ComplianceService Technical Proposal

> **Document Type:** Technical Proposal and Architecture Design **Service Name:** ComplianceService **Status:** Proposed **Date:** February 2026 **Target Delivery:** March 31, 2026 **Audience:** Engineering Leadership, Security Team, DevOps, Architecture Review Board

Table of Contents

1. Introduction

1.1 Purpose

This document proposes **ComplianceService**, a centralized policy gateway designed to automate security compliance decisions within CI/CD pipelines. The service operates as an intermediary between security scanning tools such as Snyk, Prisma Cloud, and other security tools and the deployment targets they protect. It provides a deterministic **allow or deny decision** for every deployment based on organizational compliance policies expressed as code.

1.2 Scope

ComplianceService encompasses the following functional domains:

Area	Description
Compliance Evaluation	Ingests structured security scan output from pipeline runners and evaluates that data against Rego policy bundles using the Open Policy Agent runtime in real time
Application Management	Provides a registration and configuration surface for applications, supporting per environment risk tier assignment and policy binding
Audit Trail	Persists every compliance decision alongside the full evidence chain including raw scan input, policy engine payload, and engine response for regulatory and internal review

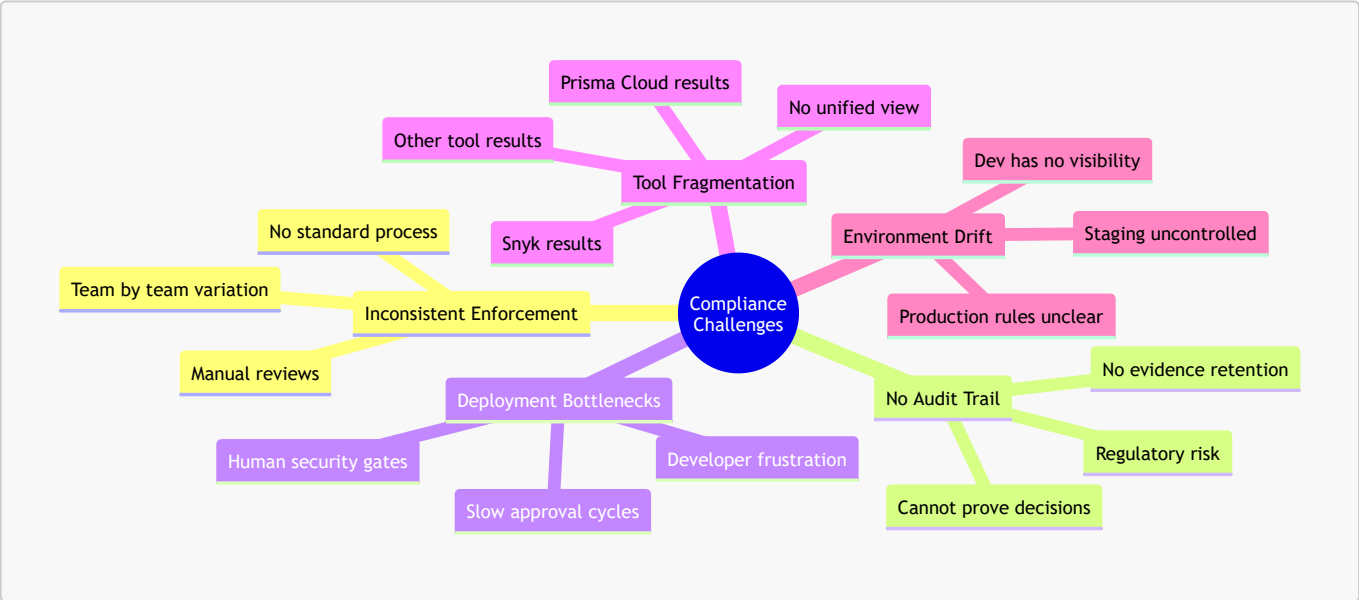
Area	Description
Alerting	Dispatches notifications to application owners and security stakeholders when a deployment is blocked or when critical severity vulnerabilities are detected
Reporting	Exposes aggregate compliance statistics across the entire application portfolio for dashboards and executive reporting

1.3 Key Stakeholders

Stakeholder	Interest
DevOps and Platform Engineering	Pipeline integration, automated deployment gating
Application Security	Policy authoring, vulnerability lifecycle tracking
Compliance and GRC	Audit trail integrity, regulatory evidence generation
Engineering Teams	Self service application registration, evaluation result visibility
Engineering Leadership	Portfolio wide security posture and trend reporting

2. Problem Statement and Business Case

2.1 Current Challenges

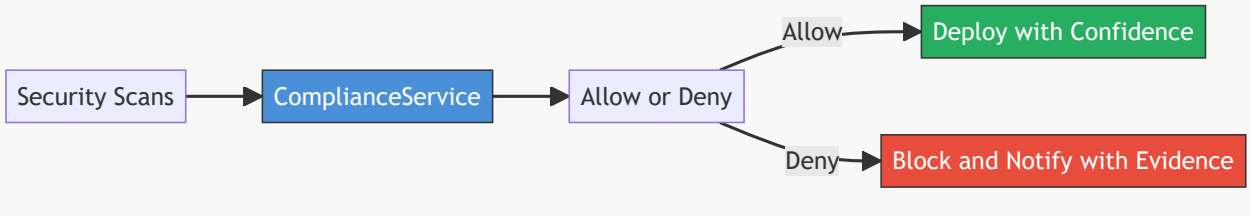


2.2 Business Impact

Impact Area	Without ComplianceService	With ComplianceService
Deployment speed	Hours or days waiting for manual approval	Seconds with automated allow or deny
Policy consistency	Varies by team, individual, and day	Identical rules applied to every deployment uniformly

Impact Area	Without ComplianceService	With ComplianceService
Audit readiness	Manual evidence gathering across multiple tools	Instant access to every decision with full evidence
Vulnerability visibility	Fragmented across individual scanner dashboards	Single consolidated portfolio wide view
Regulatory compliance	Difficult to demonstrate controls to auditors	Immutable timestamped decision records always available
Risk management	Reactive discovery after deployment	Preventive enforcement before deployment reaches infrastructure

2.3 Value Proposition



One API call. Consistent policy enforcement. Full audit trail. Zero manual intervention.

---

3. Proposed Solution

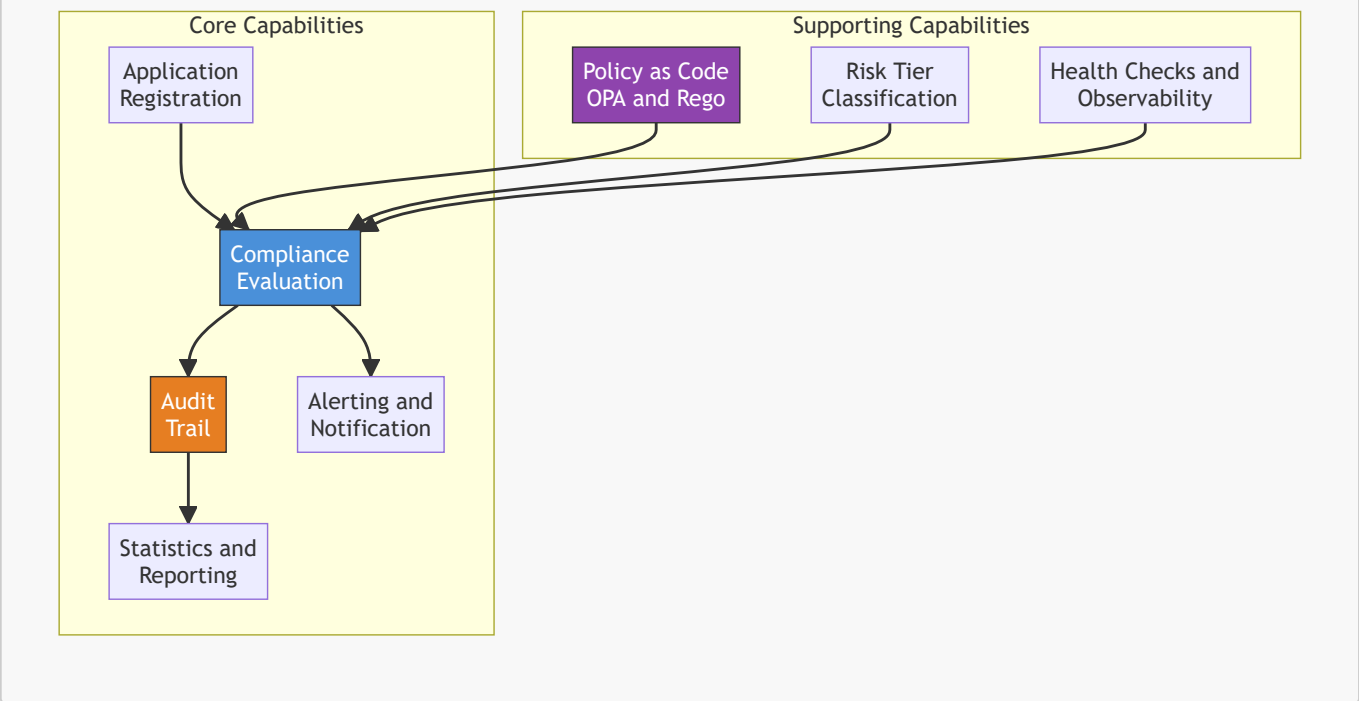
3.1 Solution Summary

ComplianceService is a RESTful API that provides automated compliance gating for CI/CD pipelines. When a pipeline reaches the security phase it sends scan results from one or more tools to the service over HTTPS. The service resolves the appropriate policy for the given application and environment, delegates the evaluation to Open Policy Agent running as a local sidecar, persists the outcome as an immutable audit record, and returns a structured response that the pipeline uses to proceed or halt.

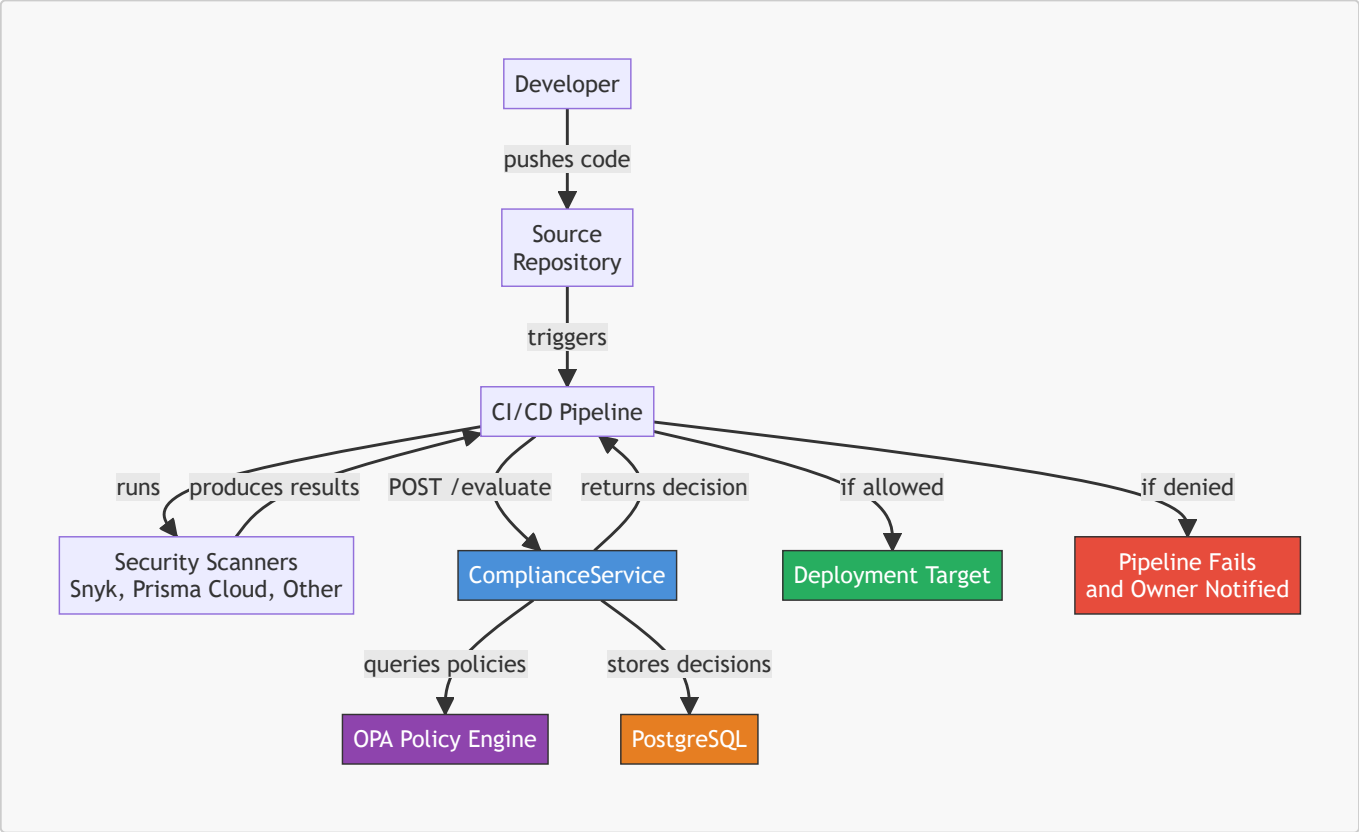
The core workflow consists of five stages:

1. **Receive** security scan results from the calling pipeline
2. **Evaluate** those results against environment specific Rego policies via OPA
3. **Decide** whether the deployment is allowed or denied based on policy output
4. **Record** the complete decision with full evidence into an immutable audit log
5. **Notify** stakeholders asynchronously when deployments are blocked or critical vulnerabilities are found

3.2 Core Capabilities



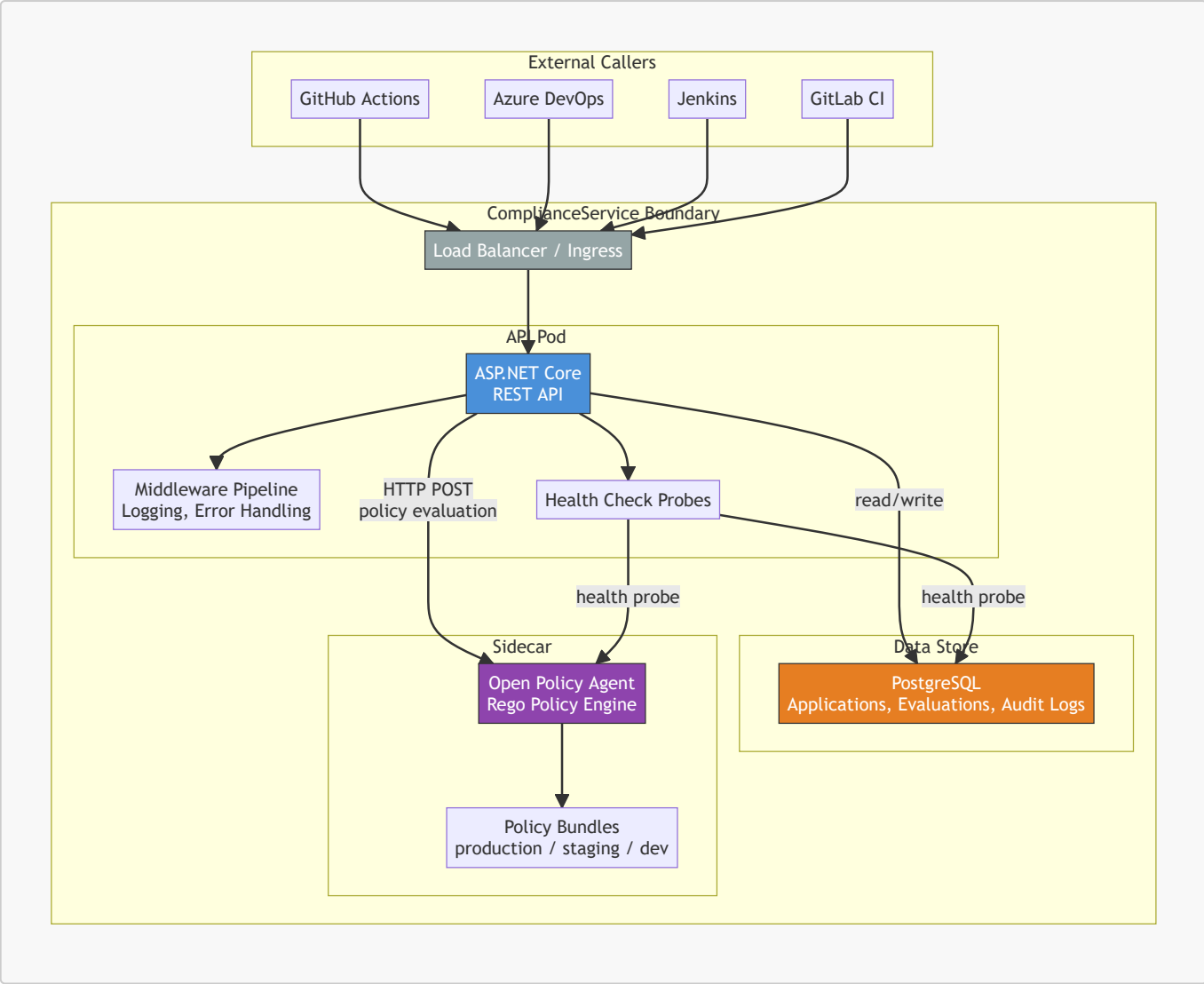
3.3 How It Fits Into the Ecosystem



4. High Level System Architecture

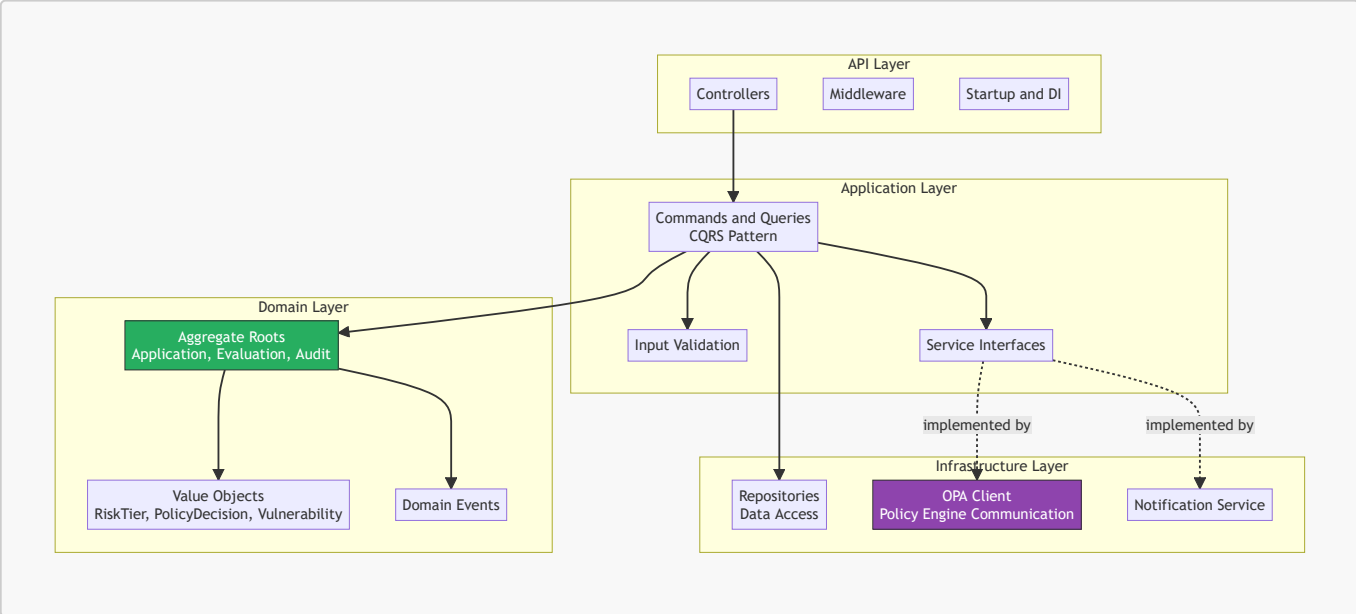
4.1 System Component Diagram

The service boundary contains three primary components: the API application, an OPA sidecar responsible for policy evaluation, and a PostgreSQL database that stores application profiles, evaluation outcomes, and audit records. External callers are CI/CD pipeline runners that communicate with the service over HTTPS through a load balancer or ingress controller.



4.2 Internal Layer Architecture (Clean Architecture)

The service is structured using Clean Architecture with four layers. Each layer has a strict inward dependency rule meaning it can only reference the layer directly beneath it. The Domain layer sits at the center with zero external dependencies, ensuring business logic remains isolated from infrastructure concerns.

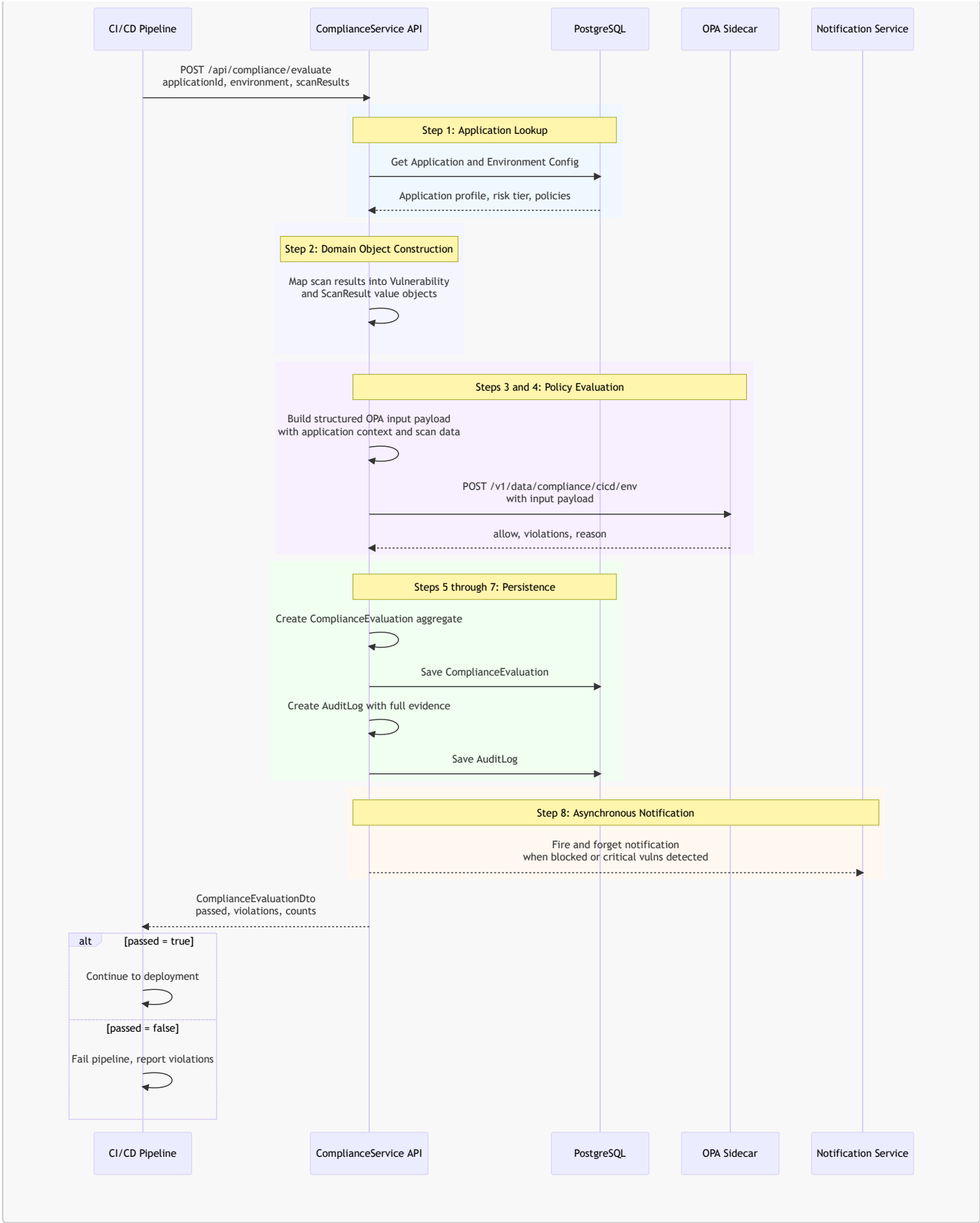


Layer	Responsibility
API	Accepts HTTP requests, applies middleware for structured logging and global error handling, configures dependency injection and health probes
Application	Orchestrates use cases through CQRS commands and queries, validates inbound request contracts, defines service interfaces consumed by infrastructure
Domain	Encapsulates all business rules within aggregate roots, value objects, and domain events with no dependency on any external framework or library
Infrastructure	Provides concrete implementations for database persistence via Entity Framework Core, HTTP communication with the OPA sidecar, and notification dispatch

5. Service Flow and Process Design

5.1 Primary Flow: Compliance Evaluation

This is the core workflow that executes every time a CI/CD pipeline submits scan results for evaluation. The entire flow is synchronous from the caller's perspective and typically completes within hundreds of milliseconds.

