from code commit through build, deployment, and operation remains an open challenge. The Continuous Compliance Framework (CCF) proposed in this paper addresses this gap by treating compliance as a data engineering problem, unifying policy enforcement, evidence collection, provenance correlation, and audit queryability within a single architectural model. Table 1 summarizes representative academic approaches to DevSecOps compliance

and governance and contrasts them with the proposed Continuous Compliance Framework (CCF). The comparison highlights differences in architectural scope, evidence handling, enforcement modality, and auditability, clarifying the specific research gap addressed by this work.

Table 1. Comparative Analysis of DevSecOps Compliance and Governance Frameworks.

Dimension	Qureshi and Farooq (2024)–ChainAgile [1]	Pourmajidi et al. (2025)–Cloud-Native Governance [2]	Lourenço et al. (2025)–Security Ontologies Survey [3]	Chittala (2024)–DevSecOps Security Frameworks [4]	CCF
Primary Focus	Agile DevOps coordination using blockchain-backed logs	Governance reference architecture for cloud-native systems	Conceptual survey of security ontologies and semantic processing	Automation of security checks in DevSecOps pipelines	Continuous compliance as a data engineering problem
Scope of Compliance	Process integrity and traceability of DevOps actions	Policy and governance control points	Conceptual mapping of security knowledge	Security controls and pipeline hardening	End-to-end regulatory, security, and governance compliance
Pipeline Integration	Partial (log anchoring post-events)	Architectural guidance, not executable	Not applicable (survey)	CI/CD-stage security automation	Native, continuous CI/CD integration
Evidence Collection Model	Immutable logs via blockchain	Event and control abstractions	Conceptual discussion of semantic logs	Tool-specific scan outputs	Canonical event model for heterogeneous artifacts
Historical State Reconstruction	Limited (log inspection)	Conceptual (no concrete mechanism)	Discussed conceptually	Not addressed	Deterministic time-travel queries via lakehouse
Queryability of Compliance State	Manual or ad hoc log analysis	Not explicitly supported	Discussed at a conceptual level	Tool-dependent, siloed	Unified, SQL-accessible compliance data product
Artifact Unification	No	Partial (conceptual abstractions)	Ontological classification	No	Yes (SBOMs, scans, attestations as versioned data)
Automated Policy Enforcement	Not addressed	Governance guidance, not enforcement	Not applicable	Rule-based security checks	Data-driven, lineage-aware PaC enforcement
Audit Readiness	Improves log integrity	Conceptual governance support	Theoretical	Limited to security findings	Auditable-by-default with cryptographic attestations
Validation Approach	Conceptual + illustrative examples	Architectural modeling	Literature survey	Descriptive framework	End-to-end experimental validation
Key Limitation	Lacks active enforcement and data integration	No concrete data model or evaluation	No operational architecture	Siloed security focus	-

1.3. Proposed Solution: The Continuous Compliance Framework (CCF)

To address these limitations, this paper positions compliance as a first-class engineering concern, framing it as a non-functional requirement (NFR) to be architected, tested, and monitored with the same rigor as performance or security. This work proposes a Continuous Compliance Framework (CCF) built on a data engineering foundation. In this context, "data engineering" refers specifically to the implementation of automated metadata lineage, the adoption of lakehouse storage patterns for versioned evidence, and the transformation of compliance state into a high-fidelity queryable asset.

This model shifts compliance validation from a peripheral, external process to an automated, intrinsic function of the software lifecycle, from the `pre-commit` hook to runtime observation. The central thesis is that verifiable compliance is not a document, but a queryable, real-time data stream. By treating compliance artifacts (SBOMs, scan results, policy decisions, attestations) as time-series data, and the compliance state as a queryable data lakehouse, we can move from post hoc manual auditing to real-time, automated assurance.

1.4. Research Questions and Contributions

To explore the validity of this approach, this research addresses the following Research Questions (RQs):

- **RQ1:** How can compliance be represented as a computable, continuously verifiable system property?
- **RQ2:** How can heterogeneous DevSecOps artifacts be unified into a queryable compliance provenance model?
- **RQ3:** What architectural trade-offs emerge when compliance is enforced pre-, intra-, and post-deployment?

Building upon these questions, this paper provides the following primary contributions:

1. **A Three-Plane Continuous Compliance Framework (CCF):** A conceptual and operational reference architecture that decouples policy definition, data instrumentation, and cryptographic validation.
2. **Compliance Data Lakehouse and Multi-dimensional Lineage Graph:** A novel data engineering pattern for persisting heterogeneous compliance events in a versioned, queryable format that maintains the "Golden Thread" of provenance across the software supply chain.
3. **End-to-End DevSecOps Integration Pattern:** A practical blueprint for embedding automated validation gates into CI/CD pipelines using Policy-as-Code (PaC) and verifiable attestations.
4. **Extensibility to AI/ML Governance:** A demonstration of how the framework scales to address emerging regulatory requirements for model lineage, training data transparency, and algorithmic accountability.

1.5. Methodology and Research Positioning

This paper adopts a Design Science Research (DSR) approach [5] to address the compliance-velocity paradox. The primary contribution is the design and articulation of an architectural artifact, the Continuous Compliance Framework (CCF), and its corresponding reference architecture.

The validation of this framework is architectural and scenario-based rather than hypothesis-testing. The utility of the CCF is demonstrated through a detailed implementation scenario involving a regulated microservice. In addition, the empirical evaluation presented in Section 6 provides scoped measurements of pipeline performance, ingestion overhead, and query latency to establish a baseline for the technical feasibility of

the Compliance-as-Data paradigm. Threats to validity and limitations of this evaluation approach are discussed in Section 6.

1.6. Paper Structure

The remainder of this paper is organized as follows. Section 2 provides background on regulatory and engineering anchors. Section 3 details the architectural planes of the Continuous Compliance Framework. Section 4 discusses the data engineering foundation and the Compliance Data Lakehouse pattern. Section 5 presents a practical implementation scenario. Section 6 evaluates the proposed framework through empirical pipeline measurements and validation exercises. Section 7 discusses implications, limitations, and future extensibility, including organizational, architectural, and governance considerations. Finally, Section 8 concludes the paper and summarizes validated contributions and directions for future work.

2. Background and Foundational Technologies

To construct a robust continuous compliance model, it is necessary to first establish its technical and regulatory anchors. This section reviews the target state (the ”what“ of compliance) and the core engineering technologies (the ”how”) that enable its automation.

2.1. Compliance Anchors (The “What”)

The goal of automation is to provide verifiable, high-fidelity evidence that technical controls are in place to satisfy abstract policy requirements. Our framework targets the computable constraints derived from established standards.

- **Regulatory Frameworks:** These provide the high-level objectives. Our model aims to provide the technical evidence for audits against standards like *ISO/IEC 27001* (Information Security Management Systems, or ISMS) [6], *SOC 2* (Trust Services Criteria, specifically Security, Availability, and Confidentiality) [7], *GDPR* (data protection and privacy) [8], and *HIPAA* (data privacy in healthcare) [9]. The framework translates their requirements (e.g., access control, data encryption) into specific, verifiable queries (e.g., ”Show me all database instances that do not have encryption-at-rest enabled”).
- **Domain-Specific Standards:** These provide more concrete technical controls. *PCI-DSS* (Payment Card Industry Data Security Standard) [10] offers explicit rules, such as network segmentation and encryption key management, that are highly amenable to automation [10]. Furthermore, emerging AI governance frameworks (e.g., *NIST AI RMF* [11], *ISO/IEC 42001* [12], *IEEE p3395* [13]) are being treated as technical, data-driven requirements. They mandate controls for data provenance, model lineage, bias detection, and model drift, all of which are fundamentally data engineering problems that should be solved within the pipeline.

While these frameworks differ in scope and regulatory intent, their requirements ultimately reduce to a common set of technical control objectives that can be evaluated through machine-verifiable evidence. To make this translation explicit, Table 2 illustrates how representative controls from widely adopted standards map to concrete compliance artifacts and enforcement points within the proposed continuous compliance framework. This mapping is illustrative rather than exhaustive and serves to anchor subsequent architectural decisions.

Table 2. Mapping Regulatory Controls to CCF Capabilities.

