International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Compliance-as-Code for Continuous PCI DSS 4.0 Validation in Cloud-Native Financial Systems: Principles, Patterns, and Industrial Experience

Murali Ajit Varma

Abstract: The PCI DSS defines requirements to protect cardholder data. Older compliance mechanisms based on evidence collection processes and manual audits do not scale well in the fast, dynamic environment of cloud-native infrastructure. Compliance as Code is a software and engineering approach to compliance. It enforces compliance rules in real-time as a part of the software development life cycle rather than validating compliance post-factum. This article uses policy, implementation patterns, and empirical research on adoption to classify software solutions into architectural categories: policy as code translation, CI pipeline integration, immutable audit trail, and automated remediation. It maps PCI DSS 4.0 requirements to network segmentation, encryption, access control, and audit logging best practices for ephemeral containers and microservices. A production case study from a major B2B2C issuer-processor shows that Compliance-as-Code preventive enforcement (External Secrets Operator population, Config-Init pre-startup validation, Helm chart abstraction, and Kubernetes admission gate) can eliminate compliance exposure windows while bringing developer friction to near zero. Compliance-as-Code preventive enforcement may be considered the counterpart (not the opposite) of detective Governance, Risk and Compliance (GRC) monitoring and auditing activities, two components of the cloud-native regulatory compliance live operational reality.

Keywords: *Compliance-as-Code, PCI DSS Enforcement, Cloud-Native Security, Continuous Compliance Validation, Kubernetes Admission Control, Policy-as-Code*

Introduction

PCI data movement in the cloud depends on the adoption of cloud networking layers and infrastructure to improve scale and reduce costs [1, 2]. Most compliance processes still rely on snapshot verification, evidence collection and ad hoc audits [3] and thus lag cloud-native deployment and infrastructure as code with continuous monitoring [4]. The distributed service model raises new ownership, control and trackability issues, and introduces compliance and other regulatory difficulties [5]. Security risks include loss of data, insecure application programmer interfaces, misconfiguration and unauthorized use of an account and its service traffic. This can result in financial or reputational damage to individuals and organizations. However, the shared cloud service model has created confusion over the delineation of responsibilities for certain security controls, leading to incidents of non-compliance and data loss [1].

Independent Researcher, USA

More recently, several papers [2, 6] have demonstrated the real-world applicability of Compliance-as-Code in automating ISO 27001 and PCI DSS compliance in cloud environments. Compliance-as-Code presents an important opportunity for the compliance industry due to the manual nature of existing compliance processes that are not suitable for continuous monitoring and real-time validation in rapidly changing cloud environments. Through AWS Config and OPA, Compliance-as-Code enables automation of 85 of 114 controls in the ISO 27001 standard and 10 of 12 controls in the PCI DSS standard leading to meaningful reductions in audit time and errors. For example, Audit time was reduced to as low as 70% of the original time it would take to do an initial compliance audit and continuous monitoring audit, 24 to 8 hours and 10 to 3 hours respectively [6]. When using Compliance-as-Code, human error decreases by 83.3% for ISO 27001 and 90% for PCI DSS [6]. The development of Compliance-as-Code in the model of cloud governance lowers the risk of non-compliance, increases scalability, and

enables real-time visibility into the compliance status of the cloud.

Problem Statement and Objectives: While automated compliance solutions exist, point in time compliance is not fit for purpose for cloud-native financial systems that process cardholder data [6]. Compliance drift is a consequence of the audit time-boxed nature of compliance systems and the fast, continuous deployment of DevOps implementations. The average cost of a data breach in the financial services sector reached USD 6.08 million in 2024, well above the global average [7], underscoring the consequences of compliance failures. This exposure of organizations leads to compliance risks and therefore the consequences of being in breach of such requirements. The aim of this article is to provide: (1) an overview of the structural limitations of customary compliance frameworks with a focus on cloud-native applications, (2) architectural design principles and implementation patterns of the Compliance-as-Code compliance framework of choice for PCI DSS 4.0 compliance, (3) an overview of the distinctive properties of Compliance-as-Code in contrast to customary GRC frameworks, namely its superior timing and enforcement capabilities, (4) design patterns for the most common PCI DSS 4.0 controls, such as network segmentation, and (5) use a case study showing the complete elimination of compliance exposure windows at scale in a production environment for a B2B2C financial infrastructure.

Background and Regulatory Context

PCI DSS 4.0 Requirements

Several PCI DSS 4.0 requirements are directly applicable to the automated enforcement and monitoring of security controls in cloud-native environments [31]. For example, requirement 3.3.2 says that technical controls must be in place to prevent copying or moving PAN (Primary Account Number) via remote-access technologies. Requirement 3.3.3 requires that if the PAN is stored, it must be protected by strong cryptography (per Requirements 3.5.1.1/3.6.1 key management requirements). Key hashes must be part of a key-management process. Key-management controls must address the generation, distribution, storage, retirement, and replacement of keys. Requirement 8.6.3 requires that application and system accounts be managed according to a defined frequency and complexity [31].

Taken together, these requirements make one thing clear: compliance controls need to be automated and run continuously. Furthermore, the limits of manual evaluation, the requirement that the standard envisions for controls to be performed and continuously collected at all times, and the new custom approach validation, which allows organisations to use alternate approaches to demonstrate compliance when the underlying control objectives can be achieved, and make an automatic evidence approach more compelling [31].

CISA Secure-by-Design and NIST SP 800-218

To help ensure security is integrated throughout the software development life cycle, CISA's 2023 Secure-by-Design policy recommends adopting practices such as [32] and NIST SP 800-218, the Secure Software Development Framework (SSDF), includes the practice groups Protect Software (PS) and Produce Well-Secured Software (PW) [33]. This recommendation is echoed in the 2024 U.S. Treasury report on cybersecurity risks in the financial services sector concerning AI [34]. Config-Init containers leverage the practice group of PS by validating configuration prior to the launch of the workload, as explained in Section 5. Helm chart abstraction implements the PW practice group by baking secure deployment patterns into reusable, versioned packages.

The B2B2C Infrastructure Context

This is the only compliance multiplier in the B2B2C issuer-processor case. Building compliance tooling at the platform layer rather than at the product or customer layer multiplies the impact of compliance across the entire customer base of the platform and thus across all downstream end consumer customers rather than one. Issuer-processors such as FinPlatform also provide the APIs, ledgers, card processing, compliance tooling, and other backend infrastructure for consumer-facing fintechs including large neobanks, investment apps, and digital banks [35]. The compliance multiplier is especially high at this layer of infrastructure, where individual implementations may onboard hundreds of millions of consumer accounts via dozens of downstream customers.

Methodology

This paper proposes a literature review and a case study of Compliance-as-Code technologies for

meeting PCI DSS 4.0 standards in the creation of cloud-native financial systems. This approach is to identify and analyze the available literature and methodologies for identifying studies related to a particular research topic. To achieve the goal above, this paper examined academic publications from 2020 through 2025 in IEEE Xplore, ACM Digital Library, ScienceDirect, SpringerLink and the major peer-review journals on cybersecurity, cloud computing, electronic and digital banking, and regulatory compliance. Industry white papers of leading technology companies, regulatory agencies, and professional organizations were also analyzed for best practices in implementation.

The search terms were "compliance-as-code," "PCI DSS automation," "cloud-native security," "policy-as-code," "DevSecOps," "continuous compliance," and "regulatory enforcement." Preventive controls are different from detective controls, which detect compliance violations after a policy violation when a change has already been deployed, rather than preventing a non-compliant change from being deployed in the first place. In common with cloud-native architectures generally, preventive controls rely heavily on containerisation and infrastructure-as-code, as well as automated enforcement of security checks as part of the continuous integration and continuous deployment pipeline, rather than audits at less frequent points.

The Structural Limitations of Traditional Compliance Frameworks

The Point-in-Time Certification Problem

The PCI compliance model is prescriptive on some infrequent and static snapshot of system security posture (once per year or less) that cannot be reasonably mapped to the continuous and fluid in-scope infrastructure of cloud-native development models. As PCI DSS assessment methodologies, both the SAQ and RoC involve the establishment of configuration states, access controls, or security implementations that immediately become outdated historical records rather than accurate representations of the operating state of a cloud-native system. Tool-siloed compliance processes based on snapshot-in-time audits are poorly-suited for containerized, continuous delivery environments, leading to compliance drift and delayed remediation [11].

Research has shown that point-in-time vendor risk management models, which rely on one-time, annual surveys are not well suited for security risk, which can change rapidly due to zero-day vulnerabilities, attrition of security professionals, or misconfigurations [10]. Self-reported information can be outdated, incomplete, or inaccurate, leading to false positives and the overestimation of security guarantees [10]. The classic PCI DSS compliance model assumes it is possible to take a picture of the environment once a year, but in a cloud-native environment that picture is not static. The container is replaced in seconds, and the network policies change. The reality of configuration drift is not if it will occur, but instead when it will occur.

