INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Policy-as-Code Enforcement in DevSecOps Pipelines

ALICE MIRANDA RESENDE
Junho de 2026

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Policy-as-Code Enforcement in DevSecOps Pipelines

Alice Resende

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Cybersecurity And Systems Administration

Advisor: Luís Nogueira (ISEP)

Porto, June 21, 2026

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, June 21, 2026

Dedicatory

To my family and friends,

who endured countless conversations about my dissertation, celebrated the small victories, tolerated the frustrations, and never stopped believing I would reach this point.

This achievement is as much yours as it is mine.

Abstract

The increasing adoption of DevOps and Continuous Integration and Continuous Delivery (CI/CD) pipelines has significantly accelerated software development and deployment processes. However, this acceleration has also intensified challenges related to security governance and regulatory compliance, particularly in cloud-native and infrastructure-as-code environments. DevSecOps has emerged as a response to these challenges by promoting the integration of security throughout the software delivery lifecycle, yet many security and compliance controls remain manually enforced, inconsistently applied, or introduced too late in the pipeline.

Policy-as-Code (PaC) has gained attention as a promising approach for formalising security and compliance requirements as executable, machine-readable policies that can be automatically enforced. Despite growing industrial adoption, the scientific literature reveals a lack of systematic, lifecycle-oriented frameworks for integrating PaC into DevSecOps pipelines in a consistent, auditable, and scalable manner.

This dissertation investigates how Policy-as-Code can be effectively integrated into DevSecOps environments, with a particular focus on early enforcement within CI/CD pipelines. The work analyses the current state of the art, identifies methodological and architectural gaps, and proposes a structured approach for defining, managing, and enforcing policies across different stages of the pipeline. The proposed approach is evaluated through a proof of concept implemented using controlled scenarios and synthetic Kubernetes workload manifests, considering detection effectiveness, integration feasibility, and performance overhead.

The results aim to contribute to a clearer understanding of Policy-as-Code as a foundational governance mechanism for secure software delivery and to provide practical guidance for its adoption in modern DevSecOps workflows.

Keywords: Policy-as-Code, CI/CD Pipeline Enforcement, DevSecOps Governance, Infrastructure as Code Security, OPA/Rego, Shift-Left Compliance

Resumo

A crescente adoção de práticas DevOps e de pipelines de Integração Contínua e Entrega Contínua (CI/CD) tem acelerado significativamente os processos de desenvolvimento e disponibilização de software. No entanto, esta aceleração tem também intensificado os desafios associados à governação de segurança e à conformidade regulatória, particularmente em ambientes cloud-native e baseados em Infrastructure as Code. O paradigma DevSecOps surge como resposta a estes desafios, ao promover a integração da segurança ao longo de todo o ciclo de vida do software, embora muitos mecanismos de segurança e conformidade continuem a ser aplicados de forma manual, inconsistente ou tardia.

O conceito de Policy-as-Code (PaC) tem vindo a afirmar-se como uma abordagem promissora para formalizar requisitos de segurança e conformidade sob a forma de políticas executáveis e legíveis por máquina, permitindo a sua aplicação automática. Apesar da sua crescente adoção industrial, a literatura científica evidencia a ausência de modelos sistemáticos e orientados ao ciclo de vida que suportem a integração consistente, auditável e escalável de PaC em pipelines DevSecOps.

Esta dissertação analisa a integração de Policy-as-Code em ambientes DevSecOps, com especial enfoque na sua aplicação precoce em pipelines CI/CD. O trabalho inclui uma análise do estado da arte, a identificação de lacunas metodológicas e arquiteturais, e a proposta de uma abordagem estruturada para a definição, gestão e aplicação de políticas ao longo das diferentes fases do pipeline. A abordagem proposta é avaliada através de uma prova de conceito implementada com cenários controlados e manifests Kubernetes sintéticos, considerando critérios como eficácia de deteção, viabilidade de integração e impacto no desempenho.

Os resultados pretendem contribuir para uma melhor compreensão do Policy-as-Code como mecanismo fundamental de governação em processos de entrega segura de software, bem como fornecer orientações práticas para a sua adoção em contextos DevSecOps modernos.

Contents

List of Figures

List of Tables

List of Abbreviations

ABAC	Attribute-Based Access Control
API	Application Programming Interface
CD	Continuous Delivery / Continuous Deployment
CI	Continuous Integration
CI/CD	Continuous Integration and Continuous Delivery / Deployment
CIS	Center for Internet Security
CNCF	Cloud Native Computing Foundation
CRD	Custom Resource Definition
DevOps	Development and Operations
DevSecOps	Development, Security, and Operations
FR	Functional Requirement
GDPR	General Data Protection Regulation
GitOps	Git-based Operations (infrastructure management driven by version-controlled declarative configuration)
IaC	Infrastructure as Code
JSON	JavaScript Object Notation
K8s	Kubernetes
NFR	Non-Functional Requirement
OCI	Open Container Initiative
OPA	Open Policy Agent
PaC	Policy-as-Code
PCI-DSS	Payment Card Industry Data Security Standard
PoC	Proof of Concept
RBAC	Role-Based Access Control
Rego	Rego (the policy language of Open Policy Agent; not an acronym)
RQ	Research Question
SBOM	Software Bill of Materials
SLR	Systematic Literature Review
WBS	Work Breakdown Structure
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

1.1 Context and Motivation

The rapid evolution of software delivery practices has transformed how organisations build, test, and deploy software systems. This evolution has been driven largely by Continuous Integration and Continuous Delivery (CI/CD). DevOps has emerged as a leading paradigm for enhancing cooperation between development and operations teams and speeding up delivery. Integrating security and compliance from the start has become crucial as delivery pipelines grow and infrastructures become more dispersed and intricate. Security is no longer a late-stage or optional issue. This need has led to the emergence of *DevSecOps*, a paradigm that integrates security controls, governance, and risk management throughout the entire software development lifecycle. While DevSecOps promotes early and continuous security verification, organisations still face significant challenges in ensuring that security and compliance requirements are defined, validated, and enforced in a consistent and automated manner. In practice, many governance processes remain manually executed, ad hoc, or weakly integrated with the pipeline.

Within this context, *Policy-as-Code* (PaC) has emerged as a promising approach for formalising security and compliance requirements as executable code. PaC enables the automated validation and enforcement of rules directly within CI/CD pipelines, thereby contributing to continuous compliance with both internal security standards and external regulatory obligations. Although PaC is increasingly adopted in industry, the scientific literature remains fragmented, lacking unified lifecycle models, architectural structures, and methodological guidance for its integration within DevSecOps ecosystems.

This dissertation is situated within this gap. It seeks to provide conceptual and methodological clarity on how PaC can be systematically operationalised across modern secure software delivery workflows.

1.2 Problem Description

Despite the benefits of DevSecOps, current practices reveal several challenges that motivate this research. Security and compliance activities are frequently implemented in a tool-centric or reactive manner, without a unifying architectural perspective or defined lifecycle. PaC tools, such as Open Policy Agent (OPA), Rego, Kyverno, or Conftest, enable powerful enforcement capabilities, yet organisations often lack consistent approaches for modelling, testing, validating, deploying, and auditing policies across the pipeline. This results in inconsistent enforcement, limited traceability, and misalignment with regulatory requirements such as ISO 27001, GDPR, or PCI-DSS.

The academic literature further reveals a scarcity of structured models that integrate PaC into DevSecOps workflows. Existing studies tend to focus on isolated tools, specific use-cases, or narrow enforcement mechanisms, rather than addressing broader governance, life-cycle management, or empirical validation. Consequently, organisations adopting DevSec-Ops at scale face operational risks, compliance gaps, and architectural inconsistencies due to the absence of cohesive and well-defined PaC integration strategies.

The central research problem addressed in this dissertation is therefore defined as follows:

> *There is a lack of structured and empirically evaluated approaches for integrating Policy-as-Code enforcement into DevSecOps pipelines.*

This problem affects a wide range of stakeholders, including software development teams, DevOps and platform engineers, security architects, compliance officers, auditors, and or-ganisations operating under strict governance or regulatory constraints.