Standard/Regulation	Control Objective	Example Control	CCF Plane	Artifact
ISO/IEC 27001	Change management	Changes must be authorized and traceable	Policy Plane	Rego policy
SOC 2	System integrity	Unauthorized changes prevented	Validation Plane	Admission decision
PCI DSS	Vulnerability management	No critical CVEs in production	Instrumentation Plane	CVE scan, SBOM
NIST AI RMF	Transparency	Training data provenance	Data Lakehouse	Lineage graph
GDPR	Data minimization	No PII logging	Policy Plane	SARIF result

2.2. Engineering Anchors (The "How")

The shift from manual audits to continuous assurance is made possible by a specific set of modern engineering tools and practices. These anchors move compliance from a textual document to executable code.

- **Policy-as-Code (PaC) Engines:** At the heart of automated compliance is the ability to express policies in a declarative, machine-readable format. Instead of a 100-page PDF, the policy becomes a version-controlled code artifact.
- **Open Policy Agent (OPA):** OPA [14] has emerged as the *lingua franca* for this domain. Using its query language, Rego, organizations can write a single policy and enforce it across the entire CI/CD pipeline. The same Rego policy can validate a Terraform plan (Infrastructure-as-Code), a Kubernetes manifest (admission control), and even an API call (microservice authorization).
- **Kubernetes-native Policy:** Tools like Kyverno [15] provide a similar function but are deeply integrated into Kubernetes, allowing for policy-based mutation and validation of resources as they enter the cluster.
- **Supply Chain Security and Attestation:** A compliance report is only as good as the integrity of the data on which it is based. Recent advances in software supply chain security provide the "cryptographic glue" for a verifiable compliance system.
- **SLSA (Supply-level Security for Software Artifacts):** SLSA [16] is a framework, not a tool. It provides a common language for defining the maturity and security of a build pipeline (e.g., "SLSA Build Level 3" requires a non-forgeable, versioned build service). This framework allows compliance to set tangible targets for pipeline security.
- **Sigstore:** The Sigstore project [17] (including tools like Cosign, Fulcio, and Rekor) provides the mechanism for achieving SLSA's goals. It enables "keyless" signing of software artifacts, build steps, and policy results. A build process can cryptographically attest that it passed a vulnerability scan. This signature is then stored in a tamper-evident transparency log (Rekor). This is the critical mechanism for future-proofing audit trails: it provides a non-repudiable, verifiable, and timestamped record that a specific compliance check occurred and passed.
- **In-toto:** This framework [18] defines a layout or "blueprint" of what is supposed to happen in a pipeline. By creating a signed layout, a final product can be verified against the expected steps, ensuring no unauthorized or non-compliant steps (e.g., a test step being skipped) were part of the process. Together, these technologies form a new "compliance fabric", allowing us to build systems that are not just compliant by design but verifiable in operation.

3. The Continuous Compliance Framework (CCF)

Building on the control-to-artifact mapping introduced in Table 1, the following section presents the architecture that operationalizes these mappings within a continuous compliance pipeline.

To contextualize the proposed Continuous Compliance Framework, Table 3 compares the CCF against existing compliance approaches commonly used in practice, including traditional Governance, Risk, and Compliance (GRC) tools, standalone Policy-as-Code frameworks, and CI-only compliance integrations. This comparison highlights that while existing approaches address isolated stages of the software lifecycle, they lack end-to-end provenance, runtime enforceability, and queryable audit evidence. The CCF differentiates itself by unifying pre-deployment, runtime, and post-deployment compliance validation within a single data-centric architecture.

Table 3. Comparison of CCF with Existing Compliance Approaches.

Capability	Traditional GRC	Policy-as-Code (OPA/Kyverno)	CI-Only Compliance	CCF (This Work)
Pre-deploy enforcement	No	Yes	Yes	Yes
Runtime enforcement	No	Partial	No	Yes
End-to-end provenance	No	No	Partial	Yes
Queryable audit evidence	No	No	No	Yes
Cryptographic attestation	No	No	Partial	Yes
AI/ML lineage support	No	No	No	Yes

The Continuous Compliance Framework (CCF) is a layered, abstract model for integrating policy definition, data collection, and enforcement into a cohesive system. It is designed as a federated control model, where centralized governance policies are executed by distributed enforcement points within the CI/CD pipeline and runtime environments. This architecture consists of three logical planes: the Policy Definition and Management Plane, the Instrumentation and Data Collection Plane, and the Validation and Attestation Plane (Figure 1). Taken together, these three planes form a closed-loop compliance system in which policies are specified as code, evidence is continuously generated as structured data, and enforcement decisions are recorded as verifiable attestations.

3.1. Policy Definition and Management Plane (The "Specification")

This plane provides the "source of truth" for all compliance rules. It transforms abstract regulatory requirements (e.g., "data must be encrypted at rest") into specific, computable, and version-controlled logic.

- **Declarative Policy-as-Code (PaC):** This is the core principle. All compliance rules are written in a declarative Domain-Specific Language (DSL), such as Rego (for OPA), Kyverno, or HashiCorp Sentinel. For example, a rule for SOC 2 (CC6.1) requiring infrastructure to be provisioned via approved, version-controlled scripts can be expressed as a policy that rejects any Terraform plan that does not originate from the blessed-for-production `git` repository. Similarly, a PCI-DSS rule for access control can be written to assert that no Kubernetes `deployment.yaml` file mounts a secret as a plain-text environment variable.
- **Policy Lifecycle Management:** Policies are no longer static documents; they are software artifacts. As such, they require their own CI/CD pipeline. This "Policy-CI" pipeline involves the following:

- **Definition:** Translating regulatory text into a human-readable YAML or JSON, which is then compiled into a machine-interpretable rule (e.g., Rego).
- **Testing:** Policies are unit-tested and integration-tested against a corpus of "known-good" and "known-bad" input data (e.g., sample Terraform plans, Dockerfiles, or K8s manifests).
- **Deployment:** Once validated, policies are versioned, packaged (e.g., as an OPA Bundle), and deployed to the various enforcement points (CI runners, K8s admission controllers).
- **Monitoring:** This plane monitors policy effectiveness, such as the frequency of policy failures, to identify and tune rules that may be overly restrictive or permissive.

- **Automated Remediation and Exception Workflows:** This plane also defines the response to a policy violation. While the default response is to fail a pipeline, a mature implementation supports automated remediation (e.g., a Kubernetes mutating webhook that automatically adds a `securityContext` or `readOnlyRootFilesystem = true`). Crucially, it must also provide a technical workflow for handling exceptions. An "exception" is not a verbal agreement; it is a time-bound, auditable, and cryptographically signed attestation (e.g., a JSON object signed by a security officer) attached to a specific artifact, granting it a temporary waiver of a specific policy.

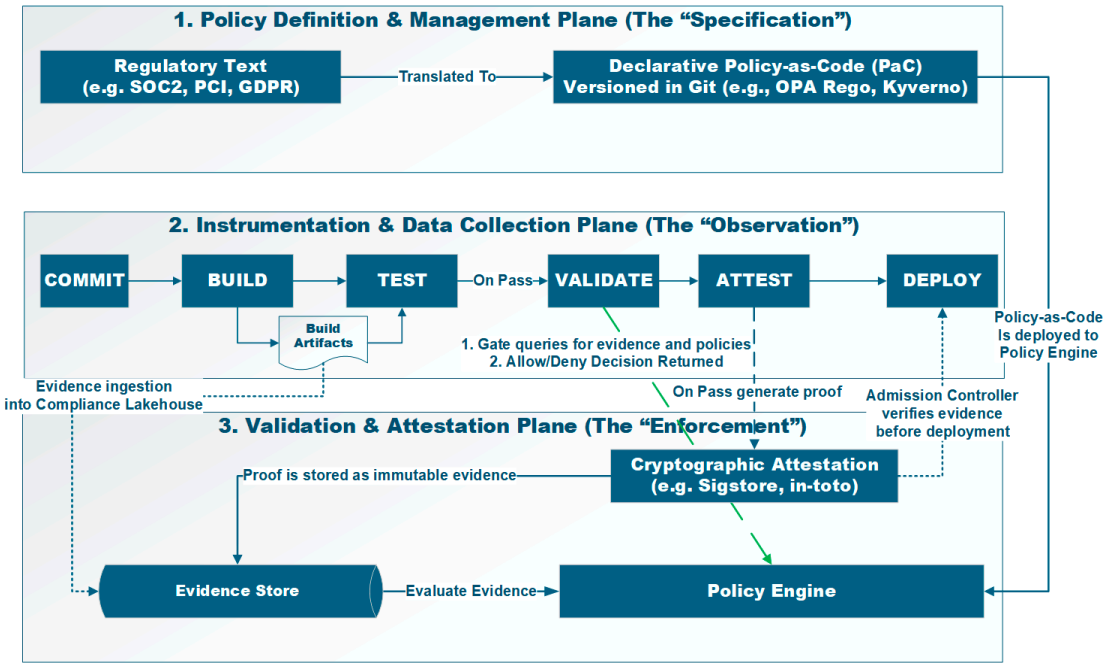

Figure 1. Logical architecture of the Continuous Compliance Framework (CCF). The figure illustrates the three conceptual planes of the Continuous Compliance Framework (CCF), showing the interaction between the Policy Definition Plane, Instrumentation Plane, and Validation Plane, with cryptographic attestation feedback loops. This view emphasizes functional separation of concerns and governance boundaries, independent of specific tool implementations or execution order. The green dashed arrow represents the validation gate interaction with the policy engine, where the gate queries evidence and policies and receives an allow/deny decision.