DORA research demonstrates that organizations lacking automated platform capabilities experience significantly higher rates of unplanned rework and operational toil [8]. Organizations that fail to maintain continuous compliance checking face regulatory penalties under PCI DSS enforcement provisions [2].

A more recent proposal for Continuous Compliance Framework (CCF) [11] which is a data-centric reference architecture, makes compliance checks part of each CI/CD build pipeline. In doing so it creates a change from a periodic point-in-time compliance goal to a continuous validation of compliance in the software development lifecycle [11]. In this model, compliance is a first-class, computable property of the system, achieved through a combination of declarative policy as code, evidence collection mechanisms, and cryptographic attestation [11]. The resulting collection of unverified compliance state is described in one paper as "compliance debt" - a governance liability that compounds silently until the next audit occurs [11].

Manual Evidence Collection as a Bottleneck

These constraints have the downside of requiring extra engineering effort, or delaying the development of productive systems by engineers and architects. DORA research consistently shows that teams without automated compliance and platform capabilities spend a disproportionate share of their time on rework and pipeline remediation rather than productive feature development [8]. The Uptime Institute's annual outage analysis confirms that the majority of significant IT outages are attributable to human-error misconfiguration and procedural failures [9]. As a result, engineers spend time pulling configuration screenshots, correlating access logs across systems, and mapping that evidence to the PCI DSS control domains when it would have been better spent building and shipping

a product. These evidence bundles are not programmatically verifiable and become invalid with any change to the infrastructure.

In a classic on-premise system, compliance evidence is always incomplete because the schemas, stored procedures, triggers, and application dependencies are not fully captured [12]. Compliance has always been a retrospective manual audit activity that occurs after the fact, sometimes weeks to months into the operational mode of the system. The compliance burden has led to the term "compliance debt," and regulatory risk has become one of the few systemic governance risks for regulated industries, such as finance, healthcare and government [11]. A solution is the Continuous Compliance Framework (CCF) [11], which defines compliance as a computable property and verifies it on every CI/CD build. It is based on declarative policy-as-code and cryptographic attestations stored in a compliance data lakehouse.

Fragmented solutions make it impossible for organizations to have a global compliance posture and lead to duplicative verification steps that require resources from developers to create evidence. DORA research indicates that burnout risk increases by approximately 40% in engineering teams that lack automated platform capabilities for compliance and deployment [8].

Reactive Remediation and Compliance Drift

Reactive remediation processes are also triggered when control deficiencies (typically identified due to a missing control or a misconfiguration) are found during the audit. Remedial documentation is created, priorities are established, and remediation actions are taken; control is then validated. Remediation may take weeks or months, leaving the organization exposed to penalties and real security risks during that time [10]. Without security compliance monitoring, organizations may have longer mean time to detect (MTTD) and mean time to remediate (MTTR), and may also have outstanding compliance risks and issues [8].

Without re-assessment, compliant systems drift into non-compliance as business processes change, configuration is altered, or new services are deployed without security assessments until the next audit cycle [10]. The customary security model, which defers a formal architecture review and manual vulnerability assessment until the end of the development cycle, does not suit agile software development that implements changes to the code frequently [14]. Newer data privacy regulations compound the problem, with entities subject to one or more often-conflicting requirements such as the European Union's General Data Protection Regulation (GDPR), HIPAA, and PCI DSS [13]. GRC automation can reduce the time it takes to check for compliance by as much as 70% [21], but reducing time to detection is not the same as preventing a problem from happening. That distinction underlies the concept of Compliance-as-Code described in this review.

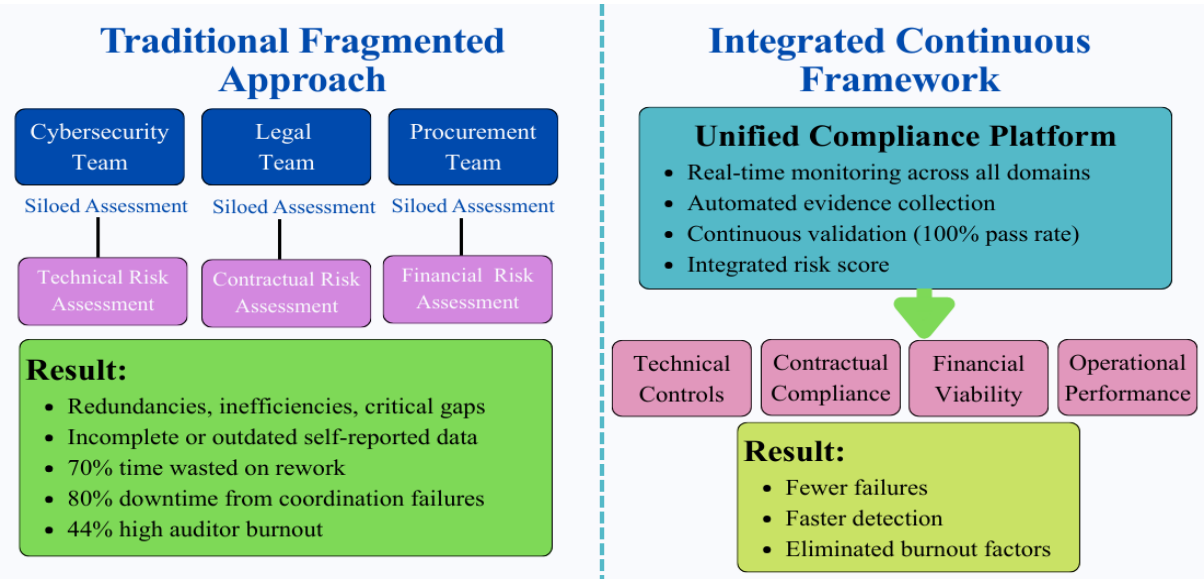

Figure 1: Compliance Framework Evolution: From Fragmentation to Integrated Framework [Author's synthesis based on 8, 11]

Compliance-as-Code: Architectural Principles and Implementation

Policy-as-Code: Codifying Requirements into Executable Rules

Compliance-as-Code is the definition of compliance requirements and policies in the form of code and their automated implementation in the software development and delivery pipeline [26, 15]. As an example, the PCI DSS requires encryption of transmissions of cardholder data across open, public networks, and limiting access to cardholder data on a need-to-know basis [2]. Implementing it in programmatic policies and enforcing them against infrastructure state, code, and controls requires wide-ranging domain knowledge in both security compliance and infrastructure automation. Compliance engineers are experts in regulatory and compliance requirements who can code, practice DevOps, and operate as a bridge between the regulatory and software development and operations domains [15]. Policy engines (e.g., Open Policy Agent, HashiCorp Sentinel, and Kyverno) can express compliance constraints in a code-like language, which is checked into version control, tested, and automatically enforced in CI/CD tooling [20]. OPA (a CNCF graduated project) is a general-purpose policy engine that uses the Rego language [12]. Sentinel has integrations with other HashiCorp products like Terraform, Vault, and Consul to create policies with the HashiCorp stack. Kyverno uses a Kubernetes-native approach to policy configuration by defining policy as a set of YAML resources, reducing the developer learning curve [20]. Although each of the three is capable, none ship with PCI DSS-specific compliance logic. Each provides the foundation, but the organisation still has to construct the full compliance structure above it: the compliance policies, the credential lifecycle management, the pre-startup validation logic, and the adoption packaging that makes the organizations comfortable to use it without becoming PCI DSS experts.

Organizations adopting policy-as-code reported improved audit readiness and reduced compliance findings [39]. Useful principles of Compliance-as-Code may be defined as transparency, repeatability, and auditability. These principles are a basis for other areas of compliance, including government, energy, and defense, with increasing complexity around data protection and cybersecurity. In addition to machine learning and AI capabilities, new systems can automate the detection of compliance drift and policy implementation anomalies to create auto-healing governance ecosystems that are more proactive than customary IT governance processes, which are largely derived from audit findings [24].