1.3 Objectives

This dissertation aims to analyse, structure, and synthesise the integration of Policy-as-Code into DevSecOps environments. The main objectives are:

- **O1:** Analyse the current state of the art regarding DevSecOps practices, security automation, and Policy-as-Code mechanisms.
- **O2:** Identify gaps, limitations, and challenges in existing frameworks, tools, and aca-demic research related to PaC in CI/CD pipelines.
- **O3:** Define a research methodology that aligns with academic standards, ensuring scientific rigour and ethical compliance.
- **O4:** Propose a structured conceptual framework or lifecycle model for the integration of Policy-as-Code in DevSecOps.
- **O5:** Validate and analyse the proposed framework through analytical, comparative, or scenario-based evaluation.

These objectives address gaps identified in the literature and contribute to improving the methodological and architectural understanding of PaC within secure software delivery work-flows.

1.4 Scope and Practical Delimitation

Although the broader problem of Policy-as-Code integration in DevSecOps spans many tools, artefact types, and pipeline stages, the practical scope of this dissertation is deliberately constrained to keep the evaluation focused and reproducible.

The implementation targets two enforcement points within the CI/CD pipeline: CI-level validation using Conftest and Rego, and deployment-time admission control using OPA Gatekeeper. These two points were chosen because they represent the earliest and the most authoritative enforcement boundaries in a Kubernetes-oriented delivery workflow, respec-tively.

The artefact scope is restricted to Kubernetes Deployment manifests. This reflects the most common workload type in cloud-native environments and is directly amenable to declarative

Policy-as-Code validation. Other artefact types — such as Helm charts, Terraform plans, or container image metadata — are outside the practical scope, though the conceptual approach discussed in Chapter 2 applies more broadly.

The policy set is intentionally small: five security and governance rules were selected based on their relevance to Kubernetes workload security, their suitability for declarative Rego implementation, and their compatibility with both enforcement stages. This subset does not attempt to cover all Kubernetes security domains but is sufficient to demonstrate multi-stage enforcement behaviour.

The following topics are explicitly outside the scope of the practical work:

- large-scale or production-grade policy ecosystems;
- runtime behavioural monitoring after deployment;
- distributed or multi-cluster enforcement architectures;
- formal compliance mapping to regulatory standards such as ISO 27001 or GDPR;
- organisational governance processes or policy lifecycle management at enterprise scale.

Where relevant, these topics are discussed conceptually in the literature review but are not evaluated experimentally.

1.5 Work Planning

The work is divided into several key phases, including the initial exploration of the problem space, the execution of a systematic literature review, the definition of the research methodology, the development of the conceptual framework, the analytical validation of the proposed approach, and the production of the dissertation manuscript. Each phase is associated with specific deliverables and milestones, which are monitored throughout the project to maintain alignment with objectives and ensure feasibility. To support this planning, a Work Breakdown Structure (WBS) and a Gantt chart were developed.

1.5.1 Gantt Chart

The Gantt chart visually represents the temporal distribution of tasks, highlighting their duration, sequencing, and interdependencies. This timeline ensures transparency in project execution, facilitates monitoring, and provides a clear roadmap toward the completion of the dissertation.

1.5.2 Work Breakdown Structure

In addition to the project schedule, a Work Breakdown Structure (WBS) was defined to decompose the dissertation work into manageable and logically structured tasks. The WBS provides a hierarchical view of the project activities, aligning research phases, development tasks, and evaluation activities with their corresponding deliverables. This decomposition supports scope control, progress tracking, and risk mitigation throughout the project lifecycle.

Figure 1.1 presents the WBS adopted for this dissertation. It captures the relationship between project management activities, research foundations, Policy-as-Code development,

Table 1.1: Project Schedule (Gantt Chart)

Task	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May	Jun
Problem Definition and Context	X								
Project Management and Planing	X	X							
Research Methodology		X							
State of the Art		X	X	X					
Codebase Analysis					X	X			
Functionality Implementation						X	X	X	
Integration and Functional Testing							X	X	X
Results & Conclusion									X
Writing the Dissertation		X	X	X	X	X	X	X	X

evaluation, and dissertation writing, ensuring traceability between objectives, tasks, and outcomes.

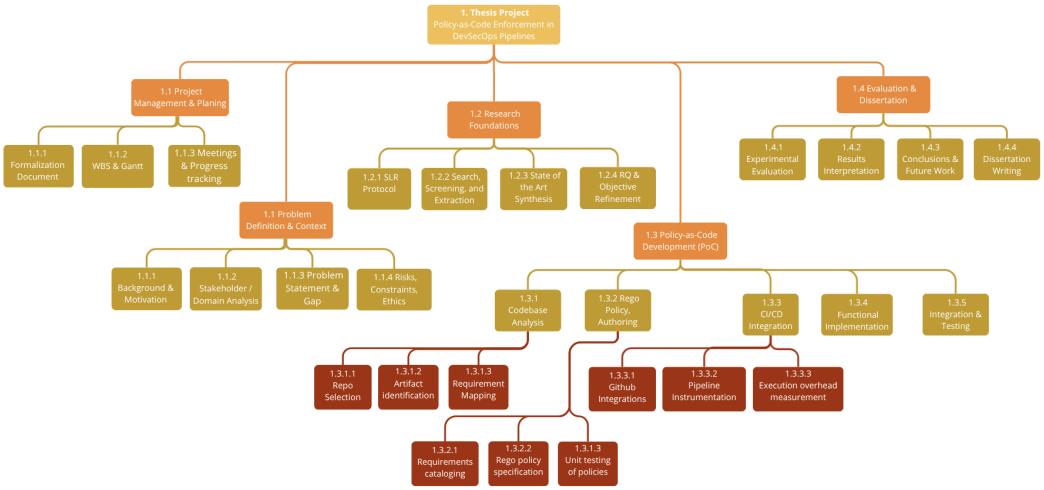

Figure 1.1: Work Breakdown Structure for the Policy-as-Code Enforcement in DevSecOps Pipelines project

1.6 Contributions of This Dissertation

This dissertation makes three concrete and distinct contributions to the body of knowledge on Policy-as-Code in DevSecOps. Each contribution answers a different dimension of the question: *what is original in this work?*

C1 — Literature contribution: Structured synthesis of PaC integration challenges. A systematic literature review of 27 peer-reviewed studies (2019–2025) identifies and categorises five persistent gaps in the adoption of Policy-as-Code within DevSecOps pipelines.

Unlike prior surveys that enumerate tools or use cases, this synthesis maps gaps explicitly to pipeline enforcement stages, producing a structured and stage-oriented view of where the field falls short. This framing directly motivates the architectural decisions in C2 and the evaluation design in C3.

C2 — Architectural contribution: A multi-stage enforcement architecture using a unified policy language. This dissertation proposes and implements an architecture in which the same Rego policy logic enforces governance at two structurally different enforcement boundaries: CI validation time (via Conftest) and Kubernetes admission time (via OPA Gatekeeper). The originality lies not in the individual tools, which are mature and well-documented, but in the demonstration that a single policy definition can span both enforcement stages without duplication. Existing approaches that use separate tools and languages for CI and admission (for example, Checkov at CI and Kyverno at admission) require maintaining parallel policy representations; this architecture eliminates that duplication. Few studies explicitly evaluate policy reuse across both enforcement stages.

C3 — Experimental contribution: Reproducible empirical baseline for Conftest Gatekeeper enforcement. The evaluation provides quantitative evidence — 261 policy assertions across 29 Kubernetes manifests, with zero false positives and zero false negatives — that the proposed architecture correctly enforces all five policies at the CI stage, and that the Gatekeeper admission layer independently rejects non-compliant workloads even when CI is bypassed. Execution overhead for Conftest validation (56.3 ms) is statistically indistinguishable from the kubectl baseline (56.8 ms), confirming that the policy evaluation cost is operationally negligible. This constitutes the first controlled, reproducible empirical baseline for this specific two-stage Conftest–Gatekeeper configuration reported in the reviewed literature.