3.2. Instrumentation and Data Collection Plane (The "Observation")

If the Policy Plane is the "brain", the Instrumentation Plane provides the "senses". This plane is responsible for generating, collecting, and standardizing the vast stream of compliance-relevant data (the "evidence") produced by the DevSecOps lifecycle. This is

not achieved with heavy "agents" but through lightweight hooks, event listeners, and native tool integrations.

- **Artifact Generation and Collection:** This layer instruments the toolchain to *produce* and *collect* standardized, machine-readable artifacts at each stage of the pipeline. Key artifacts include the following:
- **SBOMs (Software Bill of Materials):** CycloneDX or SPDX formats [19] generated during the build stage to create a complete manifest of all first-party and third-party components, including transitive dependencies.
- **VEX (Vulnerability Exploitability eXchange):** A critical companion to SBOMs, VEX documents provide a machine-readable statement about the exploitability of vulnerabilities, allowing the Validation Plane to programmatically filter noise from scanners (e.g., "CVE-12345 is present but not exploitable because the affected code path is not used").
- **Static Analysis Results:** Standardized outputs, such as *SARIF (Static Analysis Results Interchange Format)* [20], from a variety of tools. This includes SAST (e.g., semgrep), DAST, secrets scanning, and Infrastructure-as-Code scanners (e.g., tfsec, checkov).
- **Data-centric Metadata:** This is a crucial, data-engineering-focused component. The framework extends beyond just code and artifacts to include the data itself. This involves the following:
    - **Data Lineage:** Capturing data transformation lineage from tools like dbt [21], providing a verifiable chain of custody from raw data to a business intelligence dashboard.
    - **PII/SPI Scanning:** Integrating data quality and scanning tools (e.g., Great Expectations, soda-core) [22] to scan schemas and data samples, producing metadata that flags PII (Personally Identifiable Information) or SPI (Sensitive Personal Information). This metadata can then be used to trigger policies in the Validation Plane.

3.3. Validation and Attestation Plane (The "Enforcement")

This plane is the "actuator" of the framework. It connects the Policy Plane (the rules) to the Instrumentation Plane (the evidence) to enable real-time decision-making. Its output is not just a log message but a verifiable, non-repudiable cryptographic proof of compliance. This attestation-centric model aligns with emerging supply-chain security guidance, including NIST SP 800-204D, which formalizes integrity verification, trustworthy provenance, and CI/CD pipeline hardening practices [23].

- **Automated Checkpoints (Stage Gates):** These are the CI/CD pipeline jobs (`opa eval...`, `cosign verify...`) that act as automated quality and compliance gates. At each transition (e.g., from `build` to `test`, or `test` to `deploy`), a checkpoint queries the Policy Plane. For example, a pre-deployment gate would take the target Docker image's SBOM (from the Instrumentation Plane) and ask the Policy Plane, "Does this SBOM contain any 'Critical' vulnerabilities (that are not waived via a VEX) or any 'GPL' licensed-dependencies?" The pipeline only proceeds if the policy returns `true`.
- **Cryptographic Attestation Generation:** This is the critical link that transforms compliance from a "check-the-box" activity into a verifiable, data-centric process. When a checkpoint passes, it does more than write a log. It generates a structured metadata object (an "attestation") stating what was proven about which artifact. For example: {''artifact-sha'': ''sha256:abc...'', ''policy-version'': ''sec-policy-v1.4'', ''result'': ''pass'', ''scanner'': ''Trivy-v0.3.1''} This attestation is then cryptographically signed (e.g., using `cosign attest`) and stored,

often in the same registry as the artifact itself or in a transparency log like Rekor. The build artifact now carries an immutable, verifiable proof of its compliance state.

- **Real-time Enforcement (Admission and Runtime):** The framework's enforcement capability extends beyond the CI pipeline into the runtime environment.
- **Kubernetes Admission Controllers:** A K8s admission controller (like Kyverno or OPA Gatekeeper) can be configured to intercept all `deploy` requests. Its policy is not just "is the user authorized?" but "does this container image have a valid, signed attestation from our CI pipeline proving it passed the 'SAST' and 'CVE' policies?" This independently verifies the build-time attestation, creating a Zero-Trust handoff between CI and CD.
- **Service Mesh/ZTA:** In a mature Zero-Trust Architecture (ZTA), this concept extends to service-to-service communication. Using workload identity (e.g., SPIFFE/SPIRE), a service mesh sidecar (e.g., Envoy) can query OPA to enforce policies like, "Only services with a valid, non-expired pci-compliant runtime attestation can access the 'payments' database." This makes compliance a dynamic, continuous property of the running system. Similar mechanisms for behavioral threat detection and automated policy enforcement in DevSecOps environments have been demonstrated in prior research [24].

4. Data Engineering Foundation: The Compliance Data Lakehouse

The Continuous Compliance Framework (CCF) is only as effective as the data foundation upon which it is built. This foundation transforms compliance artifacts from a disparate collection of logs and JSON files into a queryable, time-indexed, and auditable data product. This *Compliance Data Lakehouse* is the data engineering system that enables continuous, real-time auditing. It reframes compliance from an external obligation to an internal property of the system—a "compliance fabric" interwoven with the data plane. This architecture is composed of three main components: a canonical event model for ingestion (Section 4.1), a structured storage layer (Section 4.2), and a graph-based lineage model (Section 4.3), providing the necessary data infrastructure to support the framework's primary functional objective: automated policy enforcement and continuous assurance (Section 4.4), illustrated in Figure 2. Figure 2 complements the logical, plane-oriented view of the framework in Figure 1 by illustrating the concrete data and attestation flows during pipeline execution.

4.1. Canonical Event Model and Ingestion

The first challenge in building a unified compliance data store is handling the volume and variety of data. The Instrumentation Plane (Section 3.2) produces a heterogeneous stream of events: SBOMs (JSON), scan results (SARIF), policy decisions (JSON), attestations (Signed JWTs), build logs (text), and data lineage events (e.g., OpenLineage JSON). To prevent this from becoming an unmanageable "data swamp", a *Canonical Event Model* is required. This is a standardized, version-controlled JSON schema (e.g., based on CloudEvents or a bespoke internal standard) that acts as a common wrapper for all incoming compliance data. Each event includes the following:

- **EventEnvelope:** Contains common metadata: `eventID`, `eventTimestamp`, `sourceTool` (e.g., Trivy, OPA, dbt), `correlationID` (e.g., `git_commit_sha`, `build_id`).
- **EventPayload:** Contains the raw or standardized artifact (e.g., the SARIF object, the CycloneDX SBOM).

This canonical model allows for a decoupled ingestion pipeline. Events are published to an event streaming bus (e.g., Apache Kafka) or a simple object storage bucket (e.g., S3/GCS) with event triggers (e.g., SNS/PubSub). This decoupled, high-throughput inges-

tion mechanism ensures that data collection does not block or introduce latency into the core CI/CD pipelines.