Aspect	Details	Key Point
Definition	Compliance coded in pipelines	Automated enforcement
OPA	Uses Rego language	CNCF graduated project
Sentinel	HashiCorp stack integration	Terraform, Vault, Consul
Kyverno	Kubernetes-native YAML	Lower learning curve
Common Gap	No PCI DSS logic	Org must build policies
Key Role	Compliance Engineers	Bridge reg & DevOps
Core Principles	Transparency, Repeatability	Auditability
Reported Benefit	40%+ less non-compliant	Better audit readiness
AI/ML Capability	Detects compliance drift	Auto-healing governance
Broader Use	Gov, energy, defense	Data & cyber compliance

Table 1: Policy-as-Code - Key Concepts, Details & Insights [12, 19, 20, 39]

Pipeline Integration: Embedding Governance into CI/CD

Compliance-as-Code introduces policies in CI/CD pipelines, instead of auditing on each deployment. Applications and infrastructure are continuously scanned in build, test and deployment stages [4]. While it is possible to verify all the above before moving code to source control or provisioning the infrastructure or updating configuration files, this Persistent Compliance is treated as a Continuous Compliance. The Continuous Compliance Framework (CCF) implements a layered federated controls model to allow centrally managing compliance policies at various decision points (e.g.

CI/CD pipelines, application runtime environments, etc.) [11]. The Policy Definition and Management Plane represents the central repository of all compliance policies. Then there is the Instrumentation and Data Collection Plane that provides assurance of compliance in an active and automated way. The Validation and Attestation Plane implements the policy and maintains the compliance status in an auditable trail. Each stage (build, test, stage, and production) in a DevOps pipeline is gated by rules that point to a policy repository. A gated pipeline will not move past a step until all violations in an actionable stage have been resolved or overridden with a justified exception stored in the Policy Repository [17]. This prevents unintentional policy divergence from propagating through the pipeline. It offers out-of-the-box triggers and integrations for popular CI/CD tools, including Jenkins, GitLab CI, and GitHub Actions, while keeping pipelines lightweight [17].

Other checks are woven into the pipeline. Static code analysis checks the application code for security anti-patterns, such as hard-coded credentials, or not properly validating inputs [37]. Infrastructure-as-code validation ensures that Terraform, CloudFormation and Kubernetes manifests comply with defined networking, encryption, access control rules. On container images, scanning ensures that deployed artifacts do not include known vulnerabilities or prohibited software libraries. Runtime policy enforcement is evaluated by capturing how many requests to deploy are checked for segregation of duties and change management workflows. Shift-left policy enforcement catches the majority of violations at the earliest pipeline stages, with modest overhead in deployment velocity that does not significantly affect high-performing teams [8]. The performance overhead of automated policy evaluation is generally acceptable compared to the impact of the achieved higher compliance level from automated validation [17]. Shift-left security can be used to check for compliance in various phases of the SDLC. Shift-left security proposes conducting security compliance checks as early in the development process as possible. For example, threat modeling is performed during the planning phase. Static application security testing (SAST), secure code review, and secure coding standards are used when the code is being written. Dependency analysis and hardening policy review are used at build time. DAST and continual security findings management occurs during the testing stage, integrity checking of distribution files occurs during the release stage, and provenance attestation and runtime policy checking during deployment. Through high-pressure full integration, human errors can be eliminated, compliance automatically checked, and traceable decision-making enabled [16].

Organizations in heavily regulated industries, such as GDPR, HIPAA, PCI DSS, FedRAMP, and ISO/IEC 27001 in its more thorough controls, experience additional challenges to adopting cloud native. The distributed nature of microservices and the short lifespan of containers can complicate data residency and also make it more difficult to conduct post-mortem analysis of change and outages. Automated governance principles of the Security-as-Code model (such as policy-as-code, continuous compliance, and continuous evidence generation for security controls all automated as part of a DevOps process) can help to address the key security principle adoption challenges in the cloud software pipeline. Organizational culture and the changing role of the cloud provider in the shared security responsibility also play an important role in the success of Security-as-Code [18].

Immutable Audit Trails through Pipeline Artifacts

All runs of the compliance-improved data pipeline record which policies and resources were evaluated and whether compliance was achieved, creating an auditable proof of continuous compliance rather than a single moment of conformance. Being immutable and append-only, the logs provide a deterministic audit trail, which reduces compliance risk, since controls are enforced and validated by default. Furthermore, the artifacts are cryptographically signed and timestamped and bound to concrete changes in the infrastructure, in contrast to the ad hoc evidence packages used by conventional compliance, where auditors ask the systems for the specific set of controls that have been applied during arbitrary time windows [11]. Kellogg et al. [26] gave first-class attention to continuous compliance as an important software architectural property. However, a system architecture that integrates credential lifecycle management and pre-startup validation with

frictionless adoption through infrastructure abstraction and admission control for the PCI DSS 4.0 compliance domain has not yet been described in the literature with production-grade artifacts. The next section of this review addresses this gap.

Manual processes that produce audit evidence, such as checklists or spot audits every few months, do not scale. Audit evidence for cloud-native applications must be produced automatically, e.g. by a machine or tool. While increased audit readiness has been theorized as being a potential benefit of a proper Compliance-as-Code process, empirical studies into cloud providers and regulatory technology implementations have demonstrated meaningful reductions in non-compliance violations through continuous policy enforcement [6, 11]. Continuous compliance makes an organization aware of violations in advance through real-time policy enforcement rather than retroactively through audit violations. With such architectural transformations, retrospective evidence collection becomes unnecessary, so it is possible to determine the current compliance posture of a distributed system. In a controlled environment where end-to-end traceability is required, immutable audit trails of configuration change, access control change, and policy evaluation results can be produced as the software delivery lifecycle progresses [26]. The open nature of Compliance-as-Code allows development teams and auditors and other stakeholders/regulators to share the data of the compliance validation and therefore save on efforts in relation to the gathering of evidence, which had been a major aspect consuming the resources of an organization [33, 32].

Security compliance continuously (e.g., security activities in CI/CD pipelines) is methodologically and technically feasible. Practitioners made roadmaps for prioritizing security practices while applying RefA and piloting RefA-MR. The next groups of practices for security compliance of the MVP in the DevOps pipeline are: (A) Plan: Applicability analysis of security standards and threat modeling for specifying product security requirements and secure design; (B) Code: Static code analysis, incorporation of secure coding standards; (C) Build: CI code static analysis, third-party software vulnerability analysis using software bill of materials (SBOM), hardening policy enforcement implementation inspection, and distribution file integrity verification. (D) Test: also includes dynamic application security testing (DAST), run-time testing, smoke tests, and continuous lifecycle management of security findings. (E) Release: verifying distribution file integrity and preparing the distribution's security policy. (F) Deploy: provenance assurance [15].

Automated Remediation and Self-Healing Infrastructure

More mature Compliance-as-Code implementations may include self-healing governance ecosystems which detect policy violations and compliance drift or anomalies in runtime applied policies [24]. If runtime checking or re-checking detects a policy violation in production, the framework applies policy remediation automatically to keep runtime configurations in compliance. It lowers mean time to remediation, as compliance violations are treated as transient exceptions rather than permanent violations [11]. Programmatic enforcement means that if inbound/outbound traffic is non-compliant, it is reverted to compliant security group rules. Unauthorized access is denied and logged, and systems are brought back to a compliance baseline. Artificial intelligence and machine learning can provide continuous monitoring for compliance drift, the prediction of when the drift will exceed the threshold, and continuous remediation to return the system to a compliance state before this prediction is realized. Organizations that take an automatic remediation approach have seen substantial reductions in standing violation counts compared to manual compliance management [25, 26].

This self-remediation capability overcomes the weaknesses of reactive remediation systems, where security teams would typically have had to manually collate, prioritize, and substantiate their remediation actions and wait weeks or months until the next compliance audit cycle was due to see if their controls were functioning as intended. Automated self-remediation repairs compliance violations directly in the deployment cycle, instead of after the next audit cycle [25]. An infrastructure management system can provide such automation by providing policy expressions as one of its core services. Open Policy Agent (OPA) by Styra (now a graduate project of the Cloud Native Computing Foundation (CNCF)) is a domain-agnostic policy-as-code language allowing policy writers to specify

policies in the form of arbitrary input requests given in JSON [19]. OPA is domain-agnostic; it does not ship with predefined binding policies. This allows organizations to create policies tailored to their regulations or tech stack. HashiCorp Sentinel is a proprietary domain-specific language (DSL) and policy-as-code framework that allows policies to be used across HashiCorp products such as Consul, Nomad, Terraform, and Vault [20]. Sentinel is a context-free language based on JSON. It is designed to be easy to write and was embedded into HashiCorp products to provide policy as code in infrastructure provisioning workflows. Kyverno is a Kubernetes-native policy engine that stores its policies as native Kubernetes resources in YAML so that developers do not need to learn to use a new policy language. Kyverno can validate, mutate, generate and clean up any Kubernetes API resource, and has complex conditionals using JMESPath and Common Expression Language (CEL). The Kyverno JSON sub-project provides a JSON implementation of the Kubernetes-native policy engine that uses the Kyverno framework to enforce policy on a set of JSON documents outside of Kubernetes clusters.