The dissertation does not claim to define a new policy language, to provide a production-ready governance platform, or to solve the policy lifecycle management problem. The contributions are bounded by the proof-of-concept scope defined in Section 1.4.

1.7 Document Structure

This dissertation is organised into nine chapters:

- **Chapter 1 — Introduction** presents the context, problem statement, motivation, objectives, scope, and planning of the dissertation, including the Work Breakdown Structure and project schedule.
- **Chapter 2 — Context and Related Work** provides a systematic literature review of DevSecOps, Policy-as-Code, CI/CD security automation, and cloud-native governance. It details the review methodology, inclusion and exclusion criteria, search strategy, study selection, synthesis of findings, and identified research gaps.
- **Chapter 3 — Research Design and Approach** describes the research methodology, the proposed Policy-as-Code enforcement approach, the evaluation strategy, and the validity and limitations of the study.
- **Chapter 4 — Ethical Considerations** addresses ethical, legal, and data protection aspects relevant to the research, including compliance with ethical research principles and responsible use of automation technologies.

- **Chapter 5 — Project Planning and Feasibility** presents the project scope, constraints, risk analysis, schedule, milestones, and monitoring and control mechanisms.
- **Chapter 6 — Solution Design and Architecture** defines the requirements, architecture, policy model, and enforcement layers of the proposed multi-stage Policy-as-Code approach, along with its limitations and scope.
- **Chapter 7 — Proof of Concept Implementation** describes the development environment, repository structure, Rego policy implementations, CI workflow configuration, Gatekeeper setup, evaluation dataset, and scenario design.
- **Chapter 8 — Evaluation and Results** presents the experimental evaluation methodology and results for the three research questions, including detection capability, governance coverage, CI bypass mitigation, and execution overhead analysis.
- **Chapter 9 — Conclusion and Future Work** summarises the findings, revisits the research questions, acknowledges the limitations, and proposes directions for future research.

Chapter 2

Context and Related Work

2.1 Literature Review

Integration of security and compliance practices into software delivery pipelines has become a central concern in the evolution from DevOps to DevSecOps. This paradigm aims to embed security controls and policy enforcement throughout the software development lifecycle, ensuring that governance, risk, and compliance requirements are verified in a continuous and systematic manner.

Within this context, *Policy-as-Code* (PaC) has emerged as a promising approach to the automation of security and compliance management. PaC enables policies to be specified, versioned, and executed as code, thereby promoting consistent, repeatable, and scalable compliance verification within Continuous Integration and Continuous Delivery (CI/CD) environments. Despite its growing adoption in industry, academic treatment of PaC remains fragmented. Recent work proposes *Security-as-Code* information models with automated policy checks and evidence capture in CI/CD pipelines, while contemporary surveys consolidate the PaC toolchain (e.g., OPA/Rego, Sentinel) and identify typical enforcement points across the pipeline.

To investigate this space in a rigorous manner, a Systematic Literature Review (SLR) was conducted, inspired by the guidelines of Kitchenham and Charters (2007) and Petersen et al. (2015). The review identifies, analyses, and synthesises approaches, tools, and frameworks for PaC enforcement in DevSecOps settings, highlighting both recent technical advances and persistent research challenges.

The review was organised around four thematic areas that reflect the research problem and the dissertation's objectives: (1) approaches and frameworks proposed for integrating Policy-as-Code into DevSecOps pipelines; (2) tools, technologies, and policy languages most commonly used for PaC implementation, with particular attention to Open Policy Agent, Rego, Conftest, and Kyverno; (3) challenges and limitations reported in applying Policy-as-Code within CI/CD pipelines; and (4) open research gaps and opportunities regarding automation, scalability, and governance of PaC in DevSecOps. These four themes structure the synthesis in Sections 2.2 through 2.8 and inform the gap analysis in Section 2.9.

2.1.1 Inclusion and Exclusion Criteria

To ensure methodological rigour and relevance, explicit inclusion and exclusion criteria were defined to guide the selection of primary studies.

Table 2.1: Inclusion and Exclusion Criteria

Inclusion Criteria	Exclusion Criteria
Peer-reviewed journal articles and conference papers (2019–2025)	Non-peer-reviewed material, white papers, or marketing content lacking technical detail
Publications written in English	Duplicated records across databases
Topics explicitly addressing Policy-as-Code, DevSecOps, or CI/CD security automation	Studies with no direct relation to software delivery pipelines, governance, or automation
Frameworks, models, tools, or conceptual approaches to policy/compliance automation	DevOps studies without a security and/or compliance component

2.1.2 Information Sources

The following digital libraries and indexing services were consulted: **ACM Digital Library**, **IEEE Xplore**, **SpringerLink**, **Scopus/Web of Science**, and **Google Scholar**. The SCImago Journal Rank (SJR; area 1700 – Computer Science) was used to verify venue relevance and scientific quality. Records exported from ACM and IEEE constituted the core corpus and were complemented by targeted searches (backward and forward snowballing) and cross-checks in Scopus and Google Scholar.

2.1.3 Search Terms

The search strategy combined targeted keyword groups to capture research at the intersection of *DevSecOps*, *Policy-as-Code*, and *CI/CD automation*. Multiple combinations were executed for each database, with searches typically restricted to titles, abstracts, and keywords, and refined iteratively.

- **DevSecOps context:** "DevSecOps", "Secure DevOps", "Security automation", "CI/CD security";
- **Policy-as-Code:** "Policy as Code", "Open Policy Agent", "OPA", "Rego", "Kyverno", "Compliance as Code";
- **Pipeline integration:** "Continuous Integration", "Continuous Delivery", "Pipeline enforcement", "Governance", "Automation".

2.1.4 Selection Process

Following an SLR-inspired process, the study selection proceeded in three main stages: (1) identification of candidate records; (2) screening of titles and abstracts to remove duplicates and out-of-scope items; and (3) full-text assessment against the inclusion and exclusion criteria. Conceptual relevance and thematic diversity were prioritised to ensure coverage of (a) DevSecOps foundations, (b) PaC/Security-as-Code enforcement, (c) Kubernetes and Infrastructure-as-Code policy enforcement, (d) compliance evidence and identity management, (e) AI-assisted policy validation, and (f) domain-specific applications (e.g., smart contracts).

Selection Table. The final set comprises all the provided papers that satisfied the above criteria. Table 2.2 lists each study with *Title*, *Authors*, and *Year*. For improved readability across pages, the table is implemented using the `longtable` environment.

Table 2.2: Selection of Included State-of-the-Art Studies (Title, Authors, Year)

Title	Authors	Year
Security as Culture: A Systematic Literature Review of DevSecOps	Mary Sánchez-Gordón; Ricardo Colomo-Palacios	2020
Quality-Aware DevOps Research: Where Do We Stand?	Ahmad Alnafessah; Alim Ul Gias; Runan Wang; Lulai Zhu; Giuliano Casale; Antonio Filieri	2021
Toward successful DevSecOps in software development organizations: A decision-making framework	Muhammad Azeem Akbar; Kari Smolander; Sajjad Mahmood; Ahmed Alsanad	2022
A DevSecOps-enabled Framework for Risk Management of Critical Infrastructures	Xhesika Ramaj	2022
SoK: Security of Microservice Applications: A Practitioners' Perspective on Challenges and Best Practices	Priyanka Billawa; Anusha Bambhore Tukaram; Nicolas E. Diaz Ferreyra; Jan Philipp Steghöfer; Riccardo Scandariato; Georg Simhandl	2022
Experimental Evaluation of Rule-Based Autonomic Computing Management Framework for Cloud-Native Applications	Joanna Kosinska; Krzysztof Zielinski	2023
Embracing IaC Through the DevSecOps Philosophy: Concepts, Challenges, and a Reference Framework	Juncal Alonso; Radoslaw Piliszek; Matija Cankar	2023
Industrial Challenges in Secure Continuous Development	Fabiola Moyón; Florian Angermeir; Daniel Mendez	2024
On DevSecOps and Risk Management in Critical Infrastructures: Practitioners' Insights on Needs and Goals	Xhesika Ramaj; Mary Sánchez-Gordón; Vasileios Gkioulos; Ricardo Colomo-Palacios	2024
Towards LowDevSecOps Framework for Low-Code Development: Integrating Process-Oriented Recommendations for Security Risk Management	Gayane Sedrakyan; Maria Eugenia Iacob; Jos Hillegersberg	2024
Supporting continuous vulnerability compliance through automated identity provisioning	Diego Gama; Andrey Brito; André Martin; Christof Fetzer	2024