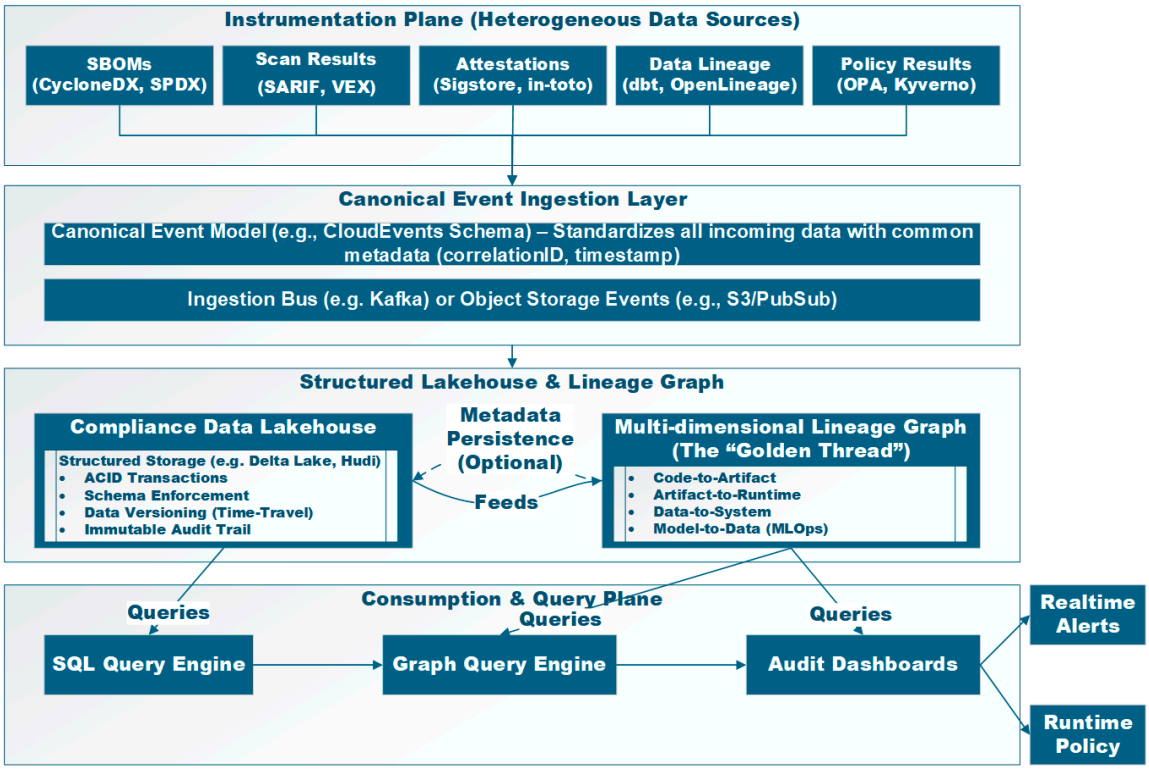

Figure 2. Data and attestation flow within the Continuous Compliance Framework. The Compliance Data Lakehouse Architecture: canonical event ingestion, structured lakehouse layer, multi-dimensional lineage, and policy enforcement. This figure depicts the runtime interactions and data flows among the CI/CD pipeline stages, compliance evidence generation, policy evaluation, and cryptographic attestation. Unlike Figure 1's logical view, this diagram emphasizes execution order, artifact propagation, and feedback loops that enable real-time compliance validation.

The choice of a canonical model is critical for interoperability. While a bespoke internal schema is a valid starting point, adopting open standards for governance metadata is a future-proofing strategy. Standards like *OpenLineage* define a common "governance facet" [25] for datasets and jobs. This standard provides a structured, extensible API for collecting data lineage and metadata from a wide variety of sources (e.g., Spark, dbt, Airflow). By aligning the canonical event model with an open standard such as OpenLineage, the Compliance Data Lakehouse can natively ingest metadata from any compliant tool, drastically reducing integration burden and providing the rich, cross-domain metadata needed to build the Multi-dimensional Lineage Graph (Section 4.3).

4.2. The "Compliance Data Lakehouse" Architecture

The Continuous Compliance Framework (CCF) treats compliance evidence as a first-class data product rather than a collection of transient logs or static reports. To support this paradigm, the framework requires a storage and query layer capable of ingesting the heterogeneous artifacts normalized in the Instrumentation Plane (Section 4.1), preserving their historical states, and enabling deterministic audit queries. This role is fulfilled by the Compliance Data Lakehouse, which serves as the system of record for all compliance-relevant events, artifacts, and attestations.

4.2.1. Architectural Motivation and Design Rationale

Traditional data management approaches are ill-suited for the rigorous requirements of continuous compliance. Relational Database Management Systems (RDBMS) provide strong consistency but struggle with the semi-structured, hierarchical, and evolving schemas characteristic of artifacts such as Software Bills of Materials (SBOMs) and SARIF reports. Conversely, log aggregation platforms (e.g., ELK Stack) excel at high-throughput ingestion but lack the transactional integrity (ACID) and deterministic "time-travel" capabilities required for authoritative audits.

The Data Lakehouse architecture [26] was selected as an optimal compromise. By leveraging open table formats (such as Apache Hudi, Delta Lake, or Apache Iceberg) running on top of standard object storage (e.g., S3 or MinIO), the lakehouse combines the flexibility of a data lake with the governance of a warehouse. This allows the CCF to store massive volumes of raw evidence while providing the schema enforcement necessary for systematic analysis.

4.2.2. Core Architectural Properties for Compliance Data

Modern lakehouse implementations provide three essential properties that transform ephemeral pipeline outputs into durable evidence:

- **ACID Transactions:** Atomic writes ensure that a batch of evidence, such as a set of container vulnerability scans, is either fully committed or not recorded at all. This prevents partial or corrupt data from entering the auditable record, ensuring "all-or-nothing" integrity for every pipeline run.
- **Data Versioning ("Time Travel"):** This is arguably the most critical feature for compliance. The lakehouse maintains a versioned history of table states, allowing an auditor to deterministically query the state of the compliance data as it was at any point in the past. An auditor can ask, "Show me the vulnerability status of service 'X' as it was on 15 January, before the log4j vulnerability was known", and then compare it to the status on 16 January. This provides an immutable, versioned history of the organization's compliance posture.
- **Schema Enforcement and Evolution:** The lakehouse can enforce the canonical event schema to ensure data quality while supporting graceful schema evolution as new tools or artifact types are added. This architecture enables auditors and automated systems to run powerful, high-level SQL queries against the entire compliance record, such as the following:
    - SELECT * FROM compliance_events WHERE timestamp BETWEEN '...' AND '...' AND policy_id = 'pci-dss-3.4' AND result = 'FAIL'
    - SELECT service_name, cve_id FROM vulnerability_reports WHERE severity = 'CRITICAL' AND last_seen_timestamp > (NOW()—INTERVAL '30 days')
    - `SHOW HISTORY FOR/path/to/my/app/sbom` (a command to see all versions of a specific artifact's metadata).

4.3. Multi-Dimensional Lineage Graph (The "Golden Thread")

While the lakehouse serves as a repository of events, the true analytical power of the CCF is the synthesis of these events into a *Multi-Dimensional Lineage Graph*. This graph models the relationships between all compliance entities, providing the "Golden Thread"—the continuous chain of custody that connects a line of code to its runtime execution and the data assets it touches. This graph, which can be stored in a graph database (e.g., Neo4j) or built using the lakehouse tables, must correlate multiple dimensions of lineage [27–29]:

- **Software Supply Chain Lineage (Code-to-Artifact):** Connects a `git_commit_sha` to the `build_id` that processed it, which in turn connects to the `docker_image_sha` (or other binary) it produced.
- **Operational Lineage (Artifact-to-Runtime):** Connects the `docker_image_sha` to the kubernetes_deployment that deployed it, which connects to the `pod_id` and `node_ip` where it is currently running. This allows an operator to instantly answer, "Where is the code from commit `abc123` running right now?"
- **Data Lineage (Data-to-System):** This is the data-engineering-specific dimension. Using metadata from tools like dbt (data lineage) and PII scanners (data classification), this graph connects a database table (e.g., `users`) to the service (e.g., `user-profile-svc`) that consumes it, and to the classification of the data it contains (e.g., `PII: TRUE`).
- **(Future-Proofing) AI/ML Provenance (Model-to-Data/MLOps):** This extends the graph to AI/ML workflows, which is essential for AI compliance (e.g., NIST AI RMF). This connects a specific `model_version: v1.2.3` to the `training_pipeline_run: #456` that produced it, which connects to the `dataset_hash: s3://.../data-v4-hash` it was trained on. This is the only technical way to prove that a model was not trained on non-consensual or biased data.