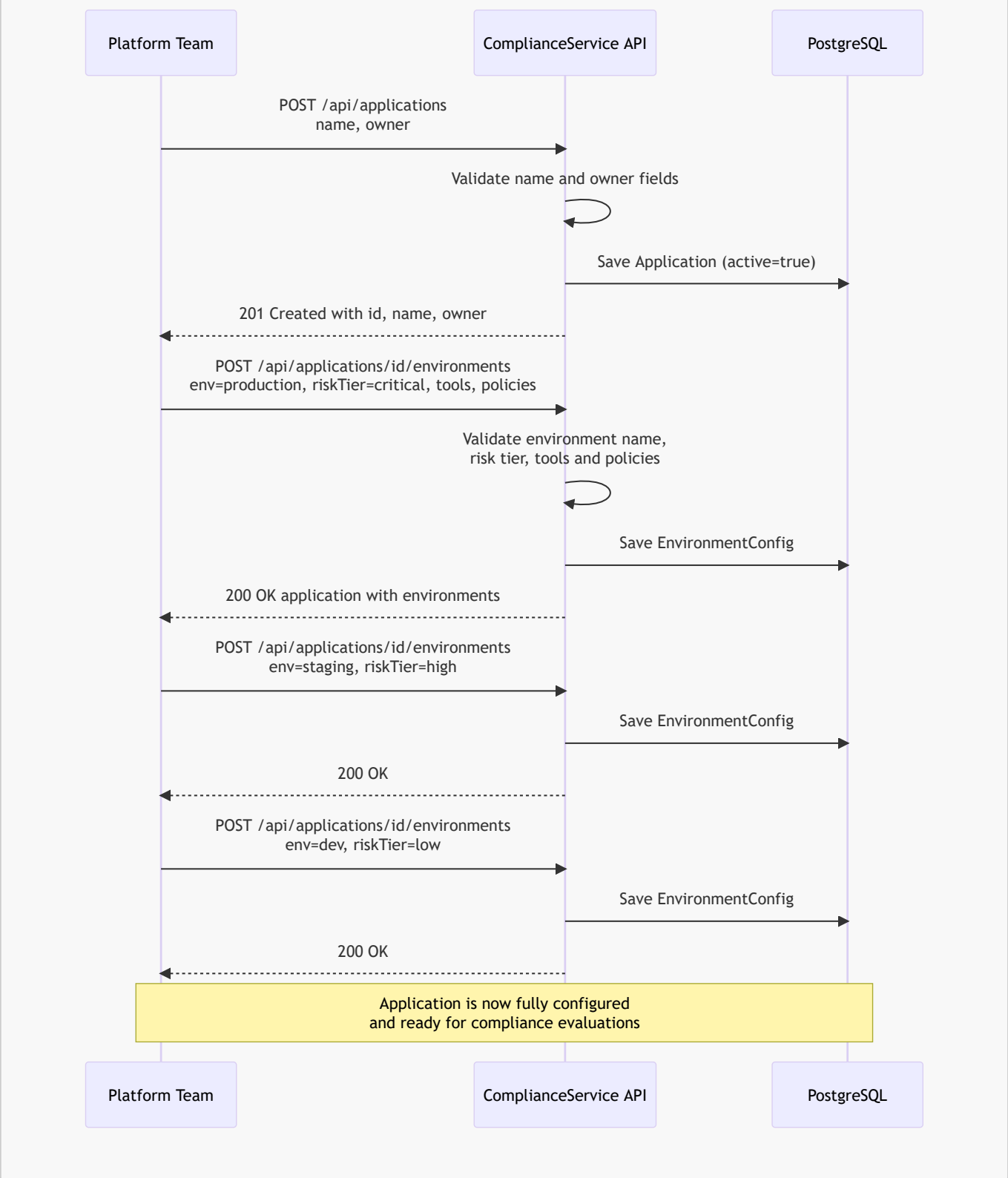


5.2 Evaluation Process Detail

Step	Action	Description

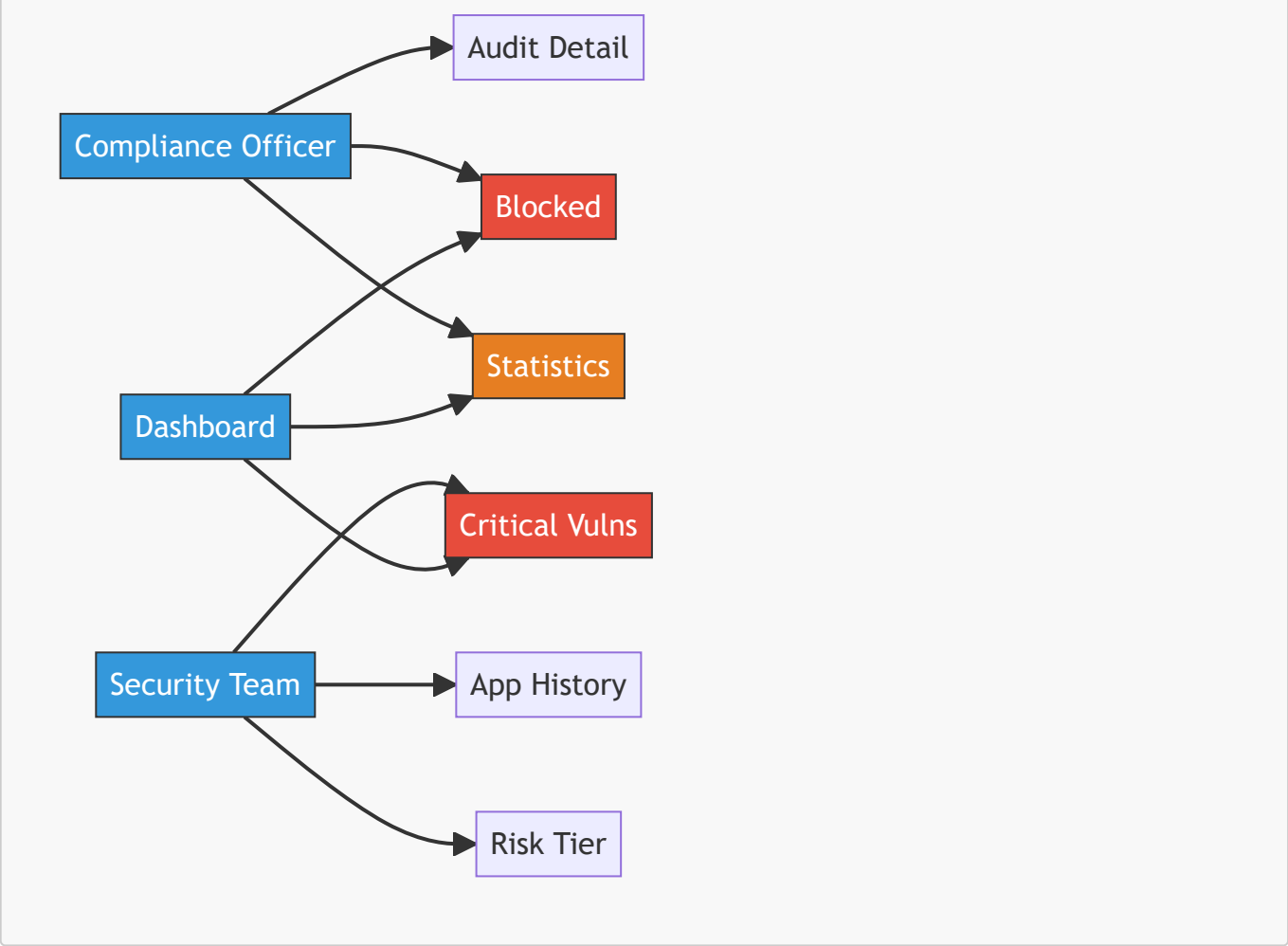
Step	Action	Description
1	**Application Lookup**	Queries PostgreSQL for the registered application profile. Validates that the application is in an active state and retrieves the environment configuration which includes the assigned risk tier, bound policy references, and required security tooling.
2	**Domain Object Construction**	Deserializes each scan result into strongly typed ScanResult and Vulnerability value objects within the domain layer. Performs structural validation on CVE identifiers, severity classification levels, CVSS scores within the 0 to 10 range, and package metadata fields.
3	**OPA Input Assembly**	Constructs a normalized JSON payload that conforms to the OPA input contract. This payload contains the full application context including name, environment, risk tier, and owner alongside the structured scan data from all submitted tools.
4	**OPA Sidecar Evaluation**	Issues an HTTP POST to the OPA Data API at the resolved policy package path. OPA evaluates the applicable Rego rules and returns a response containing the allow boolean, a violations array with individual rule messages, and a human readable reason string.
5	**PolicyDecision Creation**	Maps the OPA response into a PolicyDecision domain value object. Enforces the domain invariant that a deny decision must always carry at least one violation entry to ensure every denial is explainable.
6	**Evaluation Persistence**	Constructs the ComplianceEvaluation aggregate root containing the full scan results and policy decision then persists it through the repository layer.
7	**Audit Log Creation**	Builds an immutable AuditLog record that captures three layers of evidence: the raw scan results JSON exactly as received, the exact OPA input payload that was sent, and the exact OPA output response that was returned. Also records aggregated vulnerability counts and the final decision.
8	**Notification Dispatch**	If the deployment was denied or critical severity vulnerabilities were detected the service dispatches a notification to the application owner. This is a fire and forget asynchronous operation that does not block the response to the caller.
9	**Response Return**	Returns the evaluation result to the pipeline containing the pass or fail decision, detailed violation messages, and aggregated vulnerability counts by severity level.

5.3 Application Registration Flow



5.4 Audit Query and Reporting Flow

The audit and reporting capabilities are consumed by three primary personas. Compliance officers use aggregate statistics and blocked deployment lists to prepare for audits. Security teams drill into critical vulnerability data and risk tier views to track remediation progress. Dashboards and BI tools pull from all three endpoints to render portfolio level visualizations.

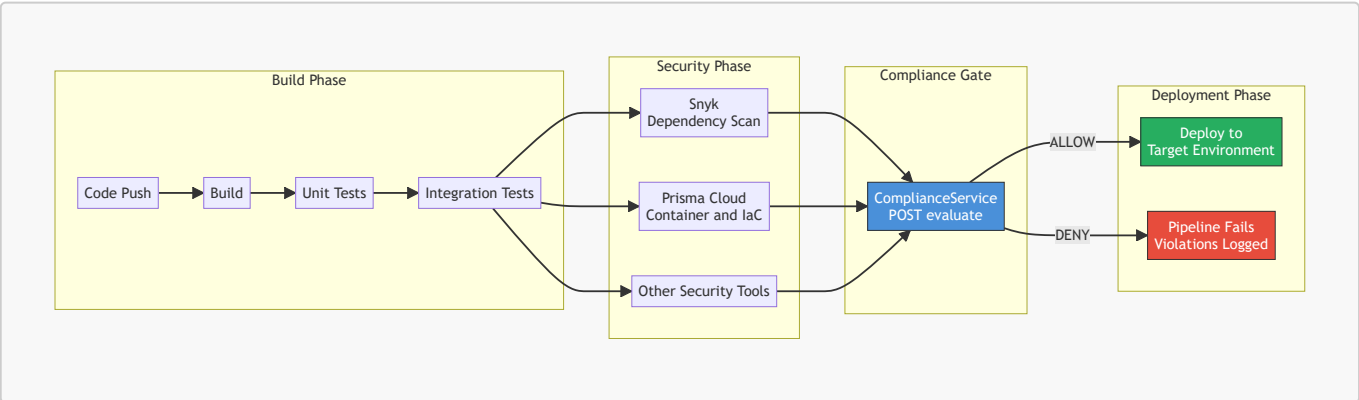


---

6. CI/CD Pipeline Integration Model

6.1 Pipeline Gate Architecture

ComplianceService operates as a **quality gate** within the CI/CD pipeline. It is positioned after the security scanning phase and before the deployment phase. All scan results from the security tools are aggregated and submitted to ComplianceService in a single POST request. The service evaluates the combined results against the applicable policy and returns a deterministic allow or deny decision that the pipeline uses to proceed or halt execution.



6.2 Integration Contract

The integration contract is designed to be tool agnostic. Any security scanner can submit results as long as they conform to the scan result schema. This allows organizations to add or replace scanning tools without modifying the ComplianceService API.

Request (from CI/CD pipeline):

```
{
  "applicationId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "environment": "production",
  "initiatedBy": "github-actions@org.com",
  "scanResults": [
    {
      "toolName": "snyk",
      "scannedAt": "2026-02-11T10:30:00Z",
      "vulnerabilities": [
        {
          "cveId": "CVE-2026-1234",
          "severity": "critical",
          "cvssScore": 9.8,
          "packageName": "lodash",
          "currentVersion": "4.17.20",
          "fixedVersion": "4.17.21"
        }
      ],
      "rawOutput": "{...}"
    }
  ],
  "metadata": {
    "pipelineId": "run-12345",
    "commitSha": "abc123"
  }
}
```

Response (to CI/CD pipeline):

```
{
  "id": "eval-uuid",
  "applicationId": "a1b2c3d4-...",
  "applicationName": "payment-service",
  "environment": "production",
  "passed": false,
  "policyDecision": {
    "allow": false,
    "violations": [
      {
        "rule": "no_critical_vulnerabilities",
        "message": "Production deployments must have zero critical
vulnerabilities. Found: 1",
        "severity": "critical"
      }
```

```
      ],
      "policyPackage": "compliance.cicd.production",
      "reason": "Production compliance violations detected"
    },
    "aggregatedCounts": {
      "critical": 1,
      "high": 0,
      "medium": 3,
      "low": 7,
      "total": 11
    }
  }
```


6.3 Pipeline Decision Matrix

This matrix defines the enforcement behavior for each environment. Production enforces zero tolerance on critical and high severity findings. Staging allows limited high severity findings while still blocking critical issues. Development operates in a permissive advisory mode that tracks vulnerabilities without blocking deployments.