(continued on next page)

Title	Authors	Year
Vulnerability detection tool in source code by building and leveraging semantic code graph.	Sabine Delaitre; José Maria Pulgar Gutiérrez	2024
Investigating Effectiveness and Compliance to DevOps Policies and Practices for Managing Productivity and Quality Variability	Dan Port; Bill Taber; Parisa Emkani	2024
Automating Cybersecurity Compliance in DevSecOps with Open Information Model for Security as Code	Henry Haverinen; Tomi Janhunen; Tero Päivärinta; Sami Lempinen; Suvi Kaartinen; Sami Merilä	2024
AI-Augmented DevSecOps Pipelines for Secure and Scalable Service-Oriented Architectures in Cloud-Native Systems	Akshay Mittal	2025
POSTER: Policy-driven security-aware scheduling in Kubernetes	Matthew Rossi; Michele Beretta; Dario Facchinetti; Stefano Paraboschi	2025
Model Driven Engineering, Artificial Intelligence, and DevOps for Software and Systems Engineering: A Systematic Mapping Study of Synergies and Challenges	Luca Berardinelli; Vittoriano Muttillo; Romina Eramo; Hugo Bruneliere; Abbas Rahimi; Antonio Cicchetti; Joan Giner-Miguelez; Abel Gómez; Pasqualina Potena; Mehrdad Saadatmand	2025
An evaluation of commonly used Kubernetes security scanning tools	Ioanna Angeliki Kapetanidou; Alexandros Nizamis; Konstantinos Votis	2025
Research on Policy-as-Code for Implementation of Role-based and Attribute-based Access Control	Oleksandr Vakhula; Ivan Opirskyy; Pavlo Vorobets; Orest Bobko; Oleh Kulinich	2025
Policy as Code: A paradigm shifts in infrastructure security and governance	Sarathe Krisshnan Jutoo Vijayaraghavan	2025
AI-driven Security as Code for software development using multi-agent systems	Oleksandr Vakhula; Ivan Opirskyy	2025
DevSecOps practices and tools	Luís Prates; Rúben Pereira	2025
ARPaCCino: An Agentic-RAG for Policy as Code Compliance	Francesco Romeo; Luigi Arena; Francesco Blefari; Francesco Aurelio Pironti; Matteo Lupinacci; Angelo Furfaro	2025

(continued on next page)

Title	Authors	Year
Enhancing Smart Contract Security Through DevSecOps: An Adaptive Approach for Vulnerability Detection	Nghia Dinh; Vinh Truong Hoang; Bay Nguyen Van; Thien Ho Huong; Ha Duong Thi Hong; Hau Nguyen Trung; Kiet Tran Trung	2025
Columbo: A Reasoning Framework for Kubernetes' Configuration Space	Matthijs Jansen; Sacheendra Talluri; Krijn Doekemeijer; Nick Tehrany; Alexandru Iosup; Animesh Trivedi	2025
Extensive Review of Threat Models for DevSecOps	S. Nagasundari; Pratham Manja; Priyansh Mathur; Prasad B. Honnavalli	2025
Practical Integration of Large Language Models into Enterprise CI/CD Pipelines for Security Policy Validation: An Industry-Focused Evaluation	Akshay Mittal; Vivek Venkatesan	2025
Empirical Investigation of Key Enablers for Secure DevOps Practices	Muhammad Azeem Akbar; Abeer Abdulaziz Alsanad	2025

2.2 From DevOps to DevSecOps

DevOps improved software delivery by reducing organisational silos, increasing automation, and shortening feedback loops. However, higher delivery velocity exposed a mismatch with traditional security assurance, which is often manual, late-stage, and detached from development workflows (Akbar, Smolander, et al. 2022).

Empirical studies report that even mature DevOps teams frequently treat security as an external activity, handled by separate teams and tools (Moyón, Angermeir, and Mendez 2024; Sánchez-Gordón and Colomo-Palacios 2020). This separation becomes risky in cloud-native systems, where vulnerabilities and misconfigurations can propagate quickly through CI/CD automation.

The literature also reports recurring operational failure modes, including insecure defaults, configuration drift, dependency vulnerabilities, and identity mismanagement (Billawa et al. 2022; Kosińska and Zieliński 2023; Port, Taber, and Emkani 2024). These issues are amplified by the distribution of security-relevant configuration across Kubernetes manifests, container images, IaC artefacts, and pipeline metadata.

DevSecOps addresses this gap by integrating security validation and governance into the delivery workflow. However, studies consistently identify adoption barriers such as cultural misalignment, fragmented tooling, limited automation, and lack of standardisation (Akbar, Smolander, et al. 2022; Ramaj 2022; Sánchez-Gordón and Colomo-Palacios 2020; Sedrakyan, Iacob, and Hillegersberg 2024).

A recurring conclusion is that DevSecOps does not scale without machine-enforceable governance that operates at pipeline speed. This motivates Policy-as-Code, which represents policies as executable logic that can be evaluated automatically and consistently across delivery stages.

Critical synthesis. The studies reviewed in this section converge on a single core finding: the security-velocity mismatch is structural, not incidental. Sánchez-Gordón and Colomo-Palacios 2020, Akbar, Smolander, et al. 2022, and Moyón, Angermeir, and Mendez 2024 all independently identify adoption barriers, but they differ in what they find to be most prominent. Sánchez-Gordón and Colomo-Palacios, through a systematic literature review focused on the human factor, identify cultural misalignment and the absence of a shared security mindset as the core barrier to DevSecOps adoption. Akbar et al., using a multivocal literature review and fuzzy TOPSIS prioritisation, find that the highest-priority challenges are the lack of secure coding standards and the lack of automated security testing tools — placing standards and tooling deficits at the top of the problem hierarchy. Moyón et al., drawing on industrial case studies with practitioners at three companies in regulated environments, find a more balanced set of challenges spanning five categories: continuous development integration, value stream visibility, efficiency, knowledge transfer, and CI/CD pipeline automation — with no single category dominant. A limitation shared across all three is that they characterise and prioritise challenges without providing or evaluating concrete technical enforcement mechanisms. This section establishes the *why* behind the problem this dissertation addresses: the convergence across independent studies confirms that the absence of automated pipeline-integrated governance is a real and persistent problem; the divergence in proposed remedies confirms there is no single accepted solution.

2.3 CI/CD Pipelines and Integration Points for Policy Enforcement

Modern CI/CD pipelines represent the operational backbone of contemporary software delivery, as they orchestrate activities like code compilation, dependency resolution, testing, artifact packaging, generation of container images, infrastructure provisioning, and deployment. Their defining attribute is automation: pipelines execute deterministic sequences of tasks at high frequency, often triggered by individual commits. The same automation that enables rapid and reliable delivery also introduces systemic risk. Misconfigurations, insecure dependencies, and policy violations can be propagated automatically and at scale.