By unifying these heterogeneous dimensions into a single graph representation, the CCF enables high-order compliance queries that are impossible with siloed tools. For example, an auditor can ask, "Show me all running pods, in production, that were built from code written by a non-full-time employee, and which have access to PII data", and receive a technical, verifiable answer in seconds.

This multi-dimensional lineage unification aligns with broader research emphasizing the centrality of provenance and governance in modern data-driven ecosystems [30]. By establishing this "Golden Thread," the CCF provides the necessary context for the Policy Plane (Section 4.4), which consumes this graph to make automated, context-aware enforcement decisions.

4.4. Automated Policy Enforcement and Continuous Assurance

The final layer of the CCF architecture moves from data persistence to active governance. While the Data Lakehouse and Lineage Graph provide the "memory" and "context" of the system, the Policy Plane serves as its "decision engine". By treating policy as a computable function of the compliance data stream, the framework enables real-time enforcement and cryptographic proof of compliance.

4.4.1. Data-Driven Policy-as-Code (PaC)

In the CCF, policies are defined as declarative code (e.g., using Rego for Open Policy Agent or similar logic languages). Unlike traditional static checks, these policies are data-aware; they do not merely inspect a single local file, but rather query the Compliance Data Lakehouse and Lineage Graph to make holistic decisions.

For instance, a deployment policy can be configured to verify the entire lineage of an artifact before allowing it to run in production. The policy engine performs a "look-back" query to ensure the following:

1. The container image was built from a trusted repository (Supply Chain Lineage).
2. All security scans were completed with no critical vulnerabilities (Instrumentation Data).
3. A valid cryptographic attestation exists for each stage of the pipeline (Verification).

4.4.2. Continuous Assurance and Feedback Loops

The CCF architecture supports two distinct modes of enforcement that ensure a continuous feedback loop between the pipeline and the compliance record:

- **Synchronous Gating (Admission Control):** During pipeline execution or at the point of deployment (e.g., via a Kubernetes Admission Controller), the framework acts as a gatekeeper. If the lineage graph or the lakehouse record indicates a compliance violation, the deployment is blocked, and a "Compliance Failure" event is fed back into the canonical model, notifying the developers immediately.
- **Asynchronous Auditing (Drift Detection):** Because the Lakehouse supports "time-travel" queries (Section 4.2), the framework continuously scans the current runtime state against historical policies. If a new vulnerability (e.g., a Zero-Day) is discovered, the system can retroactively query the "Golden Thread" to identify every running service that originated from the affected codebase, even if those services were compliant at the time of deployment.

4.4.3. Cryptographic Attestation and SLSA Alignment

To ensure the integrity of the compliance record, the CCF integrates with signing frameworks like Sigstore and aligns with SLSA (Supply-chain Levels for Software Artifacts) standards. Each successful policy evaluation generates a signed attestation—a non-falsifiable record that a specific policy was checked against a specific artifact at a specific time.

These attestations are landed back into the Lakehouse as canonical events. This creates a recursive loop of trust: the data proves the compliance state, and the cryptographic signatures prove the integrity of the data. This mechanism transforms the audit process from a manual sampling of logs into a deterministic verification of a signed, continuous evidence stream.

By unifying these components—Canonical Ingestion (Section 4.1), Lakehouse Storage (Section 4.2), Lineage Correlation (Section 4.3), and Automated Enforcement (Section 4.4)—the CCF provides a complete reference architecture for "Compliance-as-Data", fulfilling the research objective of making continuous compliance a verifiable system property.

5. Implementation Example: A Data-Handling Microservice

This section presents an illustrative implementation scenario of the Continuous Compliance Framework (CCF). The objective is not to provide a production-grade benchmark or exhaustive case study, but to demonstrate how the conceptual architecture introduced in Section 3 is operationalized into concrete compliance artifacts, policy evaluations, and cryptographic attestations within a CI/CD pipeline. The scenario is intentionally scoped to a single microservice to isolate control flow, artifact generation, and enforcement semantics.

To illustrate the framework in practice, the following synthetic microservice scenario—which provides tracing of a single change to a `user-profile-api` microservice, a component that handles PII and is subject to GDPR, SOC 2, and internal security policies—is hereby presented. While the scenario focuses on a single microservice, the framework is designed to support multi-service systems through parallel pipeline execution and shared policy and evidence planes, as discussed in Sections 3 and 6.

Scenario: A developer attempts to add a new logging feature that, unknown to them, logs a user's email address.

Policies (Policy Plane): A central, version-controlled repository (`governance-policies`) defines three relevant policies in Rego:

1. **Security Policy (`sec.rego`):** "Container image must have no 'CRITICAL' CVEs (CVE-202X-1234) unless a valid VEX attestation exists."

2. **Infra Policy (`infra.rego`):** "Kubernetes deployments must have a `readOnlyRootFilesystem: true` and must not mount host volumes."
3. **Data Privacy Policy (`data.rego`):** "Static analysis results (SARIF) must not contain any finding with the `pii-logging` tag and a 'HIGH' severity."

Treating compliance logic as version-controlled Policy-as-Code is consistent with established approaches for automated compliance management in cloud environments [31,32].

End-to-End Workflow (CI/CD Pipeline):

5.1. Git Push (Commit and Pre-Flight):

a. A developer pushes code with the new logging statement: `logger.info('User logged in: ' + user.email)`.
b. A pre-commit hook (optional, but recommended) runs a local `semgrep` check. The check fails, blocking the commit as follows:
    i. bash # (Pre-commit hook output) ERROR: PII-LOGGING VIOLATION (data.rego) File: src/handlers.go:L52 logger.info("User logged in: " + user.email)
c. The developer, now aware, refactors the code to: `logger.info("User logged in: " + user.ID)`. This passes the pre-commit check.

This step demonstrates how compliance violations can be detected and remediated at the earliest possible point in the development lifecycle, reducing downstream enforcement and audit overhead.

5.2. Build (Instrumentation and Data Collection Plane): The CI Pipeline (e.g., GitHub Actions, GitLab CI) Executes Several Parallel Jobs:

a. **build-image:** Builds the Docker image user-profile-api:sha-abc123.
b. `scan-sast`**:** `semgrep --sarif...` runs against the source. It finds the `pii-logging` pattern (from the previous commit, now fixed) and flags it as `resolved`. It produces `sast.sarif`.
c. **scan-image:** trivy image --format cyclonedx... generates sbom.cdx. trivy image --format json ... generates cves.json.
d. **scan-iac:** opa eval -d policies/infra.rego -i k8s/deployment.yaml checks the deployment manifest. It produces iac-results.json.
e. `ingest-evidence`**:** All artifacts (`sast.sarif`, `sbom.cdx`, `cves.json`, `iac-results.json`) are standardized into the Canonical Event Model (Section 4.1) and pushed to the Compliance Data Lakehouse. This links them all by the `correlationID: git-commit-sha-abc123`. Emerging AI-assisted lineage extraction techniques can further enhance this step by deriving implicit relationships from heterogeneous metadata sources [29].

This stage illustrates the role of the Instrumentation and Data Collection Plane in generating standardized, machine-readable compliance evidence as a natural byproduct of existing DevSecOps tooling.

5.3. Test-and-Validate (Validation Plane): This Is a Blocking "Stage Gate" Job That Does Not Re-Run Scans, Instead, It Queries the Policy Plane (OPA) Using the Data Just Ingested into the Lakehouse (Figure 3)

In this case, `decision.allow` is `true` because all policies passed (no critical CVEs, IaC is secure, and the PII-logging issue was resolved).

Crucially, this validation step operates purely on previously collected evidence, decoupling policy evaluation from tool execution and enabling deterministic, low-latency enforcement.

```
# Get all evidence for this commit
evidence = lakehouse.query(
   "SELECT * FROM compliance_events WHERE correlationID = 'git-commit-
sha-abc123'"
)

# Ask OPA to make a decision based on the collected evidence
# OPA pulls policies from the Policy Plane
decision = opa.eval(
   input = {
      "evidence": evidence,
      "policies": ["sec.rego", "infra.rego", "data.rego"]
   }
)

# Check the decision and report the outcome
IF not decision.allow:
   PRINT "Pipeline FAILED. Violations: {decision.violations}"
   EXIT(1)
ELSE:
   PRINT "All compliance policies passed."
```

Figure 3. Pseudocode for the validation job.