Policy Engine	Key Capabilities	Supported Frameworks	Integration Level
Open Policy Agent (OPA)	Machine-readable policies	GDPR, HIPAA, PCI DSS	CI/CD pipelines
HashiCorp Sentinel	Declarative constraints	FedRAMP, ISO/IEC 27001	Infrastructure-as-code
Kyverno	Kubernetes-native policies	Multi-framework	Container orchestration
General CaC Implementation	Automated validation	Financial, healthcare, government	Full SDLC integration

Table 2: Policy Enforcement Tools and Regulatory Framework Coverage [12, 19, 20, 39]

Architectural Differentiation from GRC Platform Solutions

Manual Workflow versus Automated Workflows

The main architectural difference between the customary Governance, Risk, and Compliance (GRC) model and the model of Compliance as Code is when and what type of controls are enforced. The customary GRC platform uses detective controls to collect evidence of security implementation, compliance program metrics and artifacts used in the manual process (spreadsheets, Word documents, etc.) from production. Real-time monitoring and reporting are helping GRC automation reduce compliance checking time by as much as 70%; however, GRC tools are still largely retrospective, checking compliance violations after the fact rather than preventing them in the first place. Products such as Drata, Vanta, and Secureframe may have contributed to making compliance operations more present-centric, providing dashboards and evidence collection while integrating with cloud providers to monitor changes to cloud configurations. CSPM tools such as Wiz scan cloud configurations for vulnerabilities. Vulnerability scanners such as Snyk perform static analysis at the code level, greatly improving on spreadsheets and screenshots as a method. All of these tools suffer from a fundamental architectural limitation, where everyone is a detective, not a preventive. These tools observe infrastructure state after a deployment has already occurred and signal when something does not look right, but cannot prevent the non-compliant deployment from going out [21]. The time it takes from when a non-compliant change reaches production until its detection, sometimes referred to as the compliance exposure window, is measured in minutes to days depending on how monitoring is configured and how alerts are routed [26].

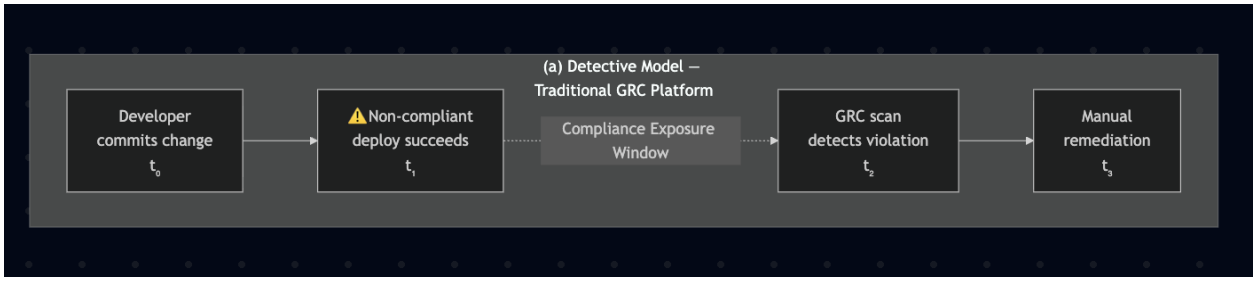

Figure 2: Detective Compliance Model - Exposure Window

In Figure 3, the Preventive compliance model is depicted. In this, the enforcement gate checks for deployments at t_1 via Config-Init validation, admission controllers, and Helm chart enforcement, blocking the non-compliant configurations from being deployed into production.

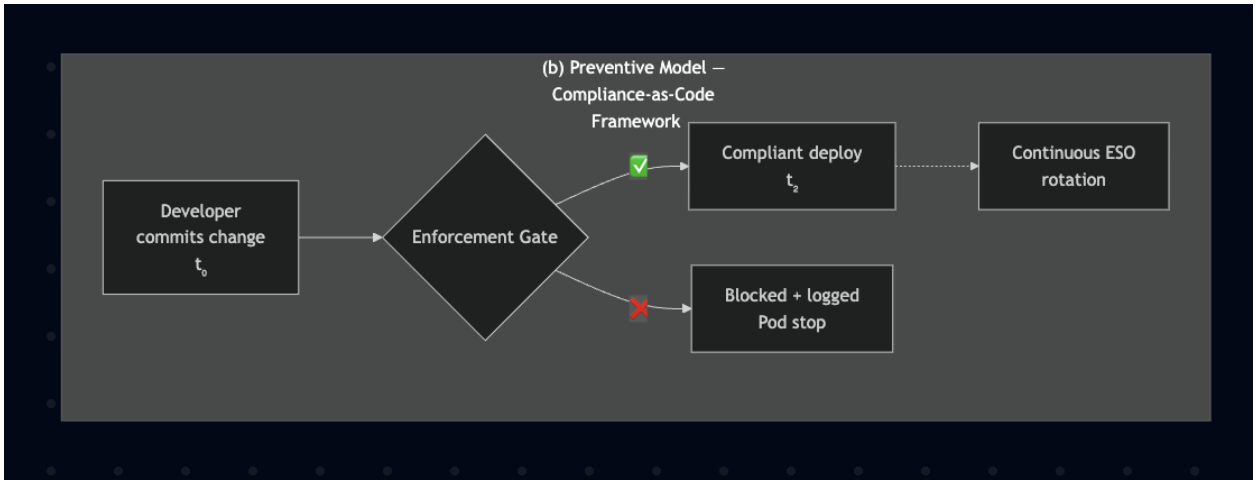

Figure 3: Preventive Compliance Model - Enforcement Gate

For example, regulatory compliance requirements (EU GDPR, HIPAA, PCI DSS, etc.) and increasing cyber threats make it inefficient, ineffective, and error-prone to use manual processes for IT GRC management, especially in the dynamic cloud environment [23]. The non-compliance rates are 82%, 76%, 79%, 71%, and 68% for security standard compliance, data privacy standard compliance, third-party risk management standard compliance, industry standards compliance, and multi-cloud compliance monitoring, respectively. These gaps may be caused by time spent by surveillance to discover violations and lack of remediation or the time between a change in configuration and a remediation to the violation [21]. Compliance-as-Code is preventative in that the change will not take effect if it is non-compliant, a shift from compliance observation to compliance enforcement [21]. AI is applied to the 5 Compliance-as-Code domains: DevOps and Security Automation, including pipeline and IaC security; Machine Learning and Anomaly Detection, including deep learning-based log analysis; Security Testing and Vulnerability Assessment, including automated model-based testing; Threat Detection and Risk Assessment, including threat intelligence with AI; and Cloud and Multi-Cloud Security, including distributed architecture [24]. Owing to the wide-ranging use of AI, proactive enforcement of compliance is feasible, as opposed to customary systems, which enable reactive governance, risk, and compliance (GRC) architecture.

In GRC, IT GRC was defined as follows: governance, risk, and compliance for information technology in a specific organization that integrates and automates activities and increases visibility of the effectiveness of technical controls and operational weaknesses impacting business objectives. Despite this important finding, little research has been done to analyze the integration of cross-domain, multi-jurisdiction, and multi-disciplinary regulation. Organizations have continued to implement GRC in silos [26].