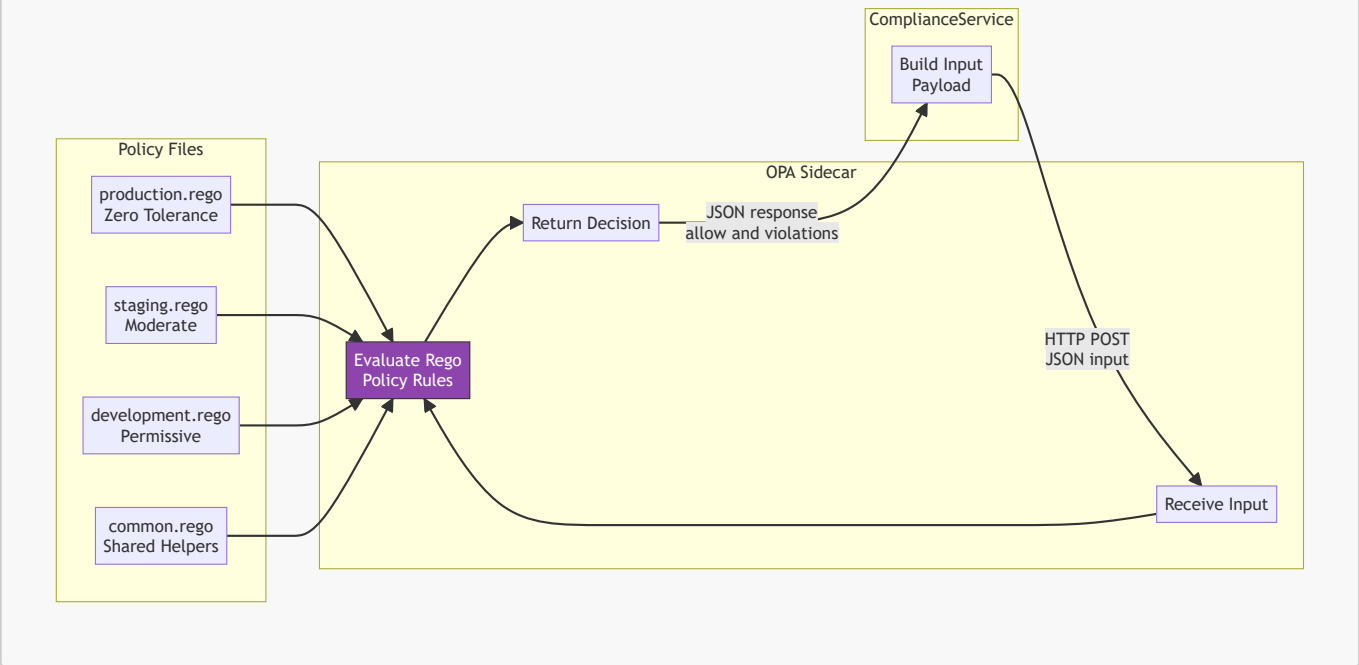
Condition	Production	Staging	Development
Critical vulnerabilities greater than 0	**DENY**	**DENY**	Allow (up to 5)
High vulnerabilities greater than 0	**DENY**	Allow (up to 3)	Allow
Missing required scan tools	**DENY**	**DENY**	Warn only
High severity license violations	**DENY**	Allow	Allow
No scans submitted	**DENY**	**DENY**	Warn only
All checks pass	**ALLOW**	**ALLOW**	**ALLOW**

7. Open Policy Agent (OPA) as the Decision Engine

7.1 What OPA Does

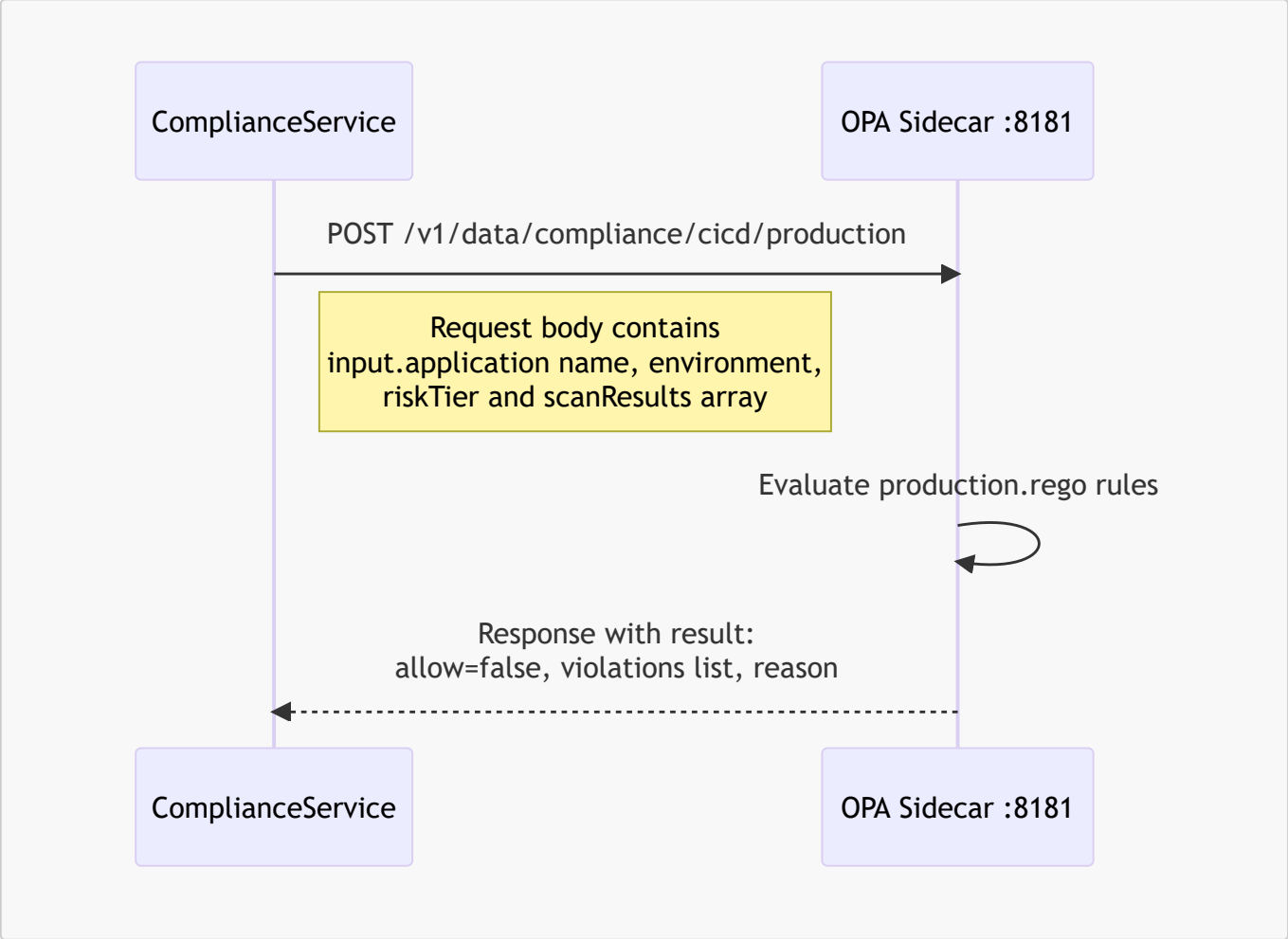
Open Policy Agent is a general purpose policy engine that decouples policy decisions from application code. Rather than embedding compliance logic directly into the service, ComplianceService delegates all decision making to OPA. The service constructs a structured JSON payload representing the current evaluation context and submits it to OPA over a local HTTP interface. OPA evaluates the input against Rego policy files and returns a deterministic decision that the service then records and returns to the caller.

This separation means policy rules can be authored, reviewed, tested, and deployed independently of the service codebase.



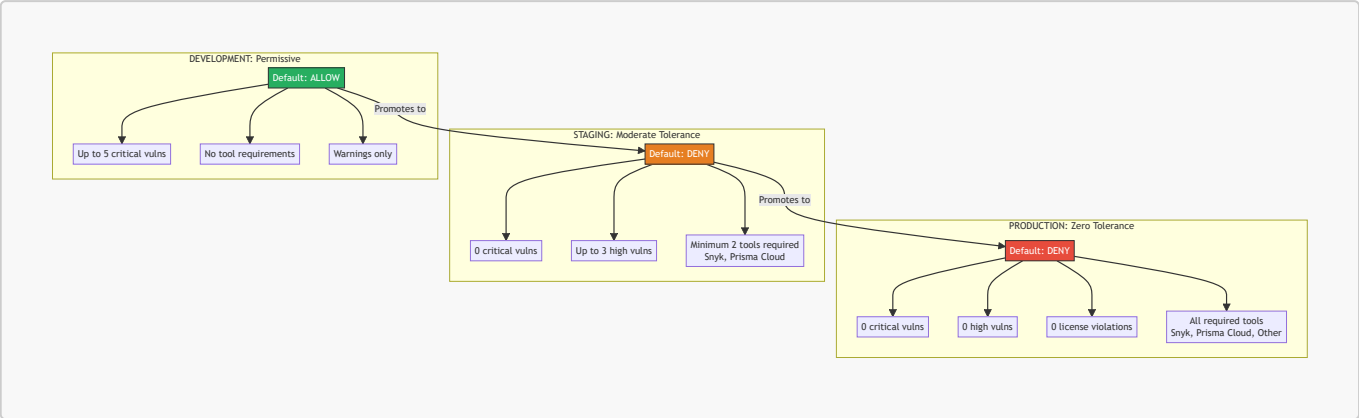
7.2 OPA Communication Pattern

The API communicates with OPA over localhost on port 8181. Because OPA runs as a sidecar within the same pod there is no network hop and latency is minimal, typically under 10 milliseconds for policy evaluation.



7.3 Policy Strictness Hierarchy

Each environment has a dedicated Rego policy file with escalating strictness. Development is permissive and advisory. Staging enforces critical severity blocking while allowing limited high severity findings. Production enforces zero tolerance across all critical and high severity categories and requires all configured scanning tools to have executed.



7.4 Policy as Code Benefits

Expressing compliance rules as Rego code rather than application logic provides several operational advantages:

Benefit	Description
Version controlled	Rego files are stored in Git alongside the service so every change is tracked with full commit history
Reviewable	Policy changes follow the same pull request and peer review workflow as application code
Testable	OPA includes a built in test framework that validates Rego rules against known inputs and expected outputs
Decoupled	Policies can be updated and deployed independently without rebuilding or redeploying the API service
Auditable	Git history provides a complete record of who changed what policy, when it was changed, and why
Declarative	Rules express what should be enforced rather than how the enforcement logic should execute

8. Policy Management at Scale

8.1 The Scaling Challenge

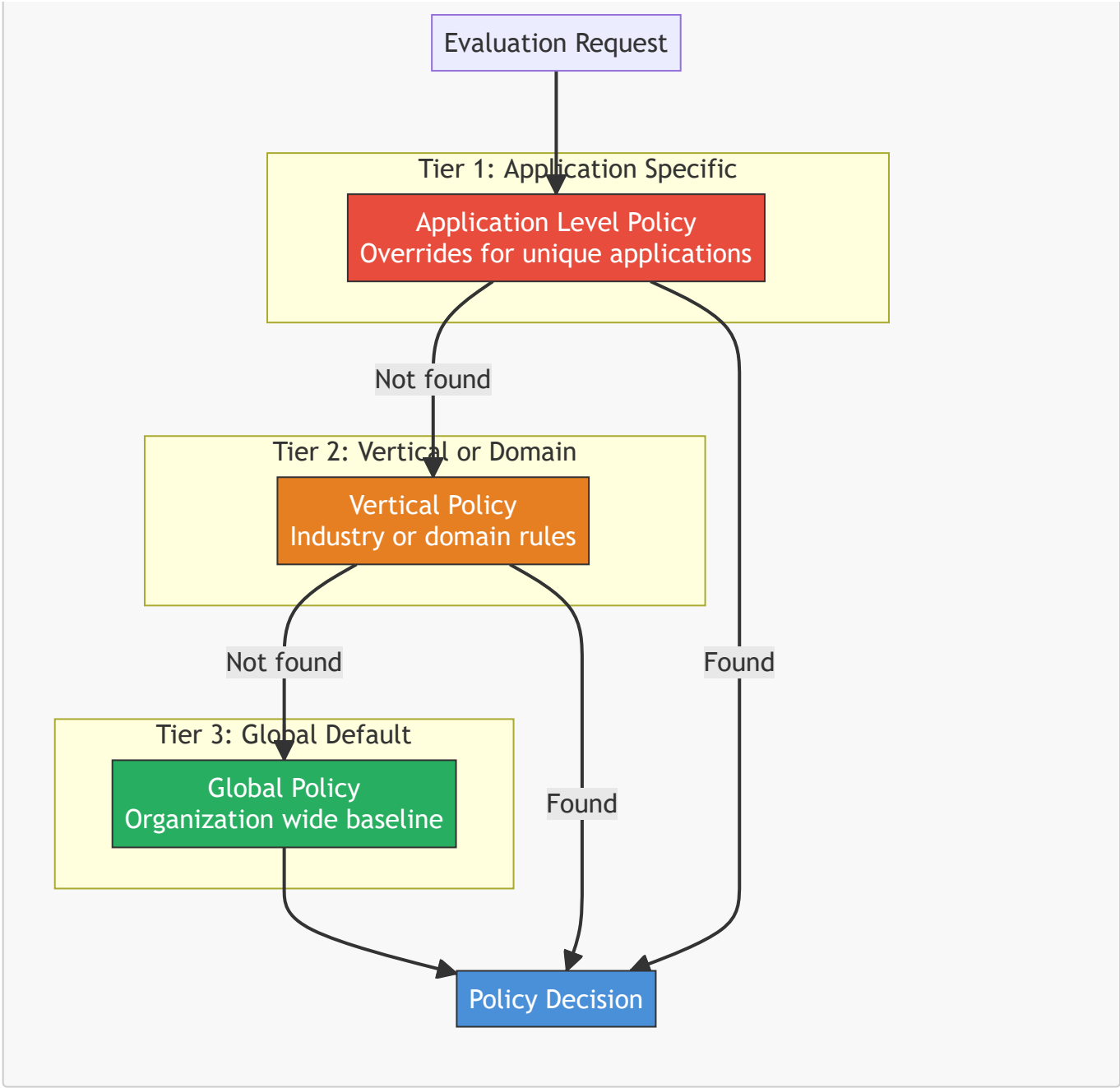
A flat policy directory with three environment specific Rego files works well for a small number of applications. However as the organization onboards more applications this approach introduces operational overhead that becomes increasingly difficult to manage.