5.4. Attest-and-Publish (Attestation Plane): Since the Validation Gate Passed, the Pipeline Now Proves It

a. The pipeline generates a JSON attestation:
   i. json { ”predicateType”: ”https://slsa.dev/provenance/v0.2”, ”subject”: [{”name”: ”user-profile-api”, ”digest”: ”sha256: abc123. . .”}], ”predicate”: { ”policy_evaluation”: { ”policy_bundle_version”: ”governance-policies:v1.4.2”, ”result”: ”PASSED”, ”evidence_correlation_id”: ”git-commit-sha-abc123” } } }
b. It then signs this attestation using Sigstore:
   i. bash # Sign the image digest with the attestation cosign attest --key $MY_KEY --type ”slsaprovenance” \ --predicate attestation.json \ https://registry.mycorp.com/user-profile-api@sha256:abc123. . .
c. The signed attestation is pushed to the OCI registry alongside the image, and the signature is recorded in the Rekor transparency log.

The resulting attestation constitutes a non-repudiable, verifiable compliance claim that can be independently validated without re-running scans.

5.5. Deploy (Runtime Enforcement):

a. The developer (or an automated CD tool) submits `k8s/deployment.yaml` to the Kubernetes API.
b. A Kyverno/OPA Gatekeeper admission controller (running in the cluster) intercepts the request.
c. It performs an independent, real-time verification using Sigstore (Figure 4):
d. Because the image `sha256:abc123...` has a valid attestation that matches the policy, the admission controller allows the pod to be scheduled. If an unattested or a failed-policy image were submitted, the API server would reject it.