Empirical studies show that many security and compliance failures originate before deployment. These issues are embedded in artefacts that pass through CI/CD workflows without adequate validation. The most common failure examples are IaC templates configured incorrectly, default permission schemes set to insecure states, libraries having known vulnerabilities, and overly permissive access controls (Alnafessah et al. 2021) (Kapetanidou, Nizamis, and Votis 2025). When security validation is absent or postponed, issues are often detected only at runtime—if at all. This leads to higher remediation costs, increased regulatory exposure, and greater operational risk. CI/CD pipelines are not only automation mechanisms, they also define the most effective control points for continuous governance. When policy checks are missing or delayed, violations often surface only at runtime, increasing remediation cost and regulatory exposure (Kosińska and Zieliński 2023). However, security tools are often bolted onto pipelines as optional steps, poorly integrated, inconsistently configured, or routinely ignored by developers (Moyón, Angermeir, and Mendez 2024). Manual

approval gates are often bypassed to maintain delivery velocity, and many organizations do not possess formally defined processes to identify where and how policy enforcement should take place (Ramaj et al. 2024). Across multiple studies, a recurring insight emerges: CI/CD pipelines expose numerous integration points for automated and enforceable security policies, yet these opportunities remain systematically neglected in practice. Effective enforcement distributes policy checks across the delivery lifecycle:

2.3.1 Source-Level Enforcement (Pre-Commit / Pre-Merge)

Policies at this stage validate:

- Adherence to secure coding standards;
- Secret detection;
- Dependency licensing and vulnerability thresholds;
- Conventions regarding structure and naming for IaC and Kubernetes artifacts.

Despite their strategic value, source-level mechanisms remain inconsistently adopted, largely due to the absence of standardized policy definitions and cross-team governance structures (Sánchez-Gordón and Colomo-Palacios 2020).

2.3.2 Build and Artifact Construction

During this phase, binary artifacts, container images, and build metadata are created by pipelines. Research reveals that:

- Container configuration errors (e.g., root user execution, missing resource limits);
- Unscanned or outdated dependencies;
- Incomplete Software Bill of Materials (SBOM) generation.

Constitutes recurrent causes of downstream security incidents (Billawa et al. 2022). Although OPA-based validation at this stage remains uncommon, studies show that, when implemented, such policies effectively prevent misconfigurations from propagating to artefact registries.(Kapetanidou, Nizamis, and Votis 2025).

2.3.3 Static Validation of IaC and Deployment Manifests

This phase represents one of the most policy-dense stages in modern pipelines. Terraform configurations, Kubernetes manifests, Helm charts, and Ansible playbooks encode the intended system state, making them ideal for syntactic, semantic, and compliance-oriented validation. The literature, however, consistently highlights the difficulty organizations face in translating abstract governance requirements—such as encryption-at-rest mandates or principle-of-least-privilege RBAC constraints—into enforceable, machine-readable rules (Haverinen et al. 2024).

2.3.4 Deployment Gatekeeping and Runtime Admission Control

In Kubernetes environments, policy enforcement can be applied at deployment time via:

- Admission controllers (e.g., OPA Gatekeeper, Kyverno);

- Runtime policy engines;
- Continuous compliance monitors.

Rossi et al. 2025 demonstrate that policy-driven admission control and scheduling significantly reduce the incidence of misconfigurations entering production clusters. Yet, these mechanisms remain unevenly adopted due to integration complexity and gaps in organizational tooling maturity.

2.3.5 Post-Deployment and Continuous Compliance Loops

Following deployment, pipelines may initiate periodic compliance scans or respond to alert-driven workflows. Identity-driven approaches, such as those described by Gama et al. 2024, show that continuous vulnerability assessment and automated identity provisioning mechanisms can reduce configuration drift. However, these mechanisms require consistently structured, machine-readable policy models to operate reliably.

Critical synthesis. Across the five pipeline stages surveyed, the literature converges on one finding: every stage offers an enforcement opportunity, and nearly every stage is underused. The divergence lies in which stage each study treats as its focus. Rossi et al. 2025 evaluate admission control and scheduling in Kubernetes, demonstrating that OPA Gatekeeper can enforce governance at deployment time. Gama et al. 2024 focus on post-deployment compliance, showing that vulnerability assessment must extend beyond the pipeline to address runtime changes. Source-level and build-time enforcement receive less empirical attention: the reviewed studies acknowledge these stages as strategically valuable but note that governance mechanisms here are rarely standardised or consistently applied (Akbar, Smolander, et al. 2022; Sánchez-Gordón and Colomo-Palacios 2020). A critical limitation of the reviewed corpus is the near-total absence of studies that empirically evaluate enforcement across more than one pipeline stage simultaneously. Each study evaluates one stage in isolation without addressing how multiple stages interact or complement each other. It is precisely this multi-stage coordination problem that this dissertation addresses: rather than choosing a single enforcement point, the proposed architecture integrates CI validation and admission control as complementary layers using a shared policy language.

2.4 Cloud-Native Security and Microservices Challenges

Cloud-native and microservices-based architectures introduce security challenges that differ fundamentally from those of traditional applications. In traditional applications, security control points are relatively stable and well understood. In contrast, cloud-native environments rely on dynamic and ephemeral resources, which complicates consistent security enforcement.

Billawa et al. 2022 report that microservices exacerbate common security vulnerabilities, including improper usage of default configurations, “dependency sprawl,” and inconsistent enforcement of access control rules. Their work on systemization of knowledge proves that “inappropriate configuration in K8s Role-Based Access Control (RBAC), too permissive service accounts, or insufficient network policies” are persistent and very problematic failure modes. They further observe that each microservice introduces additional configuration and attack surfaces, making end-to-end security verification substantially harder to manage.

Research on autonomic management aligns with this review. Kosinska and Zielinski (2023) show that cloud-native ecosystems are prone to configuration drift and resource fluctuations that cannot be effectively addressed through periodic manual scans. Their findings indicate that effective governance in dynamic environments requires policy-driven automation capable of operating across distributed system components.

Cloud-native applications rely on multiple abstraction layers, including containers, orchestration frameworks, service meshes, and Infrastructure-as-Code templates. Each has its own set of security and compliance requirements. This has created two important challenges, widely reported in existing literature:

2.4.1 Policy Fragmentation Across Layers

To govern cloud-native applications effectively, it is important to implement security policies on various levels of abstraction, including:

- Container level: image scanning, base image provenance, privilege minimization;
- Kubernetes level: RBAC constraints, network policies, admission controls;
- Service mesh level: mTLS enforcement, traffic encryption, identity propagation;
- IaC level: Terraform resource configuration, encryption requirements, network topology restrictions.

As shown by Billawa et al. 2022, organizations tend to implement these layers in inconsistent ways, resulting in fragmented or conflicting security postures. This inconsistency can act directly against DevSecOps ideals, in which consistent and uniform policy enforcement are emphasized throughout the supply chain.

2.4.2 Configuration Explosion and The Impossibility of Manual Governance

Microservices architectures generate a substantially higher volume of configuration artefacts than traditional deployment models. Typical manifestations of this complexity include:

- Hundreds of YAML manifests;
- Thousands of container images versions;
- Dynamic Deployment Scaling;
- Continuously changing inter-service communication rules.

Even with relatively minor misconfigurations, such as a nonexistent resource constraint or an unintentional network ingress, integrity can become compromised. The volume and complexity of configuration data exceed the capacity of human-driven security practices, particularly in the absence of formal governance frameworks capable of expressing enterprise security requirements as enforceable configuration rules (Moyón, Angermeir, and Mendez 2024). As a result, misconfiguration, rather than the absence of security tools, has emerged as the primary cause of cloud-native security incidents.

2.4.3 Dynamic Environments Demand Automated and Declarative Controls

Cloud-native applications are highly dynamic and short-lived: containers and pods may exist for only a few seconds. Traditional security approaches—such as perimeter defences,

scheduled scans, and manual approval gates—cannot operate effectively at this pace. Consequently, security and compliance requirements must be defined as formal, machine-readable policies that can be evaluated automatically. Policy-as-Code frameworks address this need by allowing declarative policies to be enforced during build, deployment, and runtime stages. Empirical evidence supports this approach: Kapetanidou, Nizamis, and Votis 2025 show that vulnerability scanners alone are insufficient to prevent Kubernetes misconfigurations, and that policy-based admission controllers, such as OPA Gatekeeper or Kyverno, are necessary to block non-compliant workloads before deployment.