Challenge	Impact at Scale

Challenge	Impact at Scale
Manual policy assignment per application and environment	Unsustainable when every new application requires manual setup
No automatic policy selection by risk tier	Misconfiguration risk grows across hundreds of environment configurations
Flat policy namespace	No organizational structure by business domain or team
Volume mount policy loading	Requires OPA container restart for any policy change
Single policy per evaluation	Cannot layer organization wide rules with application specific overrides
No policy versioning	Cannot roll back a problematic policy without redeploying the entire service

8.2 Proposed Architecture: Hierarchical Policy Resolution

The recommended approach introduces a **three tier policy hierarchy** where the system resolves the correct policy by walking from the most specific match to the least specific. The first match in the chain wins and is used for evaluation.

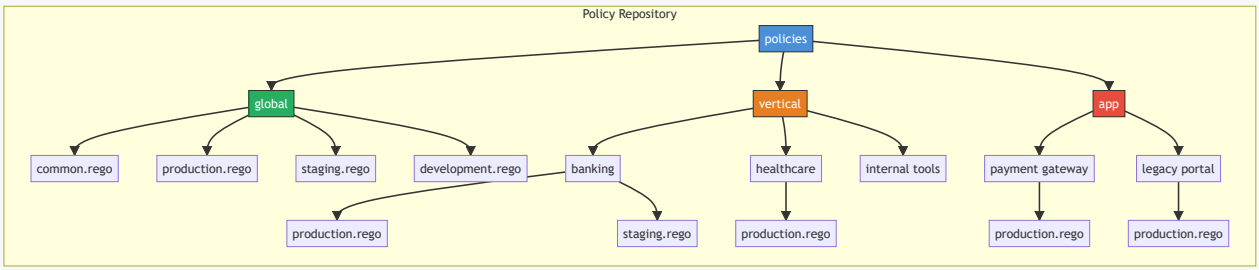


Resolution chain for each evaluation:

```
1. Application specific   -->  app/{app-name}/{environment}.rego
2. Vertical               -->  vertical/{vertical}/{environment}.rego
3. Global default          -->  global/{environment}.rego
```

Most applications (80 to 90 percent) will fall through to the global default, meaning they require zero per application policy configuration.

8.3 Policy Directory Structure

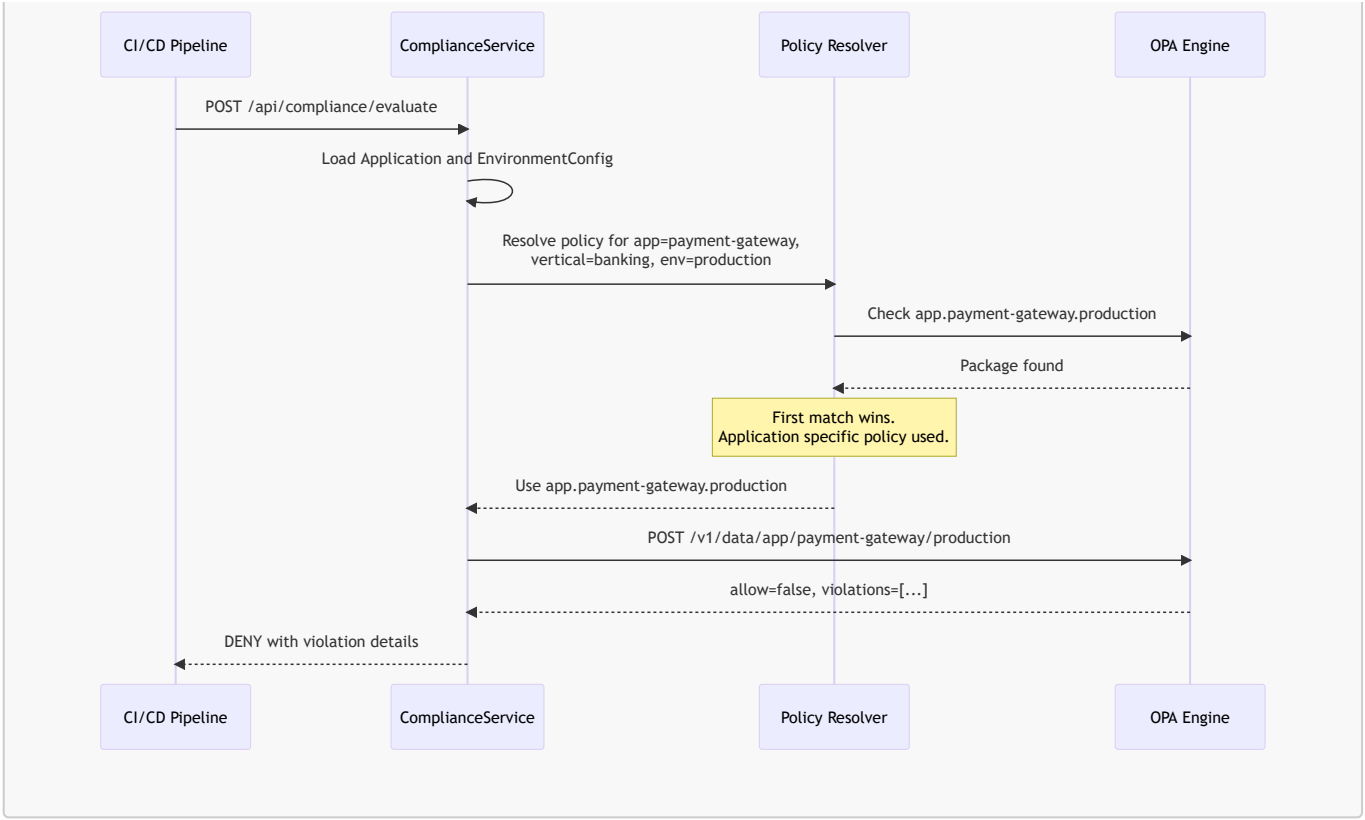


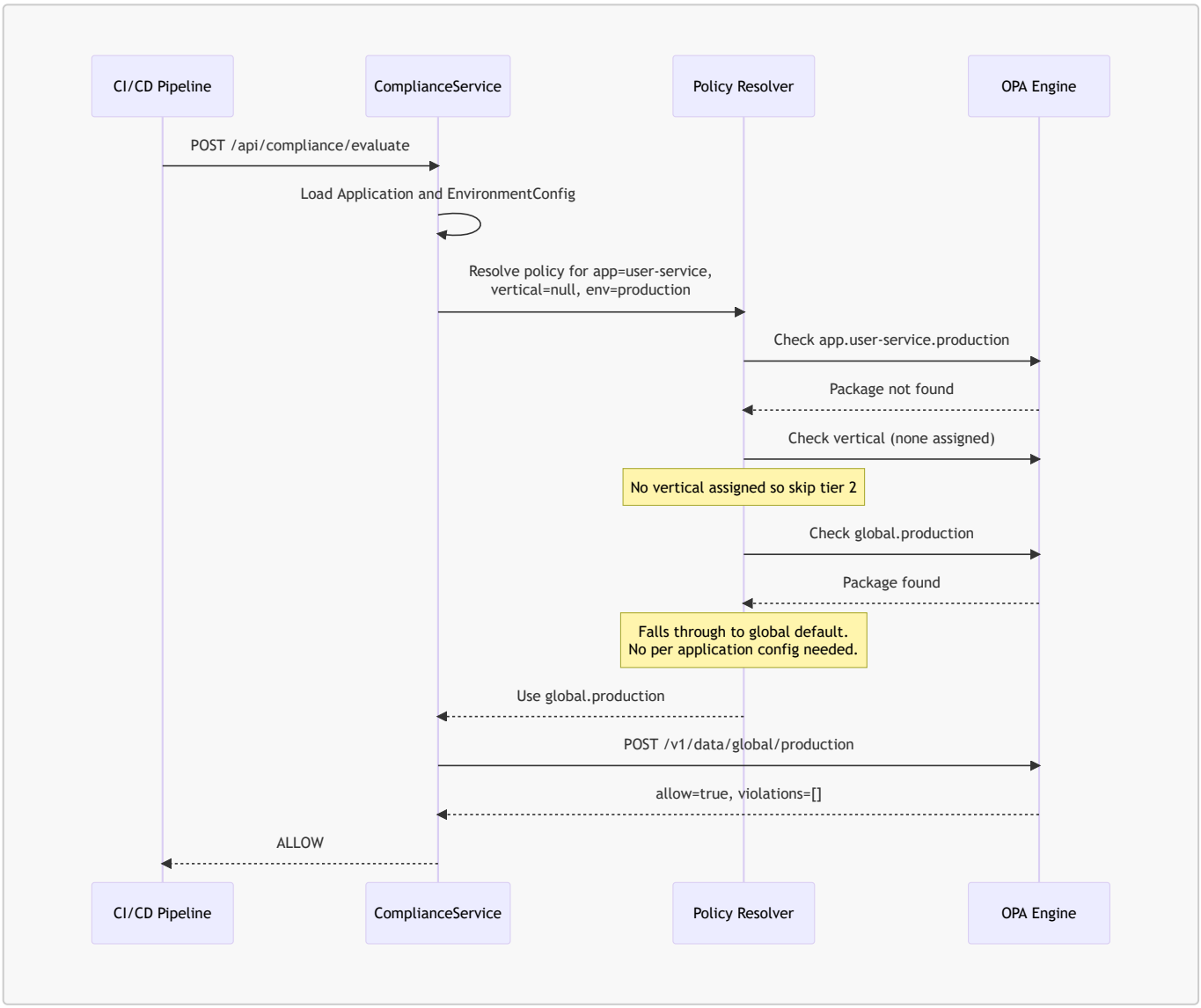
Directory layout:

```
policies/
  global/                          # Organization wide baseline (every app
inherits)
    common.rego                    # Shared helper functions
    production.rego                # Default production rules
    staging.rego                   # Default staging rules
    development.rego               # Default development rules

  vertical/                        # Industry and domain overrides
    banking/
      production.rego              # Stricter PCI DSS and SOX compliance rules
      staging.rego                 # Banking staging rules
    healthcare/
      production.rego              # HIPAA requirements
      staging.rego                 # Healthcare staging rules
    internal-tools/
      production.rego              # Relaxed rules for internal only
applications

  app/                             # Application specific overrides (rare)
    payment-gateway/
      production.rego              # Zero tolerance with custom CVE blocklist
    legacy-portal/
      production.rego              # Temporary exception waivers
```

8.4 Policy Resolution Flow

The following two sequence diagrams illustrate how policy resolution works in practice. The first shows an application that has its own specific policy. The second shows a standard application that falls through to the global default.


CI/CD Pipeline
ComplianceService
Policy Resolver
OPA Engine
POST /api/compliance/evaluate
Load Application and EnvironmentConfig
Resolve policy for app=payment-gateway, vertical=banking, env=production
Check app.payment-gateway.production
Package found
First match wins.
Application specific policy used.
Use app.payment-gateway.production
POST /v1/data/app/payment-gateway/production
allow=false, violations=[...]
DENY with violation details
CI/CD Pipeline
ComplianceService
Policy Resolver
OPA Engine


CI/CD Pipeline
ComplianceService
Policy Resolver
OPA Engine
POST /api/compliance/evaluate
Load Application and EnvironmentConfig
Resolve policy for app=user-service, vertical=null, env=production
Check app.user-service.production
Package not found
Check vertical (none assigned)
No vertical assigned so skip tier 2
Check global.production
Package found
Falls through to global default.
No per application config needed.
Use global.production
POST /v1/data/global/production
allow=true, violations=[]
ALLOW
CI/CD Pipeline
ComplianceService
Policy Resolver
OPA Engine

8.5 Policy Inheritance via Rego Import

Application specific and vertical policies do not duplicate global rules. Instead they **import and extend** the global baseline using Rego's module import system. This means a vertical policy like banking can enforce all of the standard production rules and then add PCI DSS specific checks on top. An application specific policy can further extend a vertical policy with custom CVE blocklists or exception waivers.