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
 name: check-image-attestation
spec:
 rules:
  - name: verify-compliance-attestation
    match:
     resources:
      kinds:
       - Pod
    verifyImages:
     - image: "registry.mycorp.com/*"
       attestations:
        - type: "slsa-provenance"
          subject: "registry.mycrorp.com/*"
          predicate:
           policy_evaluation:
            policy_bundle_version: "governance-policies:v1.4.2"
            result: "PASSED"
```

Figure 4. Kyverno Cluster Policy for the example.

This enforcement mechanism establishes a Zero-Trust handoff between CI and CD, ensuring that only artifacts with verifiable compliance provenance may enter runtime environments.

This end-to-end flow demonstrates how the CCF transforms compliance from a manual checklist into a verifiable, automated, and data-driven component of the CI/CD pipeline, creating a Zero-Trust handoff between build and deploy.

6. Experimental Evaluation

The implementation scenario above provides the empirical basis for the evaluation presented in Section 6. While limited in scope, it enables direct measurement of pipeline overhead, artifact generation costs, and attestation latency, allowing the feasibility and performance impact of the proposed framework to be assessed under controlled conditions.

This section evaluates the proposed Continuous Compliance Fabric (CCF) against the dimensions outlined in Table 4. The objective is not to benchmark absolute performance, but to demonstrate architectural feasibility, policy enforceability, provenance completeness, and practical pipeline impact using a minimal, reproducible experimental setup.

Table 4. Pipeline Execution Time Summary (10 Runs).

Stage	Mean (ms)	Min (ms)	Max (ms)	Std Dev (ms)
Build	6862	5799	7557	486
SBOM Generation	28,493	25,494	32,191	2187
Vulnerability Scan	1057	855	1373	163
Policy Evaluation	413	296	524	66
Artifact Ingestion	1917	1434	3395	552
Attestation	2212	1777	2805	345
Total Pipeline	41,287	38,556	44,988	2062

6.1. Experimental Environment

All experiments were executed on a single-node, resource-constrained environment to approximate a conservative lower bound on performance:

- CPU: AMD FX-8120
- Memory: 16 GB DDR3 RAM
- Operating Environment: Local Docker runtime
- Toolchain: Docker (version 29.1.3), Syft (version 1.39.0), Trivy (version 0.68.2 using vulnerability DB version 2 updated at 3 January 2026 00:35:09 +0000 UTC), Open Policy Agent (OPA) (version 1.12.1), Python (version 3.13.2), Cosign (version 3.0.3)
- Execution Model: Serial pipeline execution via shell script

Ten consecutive pipeline runs were executed using an identical configuration. All scripts, policies, artifacts, and raw execution logs are publicly available to support independent reproduction and inspection [33].

6.2. Pipeline Structure and Validation Methodology

The experimental pipeline implements a simplified but end-to-end CCF instance comprising the following stages:

1. Build: Container image construction.
2. Instrumentation: SBOM generation and vulnerability scanning.
3. Policy Evaluation: Declarative compliance evaluation using OPA.
4. Artifact Persistence: Ingestion of compliance artifacts into a queryable store.
5. Attestation: Cryptographic signing of compliance results.

Each stage records wall-clock execution time with millisecond resolution. No artificial delays, mock components, or synthetic data were introduced; all measurements reflect actual tool execution.

This structure directly exercises all three CCF planes:

- Instrumentation Plane: SBOM and vulnerability artifacts,
- Validation Plane: Policy decisions,
- Policy Plane: Rego-based compliance rules, with cryptographic linkage enabling end-to-end provenance.

6.3. Policy Enforceability and Determinism

Compliance policies were expressed as declarative Rego rules operating over structured vulnerability scan output. The evaluated policy denies builds containing disallowed vulnerability classes, producing deterministic allow/deny decisions.

Across all ten runs, policy evaluation produced consistent results, confirming the following:

- Compliance rules are machine-enforceable,
- Policy decisions are deterministic given identical inputs,
- Policy logic remains decoupled from underlying tooling.

This validates the feasibility of treating compliance controls as executable constraints rather than post hoc documentation.

6.4. Provenance Completeness and Auditability

Each pipeline execution generated a complete set of linked artifacts, including the following:

- Container image digest,
- Software Bill of Materials (SBOM),
- Vulnerability scan results,

- Policy evaluation output,
- Cryptographic attestation.

Artifacts were persisted and linked through content-addressable references, enabling direct traversal from a deployed artifact back to the exact policy checks and evidence that validated it.

This demonstrates that CCF enables queryable, end-to-end compliance lineage, rather than static audit snapshots.

6.5. Pipeline Overhead Analysis

Table 4 summarizes execution time statistics across ten experimental runs.

Several observations are notable:

- Instrumentation dominates execution time, primarily due to SBOM generation, consistent with expectations for deep dependency analysis.
- Policy evaluation latency is negligible relative to scanning and build steps.
- Cryptographic attestation adds only seconds, even on legacy hardware.
- Total pipeline overhead remains well under one minute in a fully serial execution model.

In modern CI/CD environments, where scanning and validation are commonly parallelized or cached, the relative overhead of continuous compliance enforcement would be further reduced.

6.6. Audit Readiness and Query Demonstration

Unlike traditional compliance approaches that culminate in static reports, the experimental CCF implementation produces queryable audit evidence. An auditor can issue direct queries such as the following:

- "Which policy evaluations validated this artifact?"
- "What vulnerabilities were present at build time?"
- "Which compliance rules were enforced prior to deployment?"

All such queries are answerable using persisted artifacts without reconstructing historical context or relying on human testimony.

6.7. AI Governance Extensibility

Although the experiment focused on software supply chain controls, the architecture generalizes naturally to AI governance requirements. Lineage concepts used for building artifacts extend directly to the following:

- Training dataset provenance,
- Model version lineage,
- Evaluation and bias metrics,
- Deployment attestations.

This confirms that CCF provides a unified substrate for both traditional security compliance and emerging AI governance obligations.

6.8. Summary

The experimental results validate that Continuous Compliance Fabric is as follows:

- Architecturally sound,
- Operationally feasible,
- Low-overhead, even on constrained hardware,
- Audit-ready by construction,
- Extensible to AI governance use cases.

Most importantly, compliance is demonstrated as a continuous, cryptographically verifiable system property, rather than a periodic, document-driven exercise.

7. Discussion and Implications

Building on the empirical measurements presented in Section 6, this subsection discusses the practical implications of pipeline overhead in real-world DevSecOps environments.

The proposed Continuous Compliance Framework (CCF) represents a shift in engineering culture and tooling, moving from periodic, manual auditing to continuous, automated assurance. This section discusses the implications of the Continuous Compliance Framework (CCF), interpreting the evaluation results in terms of operational impact, organizational benefits, and extensibility to future regulatory and technological challenges.

The discussion below contextualizes the evaluation dimensions summarized in Table 5, focusing on their broader engineering and governance implications rather than repeating empirical results.

Table 5. Evaluation Dimensions and Validation Methods.

Dimension	Validation Method	Evidence Type
Architectural Soundness	Design Analysis	Reference Architecture
Policy Enforceability	Scenario Execution	Policy Decisions
Provenance Completeness	Lineage Inspection	Linked Artifacts
Pipeline Overhead	Scoped Measurements	Timing Metrics
Audit Readiness	Query Demonstration	Sample Queries
AI Governance Extensibility	Conceptual Extension	Lineage Schema

7.1. Performance and Overhead Analysis

A primary concern for any new process integrated into the CI/CD pipeline is its impact on developer velocity. The CCF is designed to parallelize and decouple data collection (which is fast) from data validation (which can be selective).

- **CI Pipeline Latency:** The primary overhead introduced in the "hot path" (the developer's wait time) is the stage gates in the Validation Plane (Section 3.3).
- **Scanning:** Tools like semgrep, trivy, and tfsec are highly optimized and typically complete in seconds to low minutes, not hours. By running these jobs in parallel (as shown in Section 5.2), the added latency is the `max()` of the scan times, not the `sum()`.
- **Data Ingestion:** The decoupled ingestion model (Section 4.1) using an event bus or object storage is asynchronous. The CI pipeline "fires-and-forgets" the evidence artifacts, introducing negligible latency (e.g., <1 sec for an S3 `put` or Kafka `produce`).
- **Validation Gate:** The synchronous gate (Section 5.3) performs a policy query. OPA is designed for high-performance, sub-millisecond decision-making. Hence, the bottleneck is not the policy engine itself, but the query to the data lakehouse. For hot path validation, a "read-after-write" pattern against the lakehouse (or even a simpler key-value store cache) is required.
- **Attestation:** Cryptographic signing (Section 5.4) is extremely fast (milliseconds).
- **Overall Impact:** It is projected that a mature CCF implementation, when properly parallelized, adds less than 5–10% to the total pipeline execution time, an acceptable trade-off for the automated assurance and risk reduction it provides.
- **Data Storage Cost:** The artifacts (SBOMs, SARIF, attestations) are JSON or text and compress well. The data volume per pipeline run is in megabytes, not gigabytes. The storage cost in a cloud object store is minimal. The primary cost is in the query engine

(e.g., Databricks, Snowflake, BigQuery) used to run the Compliance Data Lakehouse, which is a predictable operational expense.

This observation aligns with recent studies noting that many DevSecOps governance approaches introduce operational friction due to coarse-grained enforcement or centralized controls, rather than pipeline-native, data-driven validation mechanisms [1,2]. By decoupling evidence ingestion from synchronous policy evaluation, the CCF demonstrates how continuous compliance can be enforced without materially degrading delivery velocity.

7.2. Qualitative and Quantitative Benefits

The benefits of the CCF extend beyond pure cost savings and into risk reduction and velocity enhancement.

- **Audit Readiness and Cost Reduction:** This is the most immediate quantitative benefit. The time required for auditors to gather *Provided by Client* (PBC) evidence is reduced from weeks of manual data calls (screenshots, interviews) to minutes. The Compliance Data Lakehouse (Section 4.2) and Lineage Graph (Section 4.3) provide a self-service, queryable, and verifiable data store that produces audit evidence on demand. The centrality of provenance and analytics governance in ensuring trustworthy decision-making in data-driven environments is well established in prior research [34].
- **Mean Time to Remediation (MTTR) for Compliance:** The framework acts as a "shift-left" mechanism for compliance. A violation (e.g., PII logging) is caught in the pre-commit or build phase (Section 5.1), not in a quarterly audit six months after deployment. The developer is notified in their native workflow (a failed PR) and can remediate the issue immediately. This reduces the MTTR for compliance-related issues from months to minutes.
- **Reduction in "Compliance Drift":** *Compliance drift* is the divergence of a running system's state from its intended, as-code policy. By extending the CCF into runtime enforcement (Section 3.3) via admission controllers and service mesh policies, the system can prevent drift (e.g., rejecting a non-compliant `kubectl apply`) and detect it in real time.

Prior DevSecOps frameworks and governance models have emphasized automation and tooling integration, but often stop short of treating compliance evidence as a persistent, queryable asset for auditors and risk functions [2,4]. The CCF directly addresses this gap by operationalizing audit readiness as a data engineering problem, rather than a documentation exercise.