Critical synthesis. The three challenges discussed in this section — policy fragmentation, configuration explosion, and the pace of dynamic environments — are consistently reported across the reviewed literature, but the proposed solutions differ in scope and depth. Billawa et al. 2022 characterise the problem at the microservice architecture level, focusing on RBAC and network policy misconfigurations; Kapetanidou, Nizamis, and Votis 2025 evaluate it at the tooling level, comparing scanners and admission controllers; Kosińska and Zieliński 2023 approach it from an autonomic management perspective, focusing on runtime drift. A notable divergence is between studies that treat configuration problems as primarily *organisational* (requiring better processes and team alignment) and those that treat them as primarily *technical* (requiring better tooling and automation). This dissertation takes the technical position: the problem is tractable through automated, policy-driven enforcement applied at well-defined pipeline stages, without requiring organisational transformation. A limitation shared by all three studies is that they describe and categorise the problem but do not provide reproducible, empirically validated enforcement mechanisms — which is what this dissertation contributes.

2.5 Formalizing Security and Compliance Requirements

A recurring problem in DevSecOps research is the absence of a uniform and practical approach for translating high-level governance obligations into implementable technical controls. This problem persists despite the widespread adoption of automation pipelines and cloud-native technologies, as regulatory obligations are still defined primarily in linguistic and non-verifiable terms. This represents a fundamental limitation of contemporary security governance frameworks.

Empirical evidence supports this claim. Ramaj et al. 2024 states that organizations have to rely on manual or retrospective compliance measures instead of imposing governance directly within CI/CD pipelines since they lack structured mappings between legal, business, and technical controls.

Haverinen et al. 2024 continue to emphasize this issue with their proposal for the Security-as-Code Information Model.Their evaluation shows that compliance enforcement is typically distributed across teams and tools, each maintaining its own policy representation. This leads to semantic drift, inconsistent enforcement, and delayed detection of violations. This fragmentation leads to semantic drift, where policy meanings diverge across organizational layers, resulting in incomplete or delayed enforcement, often detected only late in the pipeline or after deployment. Their evaluation also verifies that machine-enforceable representations for policy modeling provide capacities for automatic generation and traceable decision-making, which are inadequately present in most industry DevSecOps environments.

These challenges are particularly pronounced in environments where manual controls are infeasible due to scale or time constraints. Analysis on critical infrastructures has shown that the lack of formal, machine-readable representations for compliance is part of the factors that can cause uncertainty in security postures or can lead to higher risks during operations (Ramaj et al. 2024).

Within the reviewed literature, four structural limitations emerge invariably:

2.5.1 Lack of Standardized Compliance Life Cycle

Multiple studies identify systemic deficiencies in how organizations manage compliance artefacts:

- **Lack of version control**
  Requirements for compliance are usually contained in static documents rather than version control repositories.
- **Lack of Peer Review or Testing for Governance Rules**
  The validation of policy logic is rarely ever accomplished within realistic execution environments.
- **No mechanisms for deprecation or evolution**
  Organizational policies tend to accumulate over time and may become outdated or internally inconsistent.
- **Lack of coordination between various functions**
  It has implications on how risk, legal, security, and engineering teams interpret requirements for implementation.

Haverinen et al. suggest that simple "codification" is not sufficient but rather that "definition, validation, approval, deployment, monitoring, and deprecation" are required within a "complete lifecycle" to ensure "policy integrity." This observation is consistent with the findings reported across the surveyed literature.

2.5.2 Lack of machine-verifiable mappings between high-level requirements and technical controls

The existence of structural disparity between governance and technical means has been established in various documents. It emerges in terms of:

- vague high-level guidance that resists direct technical translation (for example, "encrypt data in transit");
- inconsistent representations of policy in documents, scripts, and checklists;
- retrospective audit-based validation rather than continuous enforcement;
- platform-specific semantics that can complicate policy translations.

Both Ramaj et al. 2024 and Haverinen et al. 2024 assert that DevSecOps pipelines are unable to achieve correction and consistency without specifications that are able to formulate constraints in terms that are deterministic and executable.

2.5.3 Lack of Integration between Evidence Collection and Auditability in CI/CD Pipelines

Compliance evidence, policy decisions, and audit records are typically:

- manually captured,
- scattered throughout diverse tools, or
- unavailable for retrospective analysis.

Haverinen et al. 2024 show how by making the policy logic computationally executable, it becomes feasible to generate evidence documents in structured form (such as JSON decision logs), track audit trials in an immutable fashion via GitOps, and support traceable rationales for enforcement outcomes. This is crucial for enforcement purposes and for regulatory audits both externally and internally.

2.5.4 Formalization as a Foundational Precondition for Policy-as-Code

The reviewed studies converge on a central conclusion: Policy-as-Code depends on prior formalization of governance requirements.

PaC tools, such as OPA or Kyverno, must support:

- precise and unambiguous policy semantics,
- explicit constraint definitions,
- deterministic evaluation models,
- machine-consumable input (JSON or YAML),
- alignment with organizational governance structures.

Vakhula, Opirskyy, et al. 2025, examining formalizations for ABAC/RBAC using PaC, argue for the importance of formalized specifications in ensuring policy correctness and maintenance. Their research illustrates that semi-formal definitions result in direct misinterpretation, conflict, and subsequent gaps in enforcement.

Critical synthesis. The four structural limitations discussed in this section are independently identified across the reviewed corpus, which is itself significant: the convergence across studies with different methodologies, domains, and motivations suggests these are systemic rather than context-specific problems. However, the studies diverge substantially on what is to be done about them. Haverinen et al. 2024 propose a concrete artefact — an open information model for Security-as-Code — that addresses lifecycle management and evidence collection at the architectural level. Vakhula, Opirskyy, et al. 2025 propose formalized ABAC/RBAC specifications as a solution to semantic drift. Ramaj et al. 2024 identify the gap but stop short of a technical solution, framing it as an organisational and process problem. A critical limitation of all three approaches is scope: each addresses a specific domain (security models, access control, critical infrastructure) and does not generalise to arbitrary DevSecOps pipelines or workload governance. None of the reviewed works addresses the formalization-to-enforcement gap in the context of Kubernetes workload governance within a CI/CD pipeline — the specific context this dissertation occupies. The formalization problem motivates the policy selection rationale in Chapter 6: the five implemented policies were chosen precisely because they are directly translatable from well-known

Kubernetes security guidance into deterministic, machine-enforceable Rego rules without requiring additional formalization steps.

2.6 Policy-as-Code Foundations

One of the most prominent developments in DevSecOps is Policy-as-Code (PaC), which expresses security, availability, and governance policies in machine-computable formats. Instead of relying on lengthy prose documents prone to interpretation and manual approval delays, PaC expresses policies as executable rules that can be automatically evaluated at multiple stages of the software development lifecycle. The literature indicates that PaC emerged in response to deficiencies in existing security practices, including fragmented policy evaluation, limited automation, and poor scalability in CI/CD-driven cloud-native environments (Haverinen et al. 2024; Ramaj et al. 2024).

2.6.1 Conceptual Foundations of PaC

Conceptually, PaC treats policies as structured knowledge expressed through declarative semantics. Policies typically specify what conditions must be satisfied rather than how enforcement is implemented, relying on structured inputs such as JSON or YAML artefacts. This has minimized interpretive work, according to Vijayaraghavan 2025, with informal assertions like "all workloads that are deployed must adhere to the organization's security standards" replaced by assertions expressed in languages such as Rego.

Formalization of policies using code offers several well-documented benefits: modularity (policies decomposable into reusable components), version-controlled traceability, testability, auditability, and automation-readiness. These properties appear consistently across the reviewed literature and are not meaningfully contested. The convergence is notable: studies as methodologically different as the practitioner survey of Akbar, Smolander, et al. 2022 and the formal information-model proposal of Haverinen et al. 2024 both arrive at the same list of desiderata for machine-enforceable governance. Where the literature diverges is on how to achieve these properties in practice — particularly the gap between PaC as a concept and PaC as an integrated, lifecycle-managed programme. This dissertation focuses on the integration gap at two specific pipeline stages, leaving lifecycle management as acknowledged future work.