Global baseline (`global/production.rego`):

```
package global.production

import data.global.common

default allow := false

allow if {
    common.total_critical(input.scanResults) == 0
    common.total_high(input.scanResults) == 0
    common.tools_executed(input.scanResults, {"snyk", "prisma-cloud"})
    no_high_severity_license_violations
}

violations[msg] if {
    common.total_critical(input.scanResults) > 0
    msg := sprintf("Found %d critical vulnerabilities",
[common.total_critical(input.scanResults)])
}
```

Vertical override (`vertical/banking/production.rego`):

```
package vertical.banking.production

import data.global.production as base
import data.global.common

default allow := false

# Banking requires ALL base production rules PLUS PCI DSS checks
allow if {
    base.allow
    pci_dss_compliant
    no_unencrypted_storage_dependencies
}

pci_dss_compliant if {
    pci_cves := [v |
        some scan in input.scanResults
        some v in scan.vulnerabilities
        startswith(v.cveId, "CVE-PCI")
    ]
```

```
    count(pci_cves) == 0
}

violations[msg] if {
    some msg in base.violations
}

violations[msg] if {
    not pci_dss_compliant
    msg := "PCI DSS compliance failure: blocked CVEs found in PCI scope"
}
```

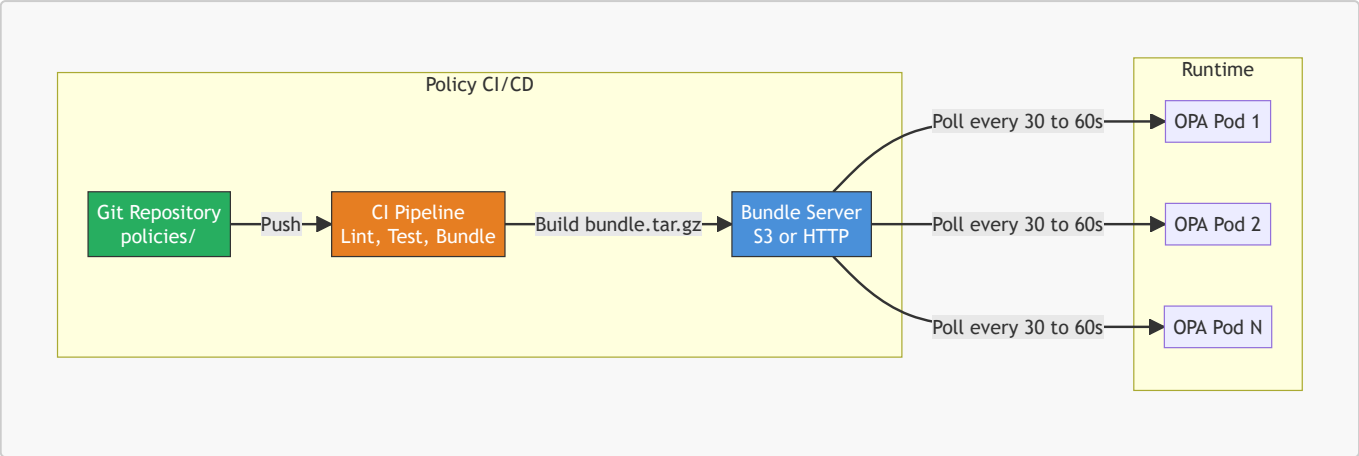
Application specific override (app/payment-gateway/production.rego):

```
package app.payment_gateway.production

import data.vertical.banking.production as banking
import data.global.common

default allow := false

allow if {
    banking.allow
    no_blocked_cves
}

blocked_cve_list := {"CVE-2024-1234", "CVE-2024-5678", "CVE-2025-0001"}

no_blocked_cves if {
    found := [v |
        some scan in input.scanResults
        some v in scan.vulnerabilities
        v.cveId in blocked_cve_list
    ]
    count(found) == 0
}

violations[msg] if {
    some msg in banking.violations
}

violations[msg] if {
    not no_blocked_cves
    msg := "Blocked CVE detected on payment gateway explicit blocklist"
}
```

8.6 Distribution Model at Scale

For local development policies are loaded via volume mount. For staging and production environments the volume mount approach should be replaced with **OPA Bundle distribution** which enables hot reloading,

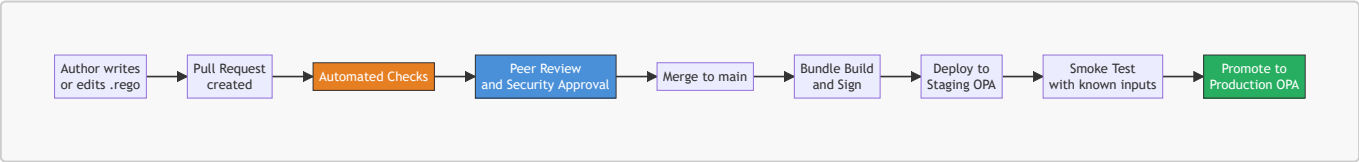
versioning, and cryptographic signing of policy artifacts.



Method	Use Case	Hot Reload	Versioned	Signed
Volume mount	Local development, Docker Compose	No (restart required)	No	No
OPA Bundles	Staging, Production, Kubernetes	Yes (polling interval)	Yes	Yes
OPA Management API	Emergency policy push	Yes (immediate)	No	No

8.7 Policy Lifecycle and CI/CD for Policies

Every policy change follows a governed pipeline before reaching OPA in any environment. This ensures that all rules are syntactically valid, tested against known inputs, and reviewed by both a peer engineer and a security team member before promotion.



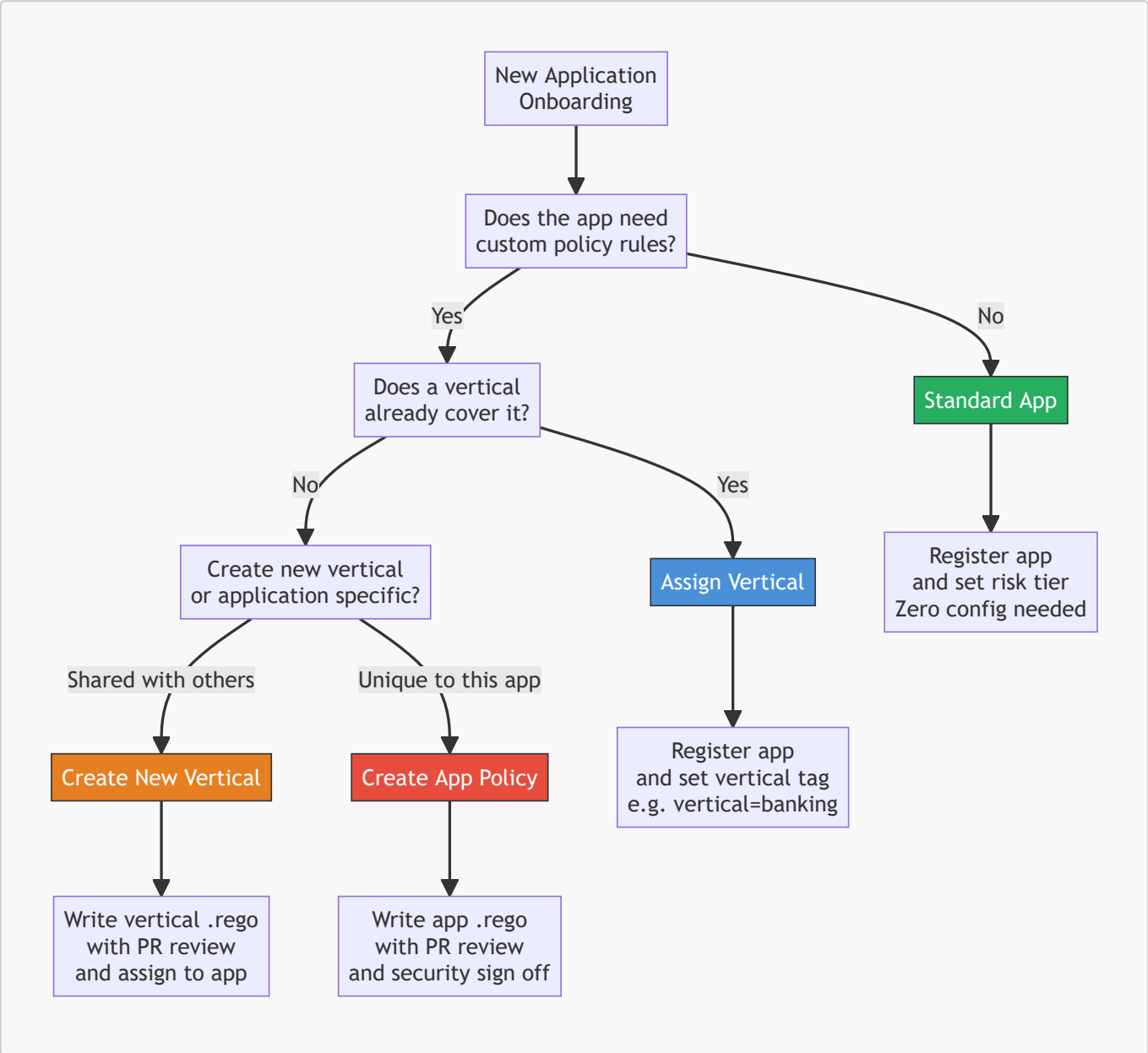
Automated checks in the policy CI pipeline:

Check	Tool	Purpose
Syntax validation	`opa check`	Ensures all Rego files parse correctly and are free of syntax errors
Unit tests	`opa test`	Runs test cases for each policy package against known input and expected output pairs
Coverage report	`opa test --coverage`	Validates that a minimum of 90 percent of policy rules are exercised by test cases
Dependency analysis	Custom script	Verifies all Rego import statements resolve to valid packages in the bundle

Check	Tool	Purpose
Breaking change detection	Diff analysis	Flags changes to allow or violations rule signatures that could affect consumers
Bundle build	`opa build`	Produces a signed and versioned bundle artifact ready for distribution

8.8 Application Onboarding Model

With hierarchical resolution in place, onboarding a new application becomes straightforward. The vast majority of applications will use the global default policy with zero additional configuration.



Onboarding Path	Effort	Frequency	Policy Files Changed
Standard app (global policy)	5 minutes	~85% of apps	None
Assign existing vertical	10 minutes	~10% of apps	None
Create new vertical	1 to 2 days	~4% of apps	1 to 2 new Rego files

Onboarding Path	Effort	Frequency	Policy Files Changed
Application specific override	1 to 2 days	~1% of apps	1 new Rego file

8.9 Scale Projections

Metric	10 Apps	100 Apps	500 Apps
Total Rego files	~4	8 to 12	15 to 25
Global policies	4	4	4
Vertical policies	0	2 to 4 verticals (4 to 8 files)	5 to 10 verticals (10 to 20 files)
Application specific policies	0	1 to 3	3 to 8
Apps needing zero policy config	10	~85	~425
Policy bundle size	Under 50 KB	Under 100 KB	Under 200 KB
OPA evaluation latency impact	None	Negligible	Under 5ms increase

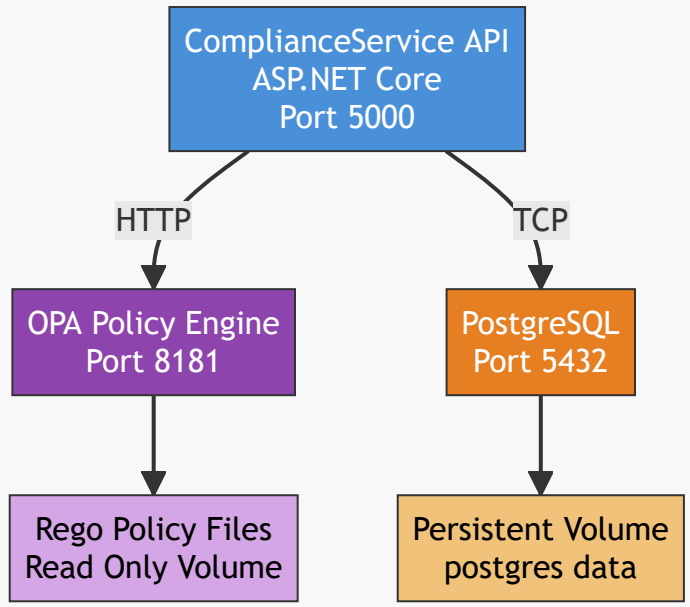
The hierarchy keeps the policy repository manageable regardless of how many applications are onboarded. Adding a new standard application requires **zero policy changes** and only a single API call to register it.