7.3. Discussion: Future-Proofing and Advanced Applications

Recent surveys of security semantics and governance mechanisms highlight the growing importance of contextual metadata, lineage, and interpretability in complex software ecosystems, while noting the lack of executable frameworks that integrate these concepts into operational pipelines [3]. The extensibility of the CCF directly responds to these observations by grounding advanced governance use cases in a unified, enforceable data model. The data-centric model is designed to incorporate new classes of risk and regulation without re-architecting.

- **From Blocking to Self-Healing:** The framework can be extended from simply *blocking* non-compliant actions to automatically remediating them. A Kubernetes mutating admission controller, guided by the Policy Plane, can automatically add a required `securityContext`, a `networkPolicy`, or a sidecar proxy to a deployment, enforcing compliance without developer intervention.

- **Runtime Compliance (ZTA):** The model naturally extends to a Zero-Trust Architecture (ZTA). As discussed in (Section 3.3), a service mesh (e.g., Istio, Linkerd) can integrate OPA as an external authorizer. The OPA policy, now operating at runtime, can query the Compliance Data Lakehouse to make dynamic, attribute-based access control (ABAC) decisions. For example:
- **Policy:** "Allow `service-A` to access `service-B` only if `service-A`'s production attestation (from the CI pipeline) is less than 30 days old and `service-B`'s runtime attestation shows no unpatched critical CVEs."

This supports treating compliance as a continuous, dynamic property of the running system, not just a pre-deployment check.

- **AI/ML Governance (MLOps):** This is the most critical future-proofing capability. Emerging AI regulations (e.g., NIST AI RMF, EU AI Act) are fundamentally data-driven. The Multi-dimensional Lineage Graph (Section 4.3) provides a practical technical mechanism to satisfy these requirements. By extending the graph to include `model_version`, `training_dataset_hash`, and `bias_test_results`, an organization can prove its compliance with AI regulations. The platform can answer audit queries like, "Show me all production models that were trained on dataset `X`", or "Block the deployment of any model that scored above a 0.8 on the `disparate_impact` bias test." A broad survey of AI governance challenges confirms the necessity of such traceability mechanisms to ensure accountability, transparency, and regulatory alignment across AI-enabled systems [35].

7.4. Limitations and Scope Constraints

While the proposed Continuous Compliance Framework (CCF) demonstrates feasibility and practical applicability, several limitations and scope constraints must be acknowledged.

First, the framework is intentionally designed as a **reference architecture** rather than a turnkey product. Its effectiveness depends on the correct integration of external tools (e.g., CI platforms, policy engines, scanners), which may vary in maturity and performance across organizational environments.

Second, the experimental evaluation focuses on **pipeline-level overhead and evidence generation**, not on long-term organizational adoption costs, human factors, or regulatory interpretation accuracy. The framework assumes that compliance requirements can be expressed as computable policies, which may not fully capture ambiguous or jurisdiction-specific regulatory language without expert interpretation.

Third, the current evaluation emphasizes **build-time and deployment-time compliance**. Although the architecture supports runtime and self-healing extensions, these capabilities are discussed conceptually and were not empirically validated in this study.

Finally, the framework does not eliminate the need for human oversight. Instead, it aims to reduce manual effort in evidence collection and validation while preserving auditability and traceability. The CCF should therefore be viewed as an **augmentation of compliance governance**, not a replacement for regulatory accountability or expert judgment.

8. Conclusions

This paper has proposed a data-centric framework for embedding continuous compliance into DevSecOps pipelines. The central thesis is that verifiable compliance is a data engineering problem, one that is solvable by transforming compliance artifacts from a disparate collection of logs into a queryable, real-time data product. By establishing a Compliance Data Lakehouse (Section 4), the framework creates a versioned, auditable, and

time-indexable record of the compliance state. This foundation enables the three-plane Continuous Compliance Framework (CCF) (Section 3) to function:

1. The Policy Plane defines compliance as executable, version-controlled code (e.g., Rego).
2. The Instrumentation Plane collects verifiable evidence (SBOMs, SARIF, scan results) from the pipeline.
3. The Validation and Attestation Plane uses cryptographic tools (Sigstore) to create non-repudiable proof that policies were enforced.

The framework was validated through an implemented CI/CD pipeline scenario (Section 5) and empirical pipeline measurements (Section 6), demonstrating that continuous compliance checks, evidence persistence, and cryptographic attestations can be integrated with bounded and predictable overhead. These results show that compliance can be shifted from a manual, post hoc, and often adversarial activity to an automated, "shift-left", and collaborative "pipeline-native" control without materially degrading deployment velocity.

By providing a technical solution to the dual pressures of velocity and control, the CCF reduces audit preparation from weeks to minutes and compliance violation Mean Time to Remediation (MTTR) from months to moments. At the same time, this work is intentionally scoped to serve as a reference architecture and for experimental validation. Runtime self-healing mechanisms and long-term organizational adoption impacts that were discussed conceptually (Section 7) remain valid targets for further empirical study.

While this work focuses on establishing verifiable, pipeline-native compliance using today's production-grade tooling, future evolutions in cryptography (e.g., post-quantum signatures) and AI-native control planes (e.g., Model Context Protocol–based governance agents) will introduce new policy, attestation, and orchestration challenges that can be naturally absorbed by the proposed data-centric architecture.

Ultimately, treating governance as a computational and data-centric problem enables the development of systems that are not only compliant-by-design but also verifiably compliant-in-operation. The future of compliance lies not in static documents, but in continuously verifiable data.

Funding: This research received no external funding.

Data Availability Statement: The experimental pipeline, policy definitions, configuration files, and execution logs supporting this study are publicly available at https://github.com/zakhalex/ccf-experiment (accessed on 1 February 2026).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Qureshi, J.N.; Farooq, M.S. ChainAgile: Enhancing agile DevOps using blockchain integration. *PLoS ONE* **2024**, *19*, e0299324.
2. Pourmajidi, W.; Zhang, L.; Steinbacher, J.; Erwin, T.; Miranskyy, A. A Reference Architecture for Governance of Cloud Native Applications. *arXiv* **2025**, arXiv:2302.11617v2. [CrossRef]
3. Lourenço, B.; Adão, P.; Ferreira, J.F.; Marques, M.M.; Vaz, C. Structuring Security: A Survey of Cybersecurity Ontologies, Semantic Log Processing, and LLM Applications. *arXiv* **2025**, arXiv:2510.16610. [CrossRef]
4. Chittala, S. Securing DevOps Pipelines: Automating Security in DevSecOps Frameworks. *J. Recent Trends Comput. Sci. Eng.* **2024**, *12*, 31–44. [CrossRef]
5. Hevner, A.R.; March, S.T.; Park, J.; Ram, S. Design Science in Information Systems Research. *Manag. Inf. Syst. Q.* **2004**, *28*, 75–105. [CrossRef]
6. *ISO/IEC 27001:2022*; Information Security, Cybersecurity and Privacy Protection—Information Security Management Systems. International Organization for Standardization: Geneva, Switzerland, 2022.
7. AICPA. *SOC 2 Trust Services Criteria*; AICPA: Durham, NC, USA, 2017.
8. European Parliament. *General Data Protection Regulation (GDPR)*; European Parliament: Strasbourg, France, 2016. Available online: https://gdpr-info.eu/ (accessed on 15 January 2026).

9. U.S. Department of Health & Human Services. *Health Insurance Portability and Accountability Act (HIPAA)*; U.S. Department of Health & Human Services: Washington, DC, USA, 1996. Available online: https://aspe.hhs.gov/reports/health-insurance-portability-accountability-act-1996 (accessed on 15 January 2026).
10. PCI Security Standards Council. *Payment Card Industry Data Security Standard (PCI-DSS)*; PCI Security Standards Council: Wakefield, MA, USA, 2022. Available online: https://www.pcisecuritystandards.org/ (accessed on 15 January 2026).
11. NIST. *AI Risk Management Framework 1.0*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2023. Available online: https://www.nist.gov/itl/ai-risk-management-framework (accessed on 15 January 2026).
12. *ISO/IEC 42001:2023*; Information Technology—Artificial Intelligence—Management System. International Organization for Standardization: Geneva, Switzerland, 2023.
13. IEEE. *P3395 Draft Standard for AI Risk Management*; IEEE: New York, NY, USA, 2024. Available online: https://standards.ieee.org/ieee/3395/11378/ (accessed on 15 January 2026).
14. Open Policy Agent (OPA). 2023. Available online: https://www.openpolicyagent.org (accessed on 1 February 2026).
15. Kyverno: Kubernetes Native Policy Management. 2023. Available online: https://kyverno.io (accessed on 1 February 2026).
16. SLSA: Supply-Chain Levels for Software Artifacts. 2023. Available online: https://slsa.dev (accessed on 1 February 2026).
17. The Sigstore Project. 2023. Available online: https://sigstore.dev (accessed on 1 February 2026).
18. Torres-Arias, S.; Afzali, H.; Kuppusamy, T.K.; Curtmola, R.; Cappos, J. in-toto: Providing Farm-to-Table Guarantees for Bits and Bytes. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 14–16 August 2019.
19. Linux Foundation. CycloneDX SBOM Standard. 2023. Available online: https://cyclonedx.org (accessed on 1 February 2026).
20. OASIS. *Static Analysis Results Interchange Format (SARIF)*; OASIS Open: Woburn, MA, USA, 2022.
21. dbt Labs Documentation. 2023. Available online: https://getdbt.com (accessed on 1 February 2026).
22. Great Expectations Documentation. 2023. Available online: https://docs.greatexpectations.io (accessed on 1 February 2026).
23. Chandramouli, R.; Kautz, F.; Torres-Arias, S. *NIST SP 800-204D*; Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines. National Institute of Standards and Technology: Gaithersburg, MD, USA, 2024.
24. Kamaluddin, K. Security Policy Enforcement and Behavioral Threat Detection in DevSecOps Pipelines. *Eur. J. Technol.* **2022**, *6*, 10–30. [CrossRef]
25. LF AI & Data Foundation. OpenLineage Specification. 2023. Available online: https://openlineage.io (accessed on 1 February 2026).
26. Databricks. What Is a Data Lakehouse? 2021. Available online: https://databricks.com/glossary/data-lakehouse (accessed on 1 February 2026).
27. Verma, R.; Shrivastava, P.; Merla, N. Tracing the Path: Data Lineage and Its Impact on Data Governance. *Int. J. Glob. Innov. Solut.* **2024**, *1*, 1–10. [CrossRef]
28. Sweet, E. Data Lineage and Compliance. *ISACA J.* **2016**, *5*, 1–4.
29. Li, Z.; Guo, W.; Gao, Y.; Yang, D.; Kang, L. A Large Language Model–Based Approach for Data Lineage Parsing. *Electronics* **2025**, *14*, 1762. [CrossRef]
30. Tyagi, D.; Sharada Devi, P.P. The Importance of Data and Analytics Provenance and Governance in the Realm of Datafication. *Int. J. Comput. Trends Technol.* **2022**, *70*, 28–33. [CrossRef]
31. Adeyinka, A. Automated compliance management in hybrid cloud architectures. *World J. Adv. Eng. Technol. Sci.* **2023**, *10*, 283–297. [CrossRef]
32. Antiya, D.S. Compliance as Code: Automating Compliance in Cloud Systems. *Int. J. Recent Innov. Trends Comput. Commun.* **2020**, *8*, 42–49.
33. Zakharchenko, A. Continuous Compliance Framework (CCF) Experimental Pipeline. GitHub Repository. 2026. Available online: https://github.com/zakhalex/ccf-experiment (accessed on 4 January 2026).
34. Lu, Q.; Zhu, L.; Xu, X.; Whittle, J.; Zowghi, D.; Jacquet, A. Responsible AI Pattern Catalogue: A Collection of Best Practices for AI Governance and Engineering. *arXiv* **2022**, arXiv:2209.04963. [CrossRef]
35. Batool, A.; Zowghi, D.; Bano, M. AI governance: A systematic literature review. *AI Ethics* **2025**, *5*, 3265–3279. [CrossRef]