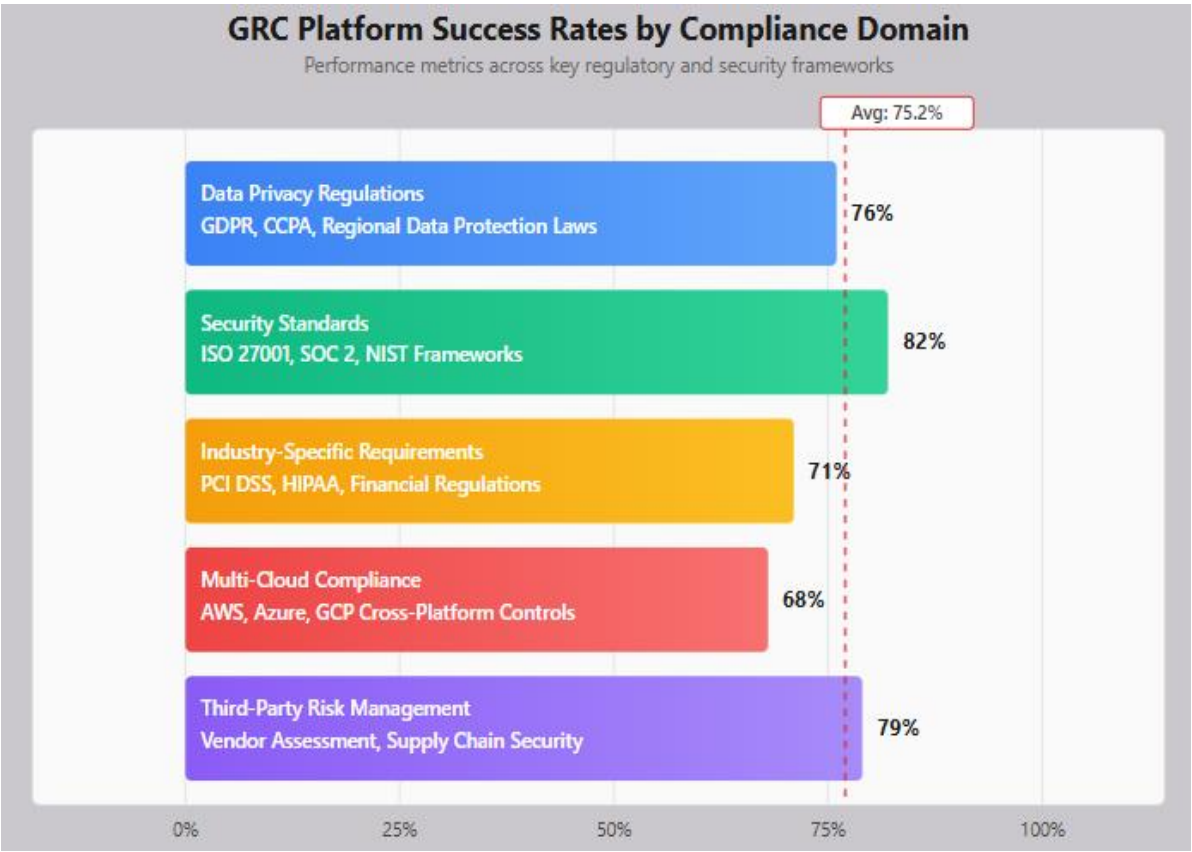

Figure 4: GRC Platform Success Rates by Compliance Domain [23]

Integrated Workflows versus Audit-Centric Design

Standard GRC products consist of evidence dashboards related to a specific set of controls and report templates for auditors and compliance practitioners. Developers are generally regarded as secondary users of GRC systems, with their time being spent responding to compliance requests. In addition, developer access to established GRC systems is generally limited to remediating compliance findings or on-demand evidence requests, thereby requiring context switching between development tools and standalone GRC systems that are not integrated with the software delivery process. Its audit-based architecture also enables traceability and hierarchical control without the need for integrated services [22].

Without a whole-of-organization approach to compliance digitalization, each silo has built its own solutions. These are still designed on the basis of internal benchmarking, instead of GRC process modeling. Today's GRC automation solutions often center around cloud-based automated policy management, which provides no meaningful developer experience and no feedback from the compliance team. Compliance-as-Code provides this capability at the point of consumption of the policy within the developer tooling, causing policy violations to appear as build failures in the developer's build environment, providing remediation guidance to developers in developer language and allowing policy to be validated in real time using existing CI/CD tooling [21]. Although integrated AI and machine learning software platforms are helping to reconcile independent audits from silos, verification remains a separate practice outside of software delivery in the basic GRC architecture. Organizations implementing automated risk assessment and third-party risk management systems still experience challenges with continuous workflows because of complexity of regulations, and the architectural separation between the measurement of compliance, and the operation of compliance [21].

Periodic Monitoring versus Continuous Control Monitoring

Customary GRC tools may only take snapshots of configurations daily, weekly, or monthly. Because it may take some time to take action after a compliance check, this creates a time window

where non-compliance may be undetected until the next snapshot is taken. Customary GRC tools are not suited for managing the fast-paced, dynamic risk environment found in modern digital infrastructure. On the other hand, regulation is compelling automation, with requirements for real-time compliance reporting imposed on jurisdictions, and supply chains facing the need to navigate increasingly complex and divergent sets of regulations across multiple jurisdictions. Static assessments are not suited to this pace of change, as an organization's security posture is subject to change between point-in-time assessments due to zero days, staff turnover, and configuration drift. Even with compliance checking tools based on AI, automated risk assessment and integrated monitoring, the periodic snapshotting model does not change. Full compliance is still impossible because the architecture is constantly reconfigured [21].

In order for Compliance-as-Code to be effective, compliance checking must take place continuously, rather than at a scheduled time. This means that the amount of time that the violation is present in the system is reduced from days or weeks down to the time it takes to make a request to violate a compliance rule. The NIST Risk Management Framework is a process framework that provides organizations with a thorough approach to integrating information security and risk management activities into the system development life cycle. This process can be tailored to the specific needs of the underlying cloud service models (IaaS, PaaS and SaaS) and regulatory environments. Continuous risk assessment and monitoring recommended in NIST, as well as stakeholder engagement, can also be automated with GRC tools [23], and the differences between blockchains and periodic logging can also be considered. Hybrid models store small immutable objects such as hashes, timestamps, and transaction IDs on-chain in performant storage, with the context stored off-chain. While immutable anchoring allows continuous log validation at no performance cost, the best model depends on the assurance required, the number of compliance events, and the resources available for the process. When firmly attached, such controls provide integrity assurances through non-manipulatable, correctly applied persistent logs [22], which in turn removes the days or weeks of window where the out-of-compliance change can be present. Non-compliant configurations are rejected before reaching a running system state.

Listing 1. Kyverno Admission Policy (Sanitized)

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
```

Implementation Patterns for PCI DSS 4.0 Requirements

Network Segmentation and Cardholder Data Environment Isolation

Network segmentation is another foundational requirement for PCI DSS compliance. It seeks to limit the number of systems involved in processing cardholder data (CHD) by isolating these from untrusted networks using manually audited firewall settings. In a cloud-native environment, a container processing payment data could be replaced with another, with a different network address, in seconds, making network segmentation ineffective [27]. Infrastructure-as-code policies, in the form of Compliance-as-Code, could fail the build if Terraform or Kubernetes deployment manifests indicate cardholder data environments would be connected to untrusted networks. Container-based services can use overlay networks and service meshes to enforce network policies at segmentation and service orchestration levels to block lateral traffic between services [27]. Admission controllers block the creation of resources on the server side at the Kubernetes API server level when the resources do not satisfy specific requirements. This holds true even if someone bypasses the Helm chart or misconfigures the manifest, as admission policy checks are enforced for the usage of approved image registries, labeling resources for audit tracking, the minimum security context (non-root and read-only root filesystem), and the existence of network policies for CDE segmentation. Here is an example Kyverno admission policy for these requirements:

```
  name: pci-dss-enforcement
  annotations:
    policies.kyverno.io/category: PCI DSS 4.0
spec:
  validationFailureAction: Enforce
  rules:
    - name: require-approved-registry
      match:
        any:
          - resources:
              kinds: [Pod]
      validate:
        message: "Images must come from an approved registry."
        pattern:
          spec:
            containers:
              - image: "registry.internal/*"
    - name: require-non-root
      validate:
        message: "Containers must run as non-root (Req 7.2)."
        pattern:
          spec:
            securityContext:
              runAsNonRoot: true
```

The container architecture needs to update firewall rules, access control lists, and intrusion detection. Each microservice runs in an individual security group and complies with PCI DSS requirements (e.g., encrypted communication, access controls, opening only the required ports) [27]. Beyond code-based network policies validated against topology models, which define the trust boundaries of network segments, DevSecOps also provides early security and compliance through the CI/CD pipeline. This is accomplished via automated static application security testing, dynamic application security testing, and continuous compliance [28]. The same frameworks are employed to automate penetration testing of deployed applications and enforce configuration compliance with security guidelines like the CIS Benchmarks [28].

Encryption of Data in Transit and at Rest

The PCI DSS encryption requirement states that if a hacker has bypassed all perimeter controls and has access to encrypted data, the data must be rendered unreadable and unusable without strong cryptographic protection [29]. This requirement is similar to the PCI DSS requirement to mask the PAN and encrypt all sensitive data. Encryption of cardholder data in transit and at rest can be accomplished by encrypting it at different layers of a containerized architecture, including ephemeral storage volumes, ephemeral container configuration management, and ephemeral containers, and on distributed data caches. In general, strong cryptographic methods must be used. PCI DSS also states that cardholder information should only be stored if required and that data should be contained. The data must also be protected by one-way hash functions, strong cryptography, truncation, index tokens and securely encrypted cryptography padding, and should use best-practices for management of cryptographic keys as described in the Security and Privacy Controls for Information Systems [38]. Static analysis tools can review application code for database connectors, APIs, and file read-write methods to check whether the appropriate encryption configuration is in use. In addition, infrastructure-as-code templates can be scanned to verify whether the selected storage is encryption-capable or whether selected load balancers are configured with TLS protocols [27].

Config-Init binaries, executed as Init containers, perform validations of required encryption configuration before the application container runs in a Kubernetes pod and check if required secrets exist and are valid. This encrypted configuration must meet PCI DSS minimums, or the pod will not start. The application is never running if it is non-compliant. This logic is illustrated in the following pseudocode:

Listing 2. Config-Init Compliance Validation Logic (Pseudocode)