---

9. Infrastructure and Deployment Architecture

9.1 Container Topology

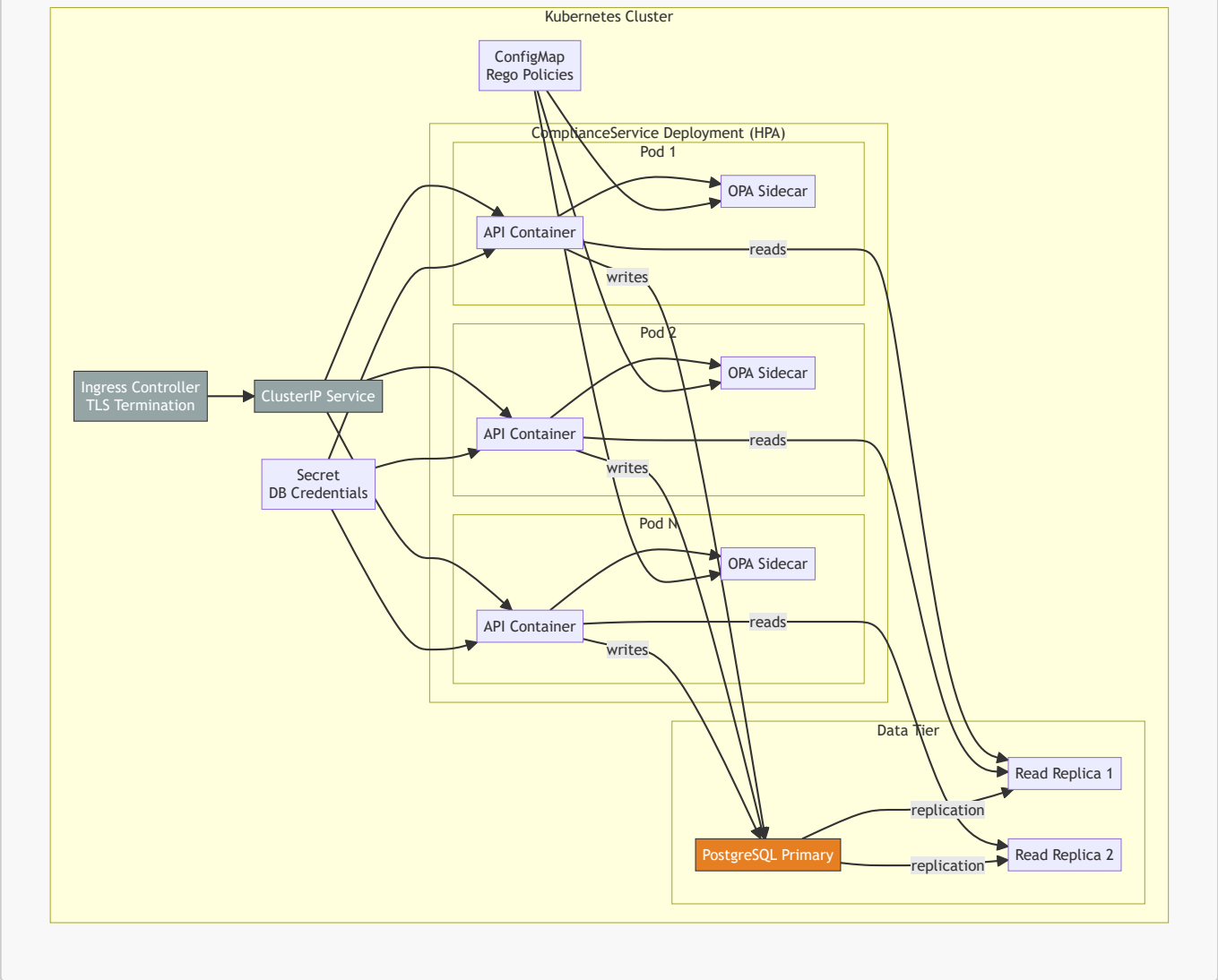
The initial deployment topology consists of three containers running on a shared Docker network. The API container hosts the ASP.NET Core application. The OPA container runs the policy engine with Rego files mounted as a read only volume. The PostgreSQL container provides persistent storage with a named volume for data durability.



Container	Image	Ports	Volumes	Health Check
`compliance-api`	.NET 8.0 build	`5000:80`	none	HTTP GET /health
`compliance-opa`	`openpolicyagent/opa`	`8181:8181`	`./policies` (read only)	HTTP GET /health
`compliance-postgres`	`postgres:16-alpine`	`5432:5432`	`postgres-data` (named volume)	pg_isready

9.2 Production Deployment Target (Kubernetes)

The production topology deploys the API and OPA as co located containers within the same Kubernetes pod. This ensures policy evaluation remains a localhost call with no network overhead. The deployment is backed by a Horizontal Pod Autoscaler that scales based on CPU utilization and request rate. The data tier uses a PostgreSQL primary with read replicas to separate write traffic from query traffic.



9.3 Database Schema Management

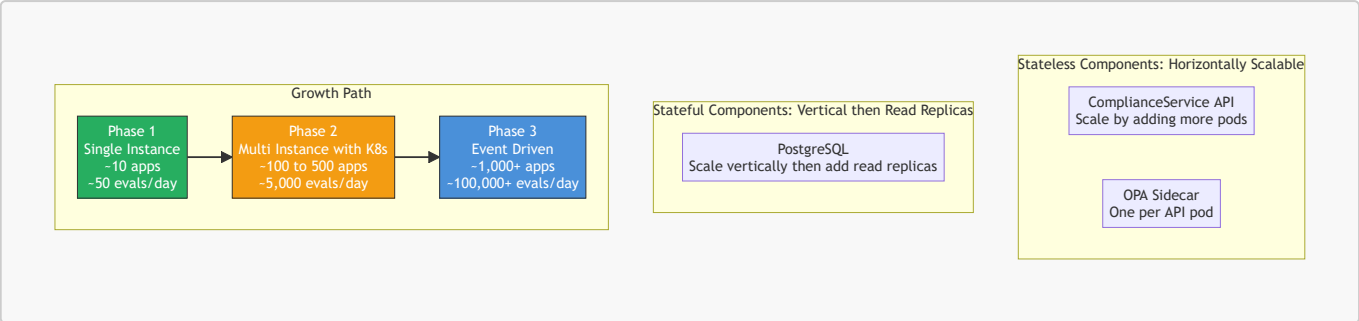
Aspect	Strategy
Migration tool	Entity Framework Core migrations
Development	Auto migrate on startup for rapid iteration
Staging	Migration scripts applied during the deployment pipeline
Production	Explicit versioned SQL scripts only; auto migration is never enabled
Rollback	Down migrations are maintained and tested in staging before any production release
Connection resilience	Retry on transient failure with configurable retry count and backoff delay

10. Scalability and Growth Strategy

10.1 Scalability Assessment

The API and OPA sidecar are both stateless which means they scale horizontally by adding more pods. PostgreSQL is the stateful component and scales vertically initially then horizontally via read replicas to

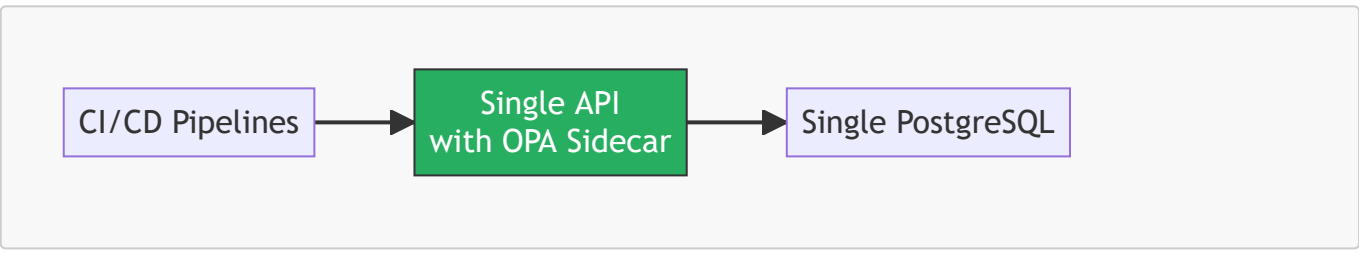
offload audit and reporting queries from the write path.



10.2 Scaling Strategies by Phase

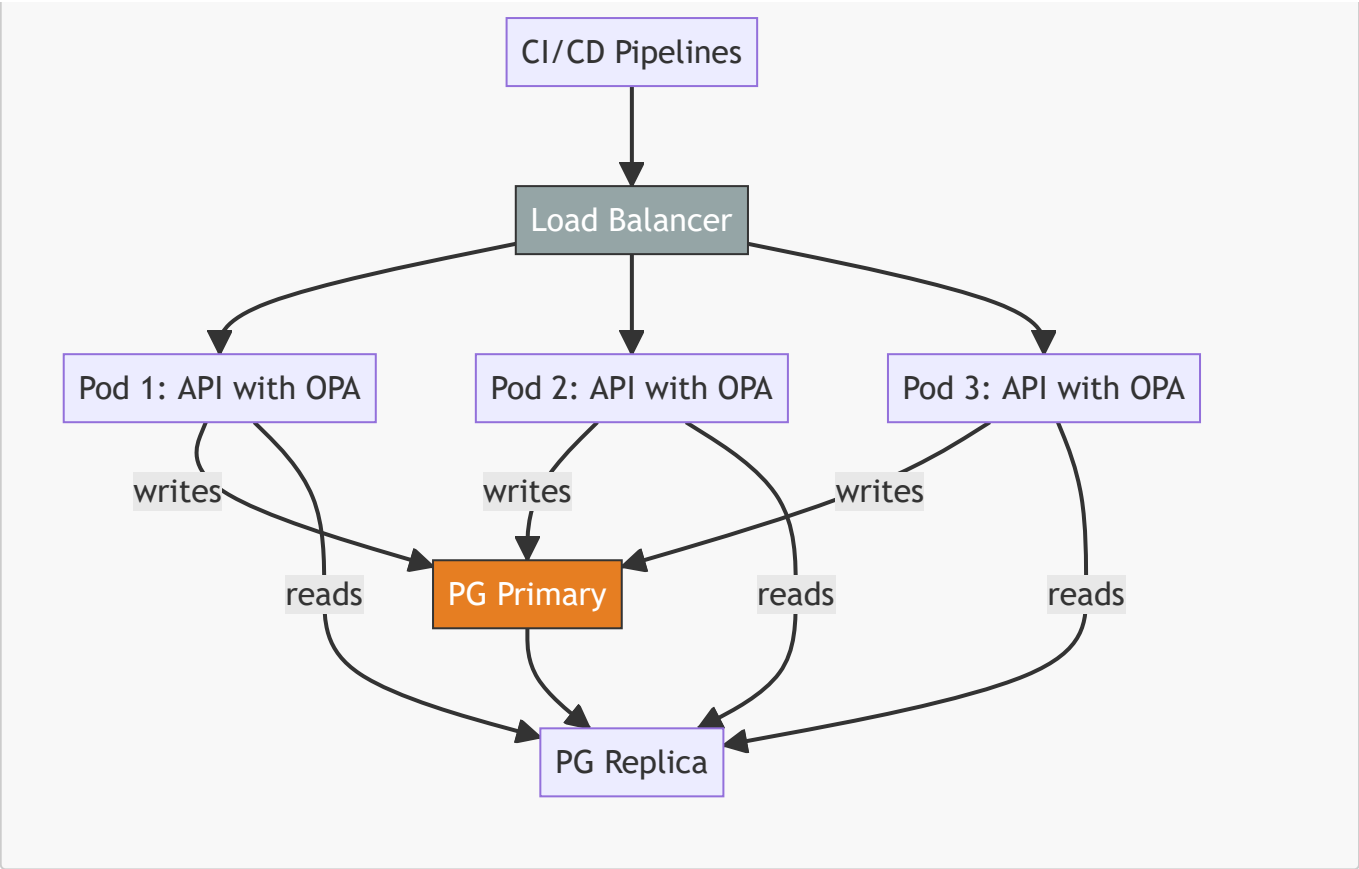
Phase 1: Single Instance

The initial deployment uses Docker Compose with a single API instance, OPA sidecar, and PostgreSQL database. This is suitable for the initial rollout with a small number of teams and provides the fastest path to having the service operational in a non production environment.