2.6.2 Evaluation Models and Enforcement Architectures

PaC requires evaluation engines capable of handling declarative policies at scale and across diverse artefact types. The most widely adopted approach is Open Policy Agent (OPA), founded on Datalog-inspired logical evaluation, constraint checks on structured configuration data, context-isolated decision-making, and architecture-agnostic policy evaluation.

The flexibility of OPA enables enforcement across CI/CD pipelines, container registries, infrastructure provisioning workflows, Kubernetes admission controllers, service meshes, and API gateways. The growing adoption of OPA within Kubernetes security workflows has been reported by Kapetanidou, Nizamis, and Votis 2025 and Rossi et al. 2025, including its use in policy-driven scheduling.

Although other engines exist — including Kyverno, Service Control Policies, and HashiCorp Sentinel — they are not equivalent in expressiveness or evaluation model breadth. Among

existing solutions, OPA/Rego remains the only policy engine with native support for both CI-stage validation (via Conftest) and Kubernetes admission control (via Gatekeeper), which is precisely the cross-stage reuse property this dissertation exploits.

2.6.3 Policy as Code for Shift-Left Securing

The literature consistently identifies PaC as a key enabler of shift-left security within DevSecOps practices. PaC introduces automated checks within early pipeline stages, enabling early detection of misconfigurations, consistent enforcement on build and deployment artefacts, cross-team reuse of standardised governance rules, automated blocking of non-compliant artefacts, and reduced reliance on manual security review.

Empirical studies show that shift-left PaC enforcement reduces recurrent problems, including improperly configured IaC templates (Haverinen et al. 2024), insecure container images (Kapetanidou, Nizamis, and Votis 2025), RBAC/ABAC misaligned models (Vakhula, Opirskyy, et al. 2025), and non-compliant Kubernetes manifests (Rossi et al. 2025). PaC operationalises these requirements by translating them into explicit, verifiable constraints.

Synthesis. Across subsections 2.6.1–2.6.3, the literature converges on the shift-left value proposition of PaC but diverges on implementation depth. Studies proposing frameworks (Haverinen et al. 2024; Vijayaraghavan 2025) describe what PaC *should* do; empirical studies (Kapetanidou, Nizamis, and Votis 2025; Rossi et al. 2025) report what specific tools do in narrow scenarios. The gap between aspiration and evidence is the space this dissertation occupies: rather than proposing a new framework or evaluating a single tool, it empirically tests whether two existing tools can be composed to achieve shift-left detection and bypass-mitigation simultaneously.

2.6.4 Challenges and Limitations of Current Approaches

Although PaC has many benefits, various factors that restrict its adoption and scalability are described within existing literature.

Complexity and Learning Barriers

Although very expressive, Rego's declarative paradigm also has cognitive complexity. According to Vakhula, Opirskyy, et al. 2025, common mistakes are often encountered in practice, for example, under-constrained rules, too restrictive constraints, and unreasonable logical conflicts. This also identifies the requirement for valid policy verification and safe guidance for model development.

Tooling and Ecosystem Fragmentation

DevSecOps pipelines handle various artefacts, including IaC templates, Kubernetes manifests, CI definitions, and container specs, that are not always amenable to uniform treatment using individual policy modules. This is why Haverinen et al. discuss how, "absent common information models for PaC, its integration can suffer from inconsistencies and redundancy" (Haverinen et al., 2024).

Lack of Policy Lifecycle Management

It appears from the various studies examined that organizations rarely undertake holistic policy lifecycles comprising:

- systematic policy review procedures;
- versioning conventions;
- deprecation workflows;
- policy-specific test suites.

The absence of standardized lifecycle management limits long-term maintainability and scalability, representing a key research gap addressed by this thesis.

Performance Considerations

Although PaC engines are generally efficient, several studies report measurable overhead when enforcing policies in high-throughput environments, such as Kubernetes admission control (Rossi et al. 2025).

Synthesis. Across the reviewed literature, three points of convergence emerge consistently. First, all studies acknowledge that PaC tools are technically mature but organisationally underused — the barrier is not capability but adoption and integration. Second, the absence of a defined policy lifecycle is identified independently by Haverinen et al. 2024, Vakhula, Opirskyy, et al. 2025, and Vijayaraghavan 2025, suggesting this is a structural rather than incidental gap. Third, enforcement is consistently described as fragmented: tools operate at isolated pipeline stages without coordination. Where authors diverge is on which gap is primary — Kapetanidou, Nizamis, and Votis 2025 treat tooling immaturity as the main barrier, while Moyón, Angermeir, and Mendez 2024 and Akbar, Smolander, et al. 2022 treat organisational misalignment as more fundamental. This dissertation positions itself squarely in the tooling gap: it does not address organisational adoption barriers, but directly tackles enforcement fragmentation by demonstrating a multi-stage approach that coordinates CI and admission-time enforcement using a single policy language.

2.7 OPA/Rego and the Policy Tooling Ecosystem

OPA together with its declarative language Rego is the largest and technically the most advanced ecosystem for the implementation of Policy-as-Code (PaC) in modern DevSecOps and cloud-native scenarios. The surveyed literature attributes OPA's eminence to its architectural neutrality, expressive policy model, and independence from specific runtime environments (Kapetanidou, Nizamis, and Votis 2025; Rossi et al. 2025; Vijayaraghavan 2025).

2.7.1 Architectural Neutrality and Multi-Layer Enforcement

As a main component of a policy decision framework (PDE), OPA may be regarded as a general-purpose PDE that can evaluate policies uniformly even in heterogeneous contexts. Contrary to a vendor-specific-level set of solutions, OPA does not pinpoint a single layer of the infrastructure thus, it may be applied to:

- build-time validation (e.g., Conftest);
- CI pipelines: e.g., GitHub Actions, GitLab CI, or Jenkins;
- Kubernetes admission control via OPA Gatekeeper;
- cloud provisioning orchestrations: e.g., Terraform or CloudFormation;
- API gateways and service meshes via Envoy external authorization;
- runtimes for microservices and multi-tenant authorization systems.

Kapetanidou, Nizamis, and Votis 2025 identify OPA as one of the few policy engines capable of supporting both static validation and runtime enforcement in dynamic environments. Complementarily, Rossi et al. 2025 demonstrate its feasibility in latency-sensitive cluster scheduling scenarios.

2.7.2 Rego: Declarative Semantics, Strengths, and Limitations

Rego is a Datalog-inspired, declarative language. It is used to express policies as logical predicates over structured data. Its academic relevance derives from three core characteristics:

Declarative Semantics. Policies define the necessary conditions to be met instead of outlining procedural enforcement steps. This enables concise representations of constraints, rules, and quantifiers.

Data-Oriented Evaluation. OPA natively consumes JSON and YAML inputs, enabling Rego policies to evaluate Kubernetes manifests, Terraform plans, CI configurations, container metadata, and API payloads.

Composability and Reusability. The policy framework may be implemented as a package along with different reusable modules thus standardizing cross-team governance principles (Vijayaraghavan 2025).

Known Limitations. In spite of the pros, Rego presents quite a steep learning curve. Vakhula and Opirskyy 2025 report that teams unfamiliar with declarative logic frequently introduce semantic errors, including over- or under-constrained rules, conflicting policies, and unexpected evaluation outcomes.

2.7.3 Policy Enforcement Runtimes: Gatekeeper, Conftest, and Complementary Tools

OPA is only the decision tool; a different tier of runtimes is needed for enforcement:

Gatekeeper (Kubernetes Admission Control)

Gatekeeper connects OPA to Kubernetes with the help of CRDs thus, policy enforcement at the time of admission becomes possible. Kapetanidou, Nizamis, and Votis 2025 demonstrate that admission control detects misconfigurations earlier than scanners, while Rossi et al. (2025) show that performance is within acceptable limits in complex scheduling scenarios.