```
#!/bin/sh
# PCI DSS 4.0 Pre-Startup Compliance Validation
CHECKS_PASSED=true

# Req 3.6.1: Verify keys exist in secret store
if ! verify_secret_exists "encryption-key"; then
  log_compliance_failure "PCIDSS-3.6.1" "Missing encryption key"
  CHECKS_PASSED=false
fi

# Req 3.3.3: Encryption algorithm meets minimum standard
if ! validate_encryption_algorithm "AES-256-GCM"; then
  log_compliance_failure "PCIDSS-3.3.3" "Weak encryption config"
  CHECKS_PASSED=false
fi

# Req 10.2: Audit logging configured and reachable
if ! check_audit_endpoint_reachable; then
  log_compliance_failure "PCIDSS-10.2" "Audit logging unreachable"
  CHECKS_PASSED=false
fi

# Req 7.2: RBAC least-privilege verified
if ! validate_rbac_least_privilege; then
  log_compliance_failure "PCIDSS-7.2" "RBAC policy violation"
  CHECKS_PASSED=false
fi

if [ "$CHECKS_PASSED" = false ]; then
  emit_compliance_event "BLOCK" "Pre-startup checks failed"
  exit 1  # Pod will not start
fi
```

For the static analyzer at rest and rest encryption, the components can either be container orchestration tools or be integrated into the provided cloud service. The transfer encryption and rest-in-transit encryption should be done using TLS configurations between all the microservices transmitting payment information [27]. Container image scanning also ensures that approved, up-to-date cryptographic libraries and ciphers are used and that encryption is enforced at all levels, including at an infrastructure or run-time level as well as within the code the container runs [27]. In some examples, there are data encryption policies stating that data at rest and in transit in continuous integration pipelines and in continuous deployment pipelines should be automatically encrypted throughout the process to avoid data leaks [28]. No Example currently provides a definitive approach

for implementing the database encryption requirements. Types of databases, connections, size, access code and encryption of the software system should be taken into consideration [28]. For any database system, there is a high level of security due to encrypted private data in the software system input and output. However, systemic data leaks can occur via log files, debug statements, and database artifacts [29]. Code Security-as-code in pipelines can help to reduce these threats continuously. It identifies security issues such as vulnerabilities or misconformities at an earlier stage of deployment [28].

Encryption Method	Reversibility	Security Level	Implementation Complexity	Use Case
One-Way Hash Functions	Irreversible	High	Low	Data that doesn't need retrieval
Truncation	Irreversible	Medium	Low	Partial data masking
Strong Cryptography	Reversible	High	High	Data requiring retrieval
Index Tokens	Reversible	High	Medium	Reference-based systems
System-Wide Encryption	Reversible	Highest	Very High	Complete data protection
Automated Pipeline Encryption	Reversible	High	Medium	CI/CD integrated protection

Table 3: Encryption Implementation Approaches for PCI DSS Compliance [30, 31]

Access Control and Least Privilege Enforcement
Along with strong authentication, the PCI DSS requires that access to systems is restricted to only what is required to perform an user's job function, the business need-to-know principle. Various containers may be supported by a host kernel on the same system. This can include threats where compromising configurations at a host or orchestrator level can elevate the perpetrator's privilege or provide access to cardholder data [27]. Conformance as Code addresses roles, permissions and data classifications via host-read policies or machine-readable conformance rules. Infrastructure as Code updates that open access to databases are validated at run time against the role model definitions and data sensitivity categories [27].

Credential-related components of the Reconciler-based External Secrets Operator (ESO) focus on PCI DSS Requirements 3.6.1 and 8.6.3 (Cryptographic key management process and Cryptographic key management and credential management for application accounts). ESO syncs credentials from a native cloud secret store to Kubernetes pods [36]. Credentials are not persisted in code, config files, or container images. Rotation occurs at known schedules without requiring manual handling or memorisation, eliminating what is probably the single most common audit finding for many PCI DSS assessments: stale or manually managed credentials. Here is an example ExternalSecret resource:

Listing 3. ExternalSecret Resource Definition (Sanitized)

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: pci-service-credentials
  labels:
    compliance.pci-dss/managed: "true"
spec:
  refreshInterval: 1h
  secretStoreRef:
    name: cloud-secret-store
```

```
  kind: ClusterSecretStore
target:
  name: pci-secrets
  creationPolicy: Owner
data:
  - secretKey: db-password
    remoteRef:
      key: /pci/production/db-credentials
  - secretKey: encryption-key
    remoteRef:
      key: /pci/production/encryption-keys
```

Automated vulnerability scanning pipelines can check images for outdated libraries, misconfigurations, and known vulnerabilities prior to deployment. Security teams can run these pipelines as part of a CI/CD pipeline to prevent major security issues from being deployed [27]. In addition to blocking the use of disallowed images via kubernetes admission controllers or other orchestration framework admission controllers, container security tools can help to reduce the risk throughout the container lifecycle by scanning for vulnerabilities in the host node, the components used by the orchestrator and in the running container [27]. In DevSecOps, security champions review new features and workloads and changes in topology and containers in all stages of the container life cycle in agile development processes to ensure compliance with PCI DSS requirements [27].

Logging, Monitoring, and Immutable Audit Records

The PCI DSS requirements state that all access to cardholder data and all components of the system must be logged and that an audit trail/monitoring system that can generate an analysis of system events related to payment data exists. Compliance-as-Code enforces this by requiring that all cardholder data processing systems emit structured logs to a centralized tamper-clear logging system and that service deployments are blocked where logs would not be emitted. The containerized deployment environments' aggregated logs may be fed into centralized log management systems that act as data sinks for orchestrator events and service mesh telemetry, as well as the logs of the organizations' customary applications. Security analysts use the resultant datasets for anomaly detection and for post-incident audit trail datasets. Automated log analytics engines require a data store with a very high ingest rate and low latency to identify attacks in near real-time, such as unauthorized file access and multiple log-ins. Log retention policies can also be controlled using automated lifecycle management rules to prevent accidental deletion of log files and to archive data based on later regulatory requirements. Real-time analysis policies generate alerts automatically if suspicious access patterns, privilege escalation, or configuration change are detected that may indicate security events [27]. Continuous integration and continuous deployment (CI/CD) pipeline logging generates a record of all actions taken. CI/CD logs can be used to help meet compliance requirements and in post-breach forensic analysis [28].

The PCI DSS requires periodic security testing and penetration testing to confirm that controls are adequate in the face of evolving threats [27]. Container-based architectures need different strategies and tools because of ephemeral workloads, dynamic service discovery, and overlapping network namespaces. Host, container and orchestration plane tests can help check for host, application, and privilege escalation vulnerabilities, to avoid introducing vulnerabilities in microservices or container images that affect the confidentiality of cardholder data [27]. Automated configuration management tools such as Ansible, Puppet, and Chef can be used for this requirement [30]. This allows organisations to ensure that servers are appropriately configured and that deployments will not have any adverse consequences. Such tools automate CI/CD systems across development and production environments [30]. Besides security policies in writing, the PCI DSS also defines requirements for incident response and training. With the PCI DSS, there is

the additional goal to reach a security standard beyond that required by law, to impart higher consumer confidence in the security of e-commerce transactions [27].

A Production Case Study: Compliance-as-Code at Scale in a B2B2C Issuer-Processor

Deployment Environment

FinPlatform is a leading US issuer-processor and B2B2C platform that builds and operates APIs, ledgers, cards, card processing, compliance tooling, risk and fraud detection, and other backend services for consumer-facing fintech products. At its peak, the platform supported hundreds of millions of consumer accounts across neobanks, investment platforms, and digital banking and payments solutions. The engineering team has several thousand engineers running hundreds of microservices in Kubernetes. FinPlatform is certified as PCI DSS Level 1, the highest level.

Framework Architecture

Together, the controls form four levels, enforcing security before deployment (Config-Init) and at the time of deployment (admission controllers). Security is further enforced as a base characteristic of the infrastructure running the workload (ESO credential rotation). The rule is simple: if not compliant, the deployment is rejected. However, compliance is not something that is only required after this has happened.

Component 1: Integrate with External Secrets Operator (ESO) to meet PCI DSS requirements 3.6.1, 8.6.3. ESO syncs secrets from the native cloud secret store to Kubernetes pods. This prevents hardcoding and automates the entire secret rotation process.