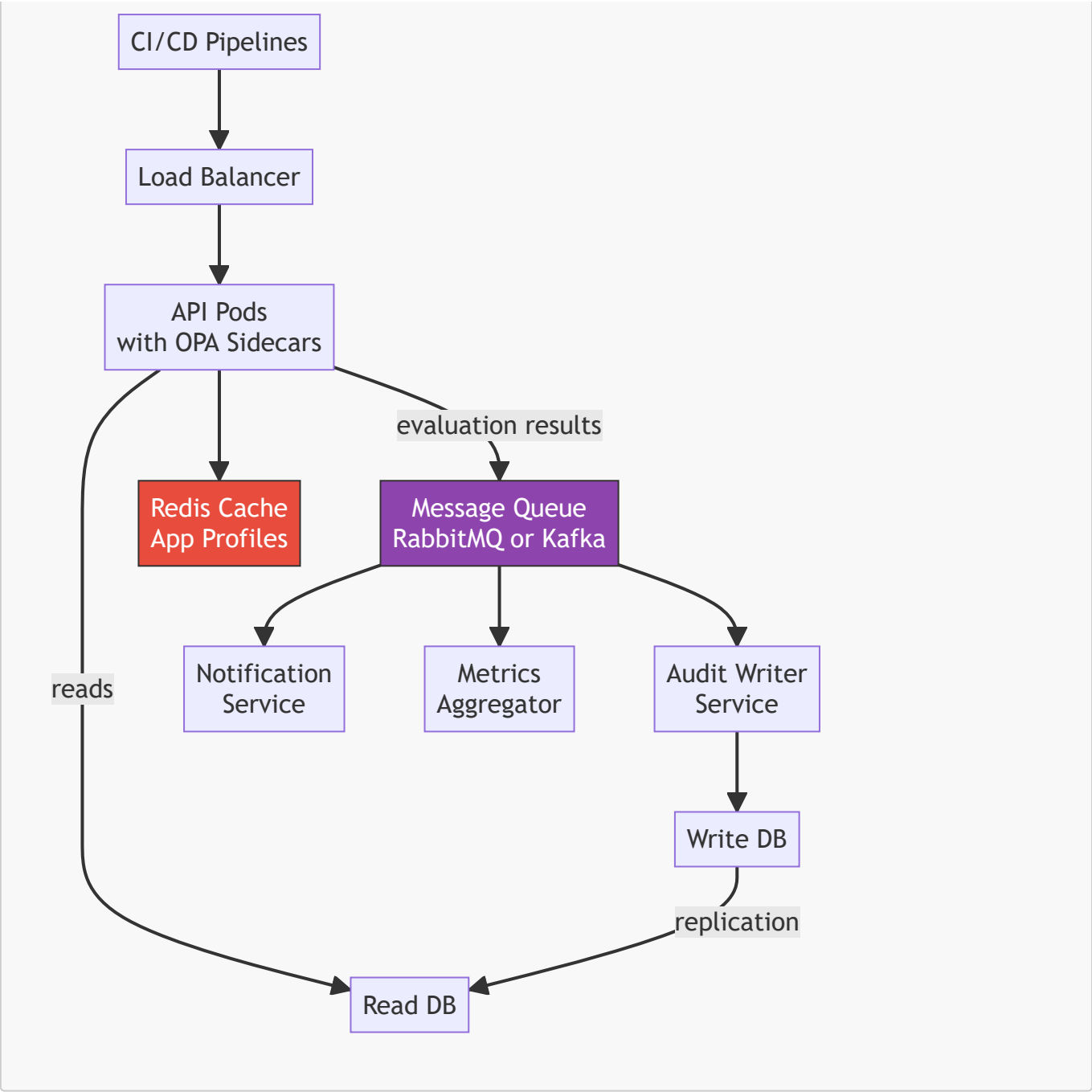
Phase 2: Kubernetes Multi Instance

As adoption grows the service moves to Kubernetes with multiple replicas behind a load balanced ClusterIP service. A Horizontal Pod Autoscaler adjusts the replica count based on CPU utilization and inbound request rate. PostgreSQL adds a read replica to offload audit queries. OPA policies are distributed via ConfigMap or an OPA Bundle Server for centralized management.



Phase 3: Event Driven Architecture

At enterprise scale the architecture evolves to decouple write heavy operations from the synchronous evaluation path. Audit log writes and notification dispatch move to asynchronous consumers reading from a message queue. A Redis cache layer reduces database load for application profile lookups. The data tier splits into dedicated write and read stores with full CQRS separation.



10.3 Performance Optimization Path

Optimization	Impact	Effort	Phase
Redis cache for application profiles	Reduces database read volume by approximately 80 percent	Low	2
Database read replicas	Offloads audit and reporting queries from the primary	Medium	2
Async audit log writes	Reduces synchronous evaluation latency by deferring persistence	Medium	2
OPA bundle server	Centralizes policy distribution with versioning and signing	Medium	2
Message queue for notifications	Decouples notification delivery from the evaluation path	Medium	3

Optimization	Impact	Effort	Phase
Request idempotency	Prevents duplicate evaluations from pipeline retries	Low	2
Database index tuning	Accelerates audit queries across time range and application dimensions	Low	2
Audit log archival to cold storage	Manages unbounded data growth in the primary database	Medium	3

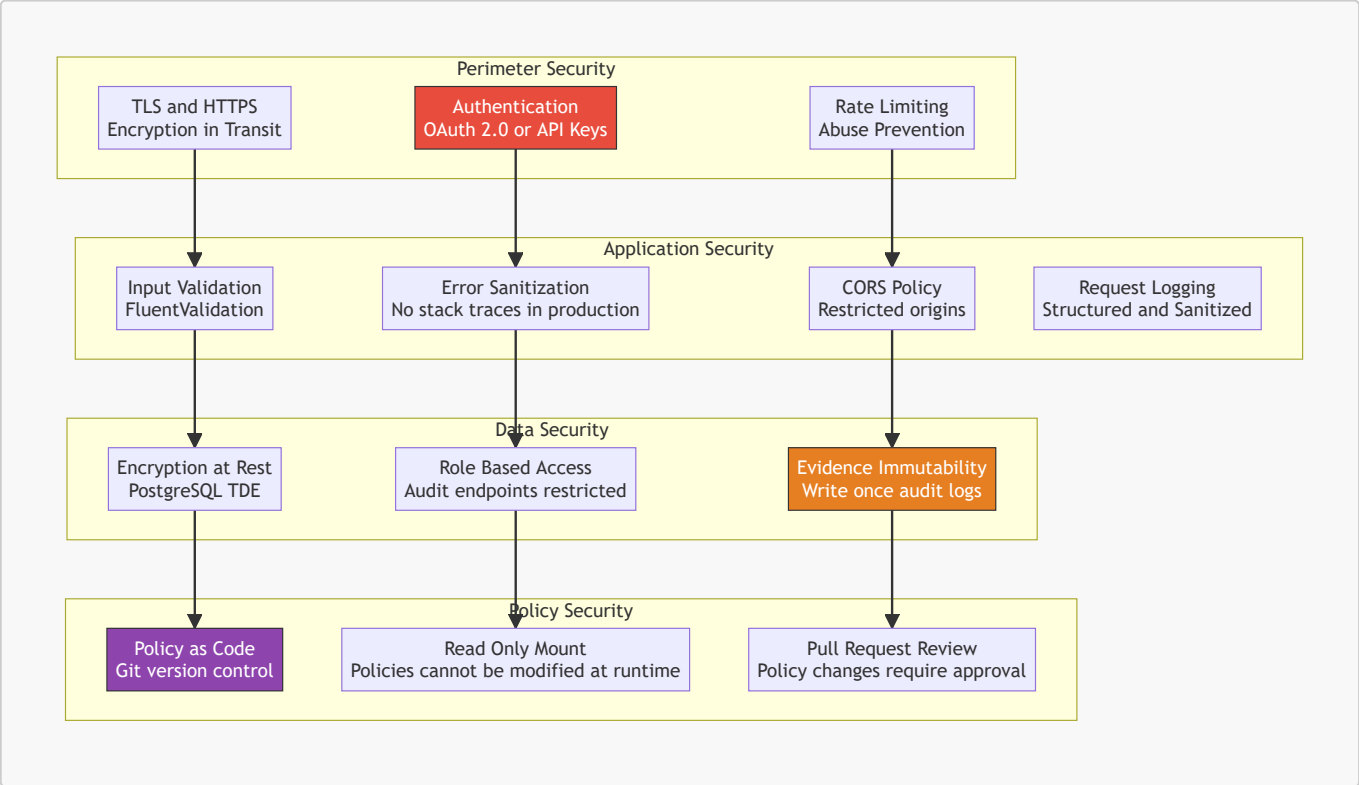
10.4 Growth Targets

Metric	Phase 1	Phase 2	Phase 3
Applications onboarded	5 to 10	50 to 100	500+
Evaluations per day	~50	~2,000	~50,000+
Evaluation latency (p95)	Under 2s	Under 500ms	Under 200ms
Audit query response time	Under 5s	Under 1s	Under 500ms
Availability	99%	99.9%	99.95%

11. Security and Governance

11.1 Security Architecture

The security model is organized into four layers. The perimeter layer handles TLS termination, authentication, and rate limiting. The application layer validates all inbound data and sanitizes error responses. The data layer ensures encryption at rest and enforces role based access controls. The policy layer protects the integrity of the compliance rules themselves through Git based version control and read only runtime mounts.



11.2 Security Controls

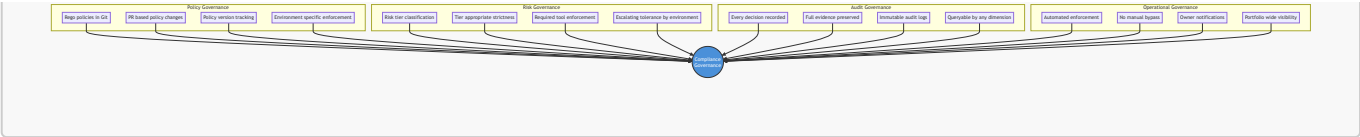
Control	Category
HTTPS enforcement for all communication	Perimeter
Authentication via OAuth 2.0 or API keys	Perimeter
Role based authorization (RBAC)	Perimeter
CORS restriction to known origins	Perimeter
Rate limiting on all public endpoints	Perimeter
Structural and semantic input validation	Application
Error response sanitization in production	Application
Structured request and response logging	Observability
Composite health check probes for PostgreSQL and OPA	Observability
Secrets management through external vault integration	Infrastructure
Transparent data encryption at rest	Data
Read only policy file mounts at runtime	Policy
Immutable full evidence audit trail	Governance
HMAC or JWT signing for scan result integrity	Data Integrity
Cryptographic bundle signing for OPA policies	Policy

11.3 Security Scanning Integration

The service integrates with industry standard security scanning tools through a tool agnostic scan result contract. Any scanner that can produce output conforming to the schema can participate in the compliance evaluation.

Tool	Purpose
Snyk	Software composition analysis covering dependency vulnerability scanning and license compliance verification
Prisma Cloud	Container image security scanning, infrastructure as code analysis, and cloud security posture management
Other Security Tools	Static application security testing, dynamic analysis, or custom internal scanners all extensible via the scan result contract

11.4 Governance Framework

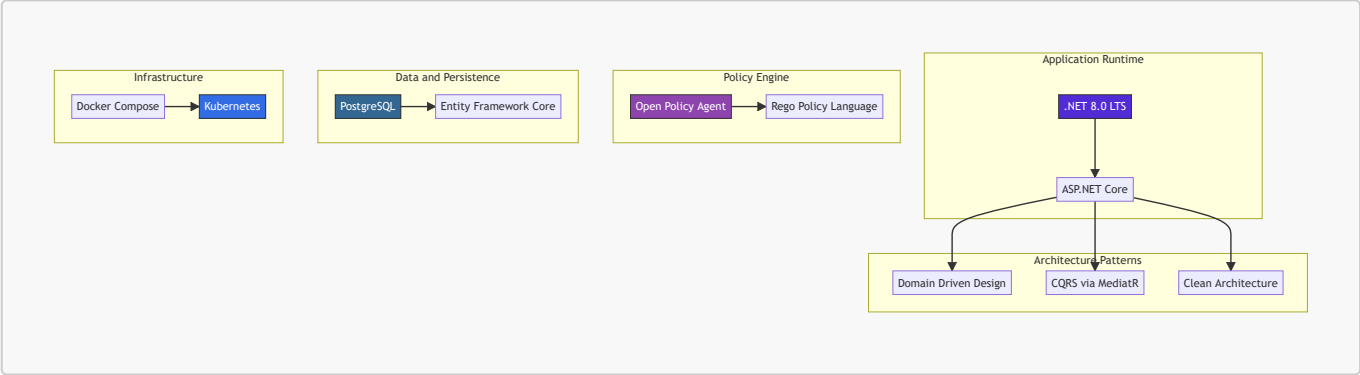


11.5 Compliance Readiness

The audit trail is designed to support evidence generation for the following regulatory frameworks:

Framework	Supported Capability
SOC 2 Type II	Continuous control evidence with timestamped decisions
PCI DSS	Vulnerability management evidence for Requirements 6.1 and 6.2
HIPAA	Access control and audit log requirements
ISO 27001	Information security management controls
FedRAMP	Continuous monitoring and vulnerability tracking

12. Technology Stack Summary



Category	Technology	Purpose
Runtime	.NET 8.0 LTS	Long term support application runtime
Web Framework	ASP.NET Core	REST API hosting and middleware pipeline
Policy Engine	Open Policy Agent	Rego based policy evaluation via sidecar
Database	PostgreSQL	Persistent relational data store for application profiles, evaluations, and audit records
ORM	Entity Framework Core	Database access layer with migration support
Mediator	MediatR	CQRS command and query dispatch with pipeline behaviors
Validation	FluentValidation	Declarative request input validation with strongly typed rules

Category	Technology	Purpose
Logging	Serilog	Structured logging with context enrichment and multiple sink support
Containers	Docker and Kubernetes	Development orchestration and production deployment

> **Next Steps:** Review this proposal with stakeholders and approve for development to begin. The target is to deliver the core service by **March 31, 2026** with the foundational architecture, security controls, policy evaluation engine, and CI/CD pipeline integration operational.

Technical Proposal prepared February 2026 ComplianceService: Policy Gateway for CI/CD Pipeline Compliance Target Delivery: March 31, 2026