Component 2: Config-Init Compliance Containers are custom init containers that run before the application container in each Kubernetes pod, validating secrets, encryption configuration, RBAC policies, and audit logging reachability. Config-Init Compliance Containers prevent the application pod from starting if the Kubernetes pod fails these checks.

Component 3: Helm Chart Abstraction Layer packages the entire compliance stack as a Helm chart dependency. Application teams need only add one line in their application-level Chart.yaml to get all of the Chart-based compliance stack resources, including Config-Init containers, ExternalSecret resources, NetworkPolicy for CDE segmentation, an audit logging sidecar, and pod security contexts:

Listing 4. Application Team Adoption (Sanitized)

```
# Application team's Chart.yaml - single addition
dependencies:
  - name: pci-compliance
    version: "2.x"
    repository:
"https://charts.internal/compliance"

# This injects automatically:
#   - Config-Init containers for pre-startup validation
#   - ExternalSecret resources for credential management
#   - NetworkPolicy for CDE segmentation
#   - Audit logging sidecar
#   - Pod security context (non-root, read-only roots)
```

Component 4: Conformance. The application of the policy is done at the Kubernetes API server with the use of a pipeline of Kubernetes Admission Controllers to disallow any resource that is not conformant from being created. This is the last backstop. If someone were to go around the Helm chart or otherwise misconfigure their manifest, Kubernetes would not accept it.

PCI DSS 4.0 Control Mapping

Table 4 shows the framework components that map to the corresponding PCI DSS 4.0 requirements and the NIST 800-218 practice groups.

PCI DSS Requirements	Requirement Description	Framework Component	NIST 800-218
3.3.2	Prevent PAN copy/relocation via remote access.	Admission controllers + network policies	PS.1
3.3.3	Encrypt stored PAN with strong cryptography	Config-Init validates encryption at startup	PW.6
3.5.1.1	Keyed cryptographic hashes per key-mgmt processes	ESO integration enforces key lifecycle	PS.2
3.6.1	Protect keys across the full lifecycle	ESO + cloud secret store, automated rotation	PS.1, PS.2
8.6.3	Application/system account password management	ESO automated credential cycling	PW.1

Table 4: PCI DSS 4.0 Requirement to Framework Component Mapping [35]

Rollout Strategy

The rollout of the framework transpired in three phases. Phase 1 started with a limited number of services that read PAN directly, for testing production readiness, and flushing out edge cases in the Config-Init logic. Services that did not follow the standard startup order or had older configuration patterns exposed unanticipated issues. The framework was subsequently expanded to cover any service that directly interacts with any credential store or encryption. In Phase 3, the Helm chart was made available platform-wide, as part of the standard toolkit. Most importantly, the Phase 3 adoption was not mandated; it followed from the sessions in which the engineers themselves found that it made their lives easier. This was important in securing adoption from the several thousand engineers across the company.

Evaluation and Results

Before the framework: FinPlatform had real compliance assurance only during the audit window itself, the few weeks out of each year during which the audit was performed. The other 48 to 50 weeks of the year were a compliance gray area. After deployment, every deployment triggers a compliance check, making the compliance exposure window drop from a matter of days-to-months to zero. During the first year that the enforcement layer was fully deployed, no non-compliant deployment made it through. Non-compliant configurations are stopped before reaching a running state.

Audit preparation: Before, teams would spend weeks preparing for an audit, gathering screenshots of configurations, evidence of credential rotations, access controls documentation, network segmentation diagrams, and manually mapping each item back to the applicable PCI DSS controls for individual systems. Now, that evidence is continuously generated in the course of normal deploying and configuring applications. Deploys produce a timestamped record of compliance in an append-only audit log.

Developer adoption and productivity: Development teams added one dependency to their Chart.yaml and received the full compliance stack with zero application code changes. Adoption tracked the three-phase rollout and the mean time-to-adoption per service is measured in hours compared to the weeks it typically takes to have the compliance manually set up.

Downstream effect: Because the framework is implemented at the issuer-processor infrastructure layer, all its encapsulated protections are automatically available to all downstream clients and their end consumers. The implementation coverage of a framework can be thought of as being approximately equivalent to that of an implementation for each client separately.

Dimension	Before (Traditional)	After (Framework)	Impact
Compliance validation	Annual point-in-time audit	Continuous, every deployment	Gaps eliminated
Time to detect violation	Days to months	Instant (blocked pre-deploy)	Zero exposure window
Developer effort	Manual remediation tickets	Zero (Helm chart adoption)	Friction eliminated
Credential management	Manual rotation, hardcoded secrets	Automated ESO rotation	Human error eliminated
Audit preparation	Multi-week manual evidence gathering	On-demand log retrieval	Always audit-ready
Scope of protection	Per-application	All platform microservices	Full downstream coverage
New service onboarding	Weeks of compliance setup	Add Helm chart dependency	Hours, not weeks

Table 5: Framework Impact - Before and After Deployment

Discussion

The Complementary Relationship between Preventive and Detective Controls

The FinPlatform production results illustrate another consideration: Compliance-as-Code preventive enforcement is not a replacement for detective GRC monitoring, such as that provided by GRC platforms and CSPM tools. Rather, it is a complement that reduces compliance exposure window and provides operational visibility. Compliance-as-Code frameworks, by contrast, eliminate the compliance exposure window completely by making the non-compliant deployment structurally impossible. Cloud-native financial institutions can combine the two layers, collecting data and alerting operationally on compliance status while preventing non-compliant configurations from ever reaching production in the first place.

The Multiplier Effect in B2B2C Infrastructure

A key lesson from the FinPlatform deployment is that once a neobank controls for compliance, it protects its own customers. Useful but bounded. When an issuer-processor implements compliance controls at the infrastructure layer, it protects the customers of every single client that comes on the platform. The economics are dramatically different. Investing in compliance engineering at the platform layer provides coverage that would otherwise be distributed over dozens of independent implementations at the application layer. If the goal is to improve PCI DSS compliance posture across the industry, the highest-leverage intervention point is not in individual fintech companies but in the platform providers.

Alignment with Government Mandates

The four-component framework proposed here implements what CISA and NIST are asking for not in spirit, but concretely. CISA's Secure-by-Design initiative, for instance, recommends that security be baked into the development process [32]. Likewise, through the Config-Init containers and the admission controllers, it is impossible to install a non-compliant deployment. NIST SP 800-218's PS and PW practice groups [33] are implemented by the Config-Init validation logic and the Helm chart abstraction layer, respectively. This has also been echoed in the U.S. Treasury Department's call for automatic controls to be put in place in the financial services industry [34] (not of course an ex post rationale, but one design goal).

Adoption as a Design Constraint

Another common theme in the Compliance-as-Code literature is that lack of adoption has been the missing link preventing compliance frameworks from reaching their potential. The FinPlatform deployment shows that frictionless adoption via abstraction (the Helm chart layer) is not simply a feature, but an architectural requirement for

Compliance-as-Code frameworks. If compliance can be expressed in one line of code rather than taking many weeks, then thousands of engineers across the enterprise can operate at the speed of software. Compliance-as-Code frameworks must, therefore, be designed with developer experience as a first-class constraint alongside correctness.

Conclusion

Moving compliance from being checked on demand to being enforced continuously throughout the software stack is a fundamental shift in security governance of a cloud-native financial infrastructure. After performing a literature review, it is observed that Compliance-as-Code addresses the extreme temporal mismatch between customary statutory periods between compliance audits and rapid software deployment cycles by encoding regulatory provisions into enforceable policies within software delivery pipelines. In this way, a deployment compliance could be considered as a governance ability that prevents a violation from reaching production, whereas a detective GRC implementation checks the compliance of the target system after it has reached the target production environment. FinPlatform, the issuer-processor of hundreds of millions of consumer accounts, is an actual real-world production implementation of this system at scale. With the four components above (ESO-based credential management, Config-Init pre-startup validation, Helm chart abstraction layer to make deployment easier, and Kubernetes admission control as defense-in-depth), it no longer matters when non-compliant deployments are discovered, such deployments cannot happen, and subsequently there are no compliance exposure windows between violations and detection. Audit preparation went from weeks to on-demand. Because developers were able to adopt the framework as is (zero code changes), and FinPlatform itself is considered B2B2C infrastructure, this coverage cascades to all other downstream clients and end consumers. This multiplier effect is meaningful, with real implications for how the industry thinks about PCI DSS enforcement. Compliance-as-Code provides an architectural pattern for regulated financial institutions that want to achieve PCI DSS compliance without sacrificing velocity or security. Policy engines, pipeline integration and audit trail patterns treat policy and compliance as infrastructure code, binding them to regulatory requirements and the software delivery lifecycle.

Limitations

The design is Kubernetes-specific and is not straightforward to port to VMs, serverless-based (Lambda, Azure Functions, etc.), or hybrid deployments. Organizations using those infrastructures would have to port the enforcement boundary to their own infrastructure, and maintain Config-Init's validation logic. This means that any policies based on PCI DSS also need to be updated and that failure to do so could create a false sense of security that is arguably worse than no framework at all. The Helm chart abstraction works well with microservices but breaks down on edge cases such as batch jobs, cron jobs and monoliths. For these cases, custom integration is required, partially weakening the zero-code adoption story. Technical controls are not included in the scope of the framework. Organizational requirements such as security awareness training (Requirement 12.6), incident response planning (Requirement 12.10), and physical access (Requirement 9) are excluded. Compliance is never solely a technical problem.

As with all literature reviews, the included studies are a finite set of completed work within the scope. Further, note that this review is limited to published work between 2020 and 2025, so practices and tooling may not be considered. Furthermore, this work targets cloud-native architectures, which may not be applicable to places with large legacy applications, where container- and infrastructure as code-based approaches are still in their infancy.

Future Scope

Further research is needed to conduct longitudinal studies on the transformation of organizations using Compliance-as-Code to identify cultural barriers, skills, and change management approaches necessary for the successful adoption and usage of Compliance-as-Code to aid compliance management. Empirical studies on the use of new AI and ML approaches to enable predictive detection of compliance drift and automated policy recommendations are also needed. Research on federated policy enforcement (where policy definition and enforcement is applied on all data, possibly even across organizations e.g., in third-party risk management), would address some of the limitations mentioned above. Finally,

with the growth of quantum computing and the state of post-quantum cryptography maturing, it would be interesting to study the applicability of post-quantum crypto on Compliance-as-Code implementations. Economic studies of tooling cost, training cost, cultural change cost, reduced audit review cost, reduced violation penalty cost and reduced breach response cost would aid in the business case for Compliance-as-Code.

References

[1] A. S. Mollashaik, "Understanding PCI DSS V4.0: A Comprehensive Guide to Payment Security Compliance," Authorea Preprints, 2025.

[2] PCI Security Standard Council LLC, "PCI DSS: v4.0.1," 2024. Available: https://docs-prv.pcisecuritystandards.org/PCI%20DSS/Standard/PCI-DSS-v4_0_1.pdf

[3] ISO, "ISO 19011:2018," 2018. Available: https://www.iso.org/standard/70017.html

[4] Weaveworks, "What is cloud native and why does it exist?," 2017. Available: https://www.cncf.io/online-programs/what-is-cloud-native-and-why-does-it-exist/

[5] Cloud Security Alliance, "Top Threats to Cloud Computing: Pandemic Eleven," CSA, 2022. Available: https://cloudsecurityalliance.org/artifacts/top-threats-to-cloud-computing-pandemic-eleven

[6] D. S. Antiya, "Compliance as Code: Automating Compliance in Cloud Systems," Int. J. Recent Innov. Trends Comput. Commun., 2020.

[7] IBM Security, "Cost of a Data Breach Report 2024," IBM/Ponemon Institute, 2024. Available: https://www.ibm.com/reports/data-breach

[8] Google Cloud DORA Team, "Accelerate State of DevOps Report 2024," Google, 2024. Available: https://dora.dev/research/2024/dora-report/

[9] Uptime Institute, "Annual Outage Analysis 2024," Uptime Institute, 2024. Available: https://uptimeinstitute.com/resources/research-and-reports/annual-outage-analysis-2024

[10] I. A. Essien et al., "Third-party Vendor Risk Assessment and Compliance Monitoring Framework for Highly Regulated Industries," Int. J. Multidisc. Res. Growth Eval., 2021.

[11] A. Zakharchenko, "Integrating Continuous Compliance into DevSecOps Pipelines: A Data Engineering Perspective," Softw. Reliab. Secur. Qual. Assur., vol. 5, no. 1, 2026.

[12] Open Policy Agent, "OPA Documentation," 2025. Available: https://www.openpolicyagent.org/

[13] M. Kansara, "A Structured Lifecycle Approach to Large-Scale Cloud Database Migration," Appl. Res. Artif. Intell. Cloud Comput., 2022.

[14] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland, OR: IT Revolution Press, 2018.

[15] F. Moyón et al., "Aligning Security Compliance and DevOps: A Longitudinal Study," J. Syst. Softw., 2025. Available: https://arxiv.org/pdf/2512.14453

[16] C. Mavani, "Security-as-Code: Enforcing Cybersecurity Standards through Automated Governance in Cloud Pipelines," IJCET, vol. 16, no. 5, 2025.

[17] R. Gouni, "Automating Compliance in DevOps Pipelines," Int. J. Comput. Exp. Sci. Eng. (IJCESEN), 2025.

[18] S. K. Suvvari, "Ensuring Security and Compliance in Agile Cloud Infrastructure Projects," Int. J. Comput. Eng., 2024.

[19] Styra, "Open Policy Agent," 2025. Available: https://www.openpolicyagent.org/

[20] B. Di Martino et al., "Review of Policy-as-Code Approaches to Manage Security and Privacy in Edge and Cloud Ecosystems," in Proc. AINA, Springer, 2025.

[21] A. Awasthi, "GRC Automation in Manufacturing: Modernizing Compliance and Risk Management," Int. J. Comput. Eng. Technol., 2025.

[22] O. M. Ijiga et al., "Blockchain-Integrated Logging Mechanisms for Ensuring Integrity and Auditability in Relational Database Transactions," Int. J. Soc. Sci. Humanit. Res., 2025.

[23] NIST, "Risk Management Framework for Information Systems and Organizations," NIST SP 800-37 Rev. 2, 2018. Available: https://csrc.nist.gov/publications/detail/sp/800-37/rev-2/final

[24] F. Binbeshr and M. Imam, "Comparative Analysis of AI-driven Security Approaches in DevSecOps," in Proc. 29th EASE, ACM, 2025.

[25] A. S. A. Alghawli and T. Radivilova, "Resilient Cloud Cluster with DevSecOps Security Model," Alexandria Eng. J., vol. 99, pp. 222–236, 2024.

[26] M. Kellogg et al., "Continuous Compliance," in Proc. 35th IEEE/ACM ASE, ACM, 2020.

[27] P. Q. Bao, "Assessing PCI DSS Compliance in Virtualized, Container-Based E-Commerce Platforms," J. Appl. Cybersecur. Analytics, 2022.

[28] E. Bonner et al., “Implementing the Payment Card Industry Data Security Standard,” TELKOMNIKA, 2011.
[29] S. J. Owoade et al., “Cloud-based Compliance and Data Security Solutions in Financial Applications Using CI/CD Pipelines,” World J. Eng. Technol. Res., 2024.
[30] J. I. Akerele et al., “Increasing Software Deployment Speed in Agile Environments through Automated Configuration Management,” Int. J. Eng. Res. Updates, 2024.
[31] PCI Security Standards Council, “PCI DSS v4.0,” PCI SSC, 2022. Available: https://www.pcisecuritystandards.org/about_us/press_releases/securing-the-future-of-payments-pci-ssc-publishes-pci-data-security-standard-v4-0/
[32] CISA, “Secure-by-Design: Shifting the Balance of Cybersecurity Risk,” 2023. Available: https://www.cisa.gov/sites/default/files/2023-10/Shifting-the-Balance-of-Cybersecurity-Risk-Principles-and-Approaches-for-Secure-by-Design-Software.pdf
[33] M. Souppaya et al., “Secure Software Development Framework (SSDF) Version 1.1,” NIST SP 800-218, 2022. Available: https://csrc.nist.gov/pubs/sp/800/218/final
[34] U.S. Department of the Treasury, “Managing AI-Specific Cybersecurity Risks in the Financial Sector,” 2024.
[35] F. Ekundayo, “Strategies for Managing Data Engineering Teams to Build Scalable REST APIs for FinTech,” Int. J. Eng. Technol. Res. Manag., 2023.
[36] External Secrets Operator, “ESO Documentation,” 2025. Available: https://external-secrets.io/
[37] OWASP Foundation, “OWASP Top Ten 2021,” OWASP, 2021. Available: https://owasp.org/www-project-top-ten/
[38] NIST, “Security and Privacy Controls for Information Systems and Organizations,” NIST SP 800-53 Rev. 5, 2020. Available: https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final
[39] CNCF, “CNCF Annual Survey 2023,” Cloud Native Computing Foundation, 2024. Available: https://www.cncf.io/reports/cncf-annual-survey-2023/