Conftest (Static Policy Validation)

Conftest allows Rego-driven checks to be run in CI processes, thus supporting IaC, Kubernetes manifests, container metadata, and pipeline config validations. The so-called "shift-left" enforcement is recognized as a very important missing link in the majority of DevSecOps pipelines (Akbar, Smolander, et al. 2022; Haverinen et al. 2024).

Comparative Ecosystem

Kyverno offers a Kubernetes-native model for policies but lacks expressiveness and is not capable of functioning outside Kubernetes contexts (Kapetanidou, Nizamis, and Votis 2025). Vendor-specific instruments like HashiCorp Sentinel or AWS SCPs provide a limited number of use cases in heterogeneous or multi-cloud environments.

Synthesis. The literature reveals an important asymmetry: Gatekeeper and Conftest are frequently cited as dominant enforcement tools (Kapetanidou, Nizamis, and Votis 2025; Rossi et al. 2025; Vijayaraghavan 2025), yet no study in the reviewed corpus empirically evaluates both tools *together* as complementary enforcement stages within the same pipeline. Studies that mention Conftest treat it as a standalone linter; studies that evaluate Gatekeeper treat it as a runtime gate operating independently. The integration of the two — using the same Rego logic across both stages — represents a gap in empirical validation that this dissertation addresses directly. Furthermore, the key differentiator between OPA/Rego-based tools and alternatives such as Kyverno is precisely this cross-stage reuse: Kyverno policies are YAML-based and have no equivalent CI-stage tool, making a unified enforcement policy impossible without maintaining two separate policy implementations.

2.7.4 Integration Challenges and Open Issues

Despite its maturity, the OPA ecosystem exhibits several limitations:

- Inconsistent tooling support across CI/CD platforms (Haverinen et al. 2024);
- Absence of standardized testing methodologies for validating policy correctness (Vakhula and Opirskyy 2025);
- Lack of formal lifecycle governance, including versioning, review, and deprecation workflows;
- Performance considerations for high-throughput workloads such as admission control and scheduling (Rossi et al. 2025).

These challenges constrain the reliability and scalability of PaC deployments and motivate the need for structured governance models, a topic addressed in the subsequent sections of this dissertation.

2.8 Infrastructure as Code and GitOps

Infrastructure-as-Code (IaC) and GitOps have transformed how cloud-native applications are provisioned, deployed, and managed. IaC enables infrastructure to be defined as machine-readable artefacts, while GitOps extends this model by establishing version-controlled repositories as the source of truth for both application and infrastructure state.

Although these approaches significantly increase automation, reproducibility, and deployment speed, they also amplify the impact of configuration errors encoded in declarative artefacts.

2.8.1 Infrastructure-as-Code: Scalability, Complexity, and Failure Modes

IaC solutions such as Terraform, Ansible, Helm, and Kubernetes configuration frameworks enable entire cloud infrastructures to be defined declaratively. Although this is very beneficial for scalability, this approach also centralizes the risk, meaning the security and correctness of production environments depend entirely on the accuracy of the underlying artefacts.

Empirical studies show that IaC alone does not eliminate configuration drift. For example, a study conducted by Alonso, Piliszek, and Cankar 2023 has shown that operational changes made outside the version-controlled pipelines might even reintroduce the problem of configuration drift without being detected.

A significant amount of existing literature points towards IaC misconfigurations being the root cause of security incidents in cloud-native systems. Issues may include:

- Overly permissive network ingress and firewall rules;
- Publicly exposed storage resources;
- Missing or inconsistent encryption-at-rest configurations;
- Misconfigured Kubernetes role-based access controls (RBAC);
- Excessive container privileges and insecure runtime settings.

As highlighted by Billawa et al. 2022 and Kapetanidou, Nizamis, and Votis 2025, artefact failure and failure to deliver artefacts are not driven by ill will. Rather, they are driven by cognitive overload, artefact heterogeneity, and a lack of machine-enforced validation.

The governance dimension of IaC remains particularly weak. Ramaj et al. 2024 observe that most organizations lack formal processes for translating governance objectives—such as least-privilege access, encryption requirements, or audit logging—into enforceable infrastructure constraints. Compliance is therefore commonly addressed through:

- Ad hoc validation scripts;
- Manual peer reviews;
- Periodic external audits.

These approaches cannot scale to the pace and complexity of the environment enabled by IaC. All the above reasons further emphasize the need for a formal governance layer like Policy-as-Code.

2.8.2 GitOps: Declarative Operations and Governance Risks

GitOps extends the IaC paradigm by introducing continuous reconciliation mechanisms alongside declarative definitions. Tools such as Argo CD monitor version-controlled repositories and automatically enforce the desired state. This model provides several operational guarantees:

- Immutable and auditable change histories;
- Reproducible and environment-consistent deployments;

- Automated reconciliation ensuring state convergence;
- Disciplined change control through pull or merge requests.

However, GitOps simultaneously amplifies governance risks. Because repositories are authoritative, any misconfiguration or security violation merged into Git is automatically propagated to production. The loops of reconciliation can further cause these misconfigured values to be reiterated in production till the respective artefact in Git is fixed. According to Vijayaraghavan 2025, pipelines for GitOps pipelines cannot operate safely without automating the enforcement of these policies at the boundaries of commit and merge.

2.8.3 Why IaC and GitOps Require Policy-as-Code

Across the reviewed literature, IaC and GitOps emerge as contexts where Policy-as-Code is not only advantageous but necessary. High-speed deployment makes human review inefficient, especially when accompanied by:

- Continuous reconciliation loops;
- Ephemeral and short-lived resources;
- Rapid scaling and redeployment events;
- Frequent environment regeneration.

Haverinen et al. 2024 emphasize that Security-as-Code must accompany IaC to ensure governance at operational speed.

Moreover, GitOps eliminates traditional safety nets. In declarative systems, an insecure definition that passes review is deployed automatically and persists until explicitly corrected. This structural characteristic makes pre-commit and pre-merge policy validation indispensable.

Finally, IaC artefacts are inherently structured, typically encoded in YAML or JSON. This makes them well suited for automated validation using PaC engines such as OPA/Rego, Kyverno, and Conftest. Policy-as-Code is therefore uniquely positioned to exploit the formal structure of IaC artefacts, enabling deterministic and repeatable governance enforcement that cannot be achieved through manual or document-based controls.

Critical synthesis. The reviewed literature on IaC and GitOps converges on one conclusion: declarative, automated infrastructure creates governance risks that only automated, declarative policy enforcement can address. However, the studies differ in scope and angle. Alonso, Piliszek, and Cankar 2023 propose a reference framework for trustworthy IaC covering four DevSecOps lifecycle phases, arguing that security must be embedded at every stage from development to operations — not added retrospectively. Vijayaraghavan 2025 focus specifically on the shift-left property of PaC, arguing that enforcement must occur at commit and merge boundaries to prevent misconfigurations from propagating through GitOps reconciliation loops. Haverinen et al. 2024 focus on the evidence and auditability dimension, showing that machine-executable policy logic enables structured audit trails that document-based governance cannot. A shared limitation is that none of these studies evaluate how pre-commit or CI-stage validation interacts with admission-time enforcement — they treat policy validation as a single-stage activity. This gap is directly addressed by the multi-stage architecture proposed in Chapter 6.

2.9 Synthesis and Research Gaps

The preceding sections cover six thematic areas: the DevOps-to-DevSecOps transition, CI/CD pipeline enforcement points, cloud-native security challenges, formalization of compliance requirements, PaC foundations and tooling, and IaC/GitOps governance. Taken together, the reviewed literature establishes a consistent picture of both what is understood and what remains unresolved.

What the literature agrees on. Across all thematic areas, a strong consensus exists on three points. First, manual and document-based security governance cannot operate at the pace and scale of modern software delivery. Second, Policy-as-Code — specifically OPA/Rego — is technically mature and capable of expressing deterministic, machine-enforceable governance rules. Third, the most common failure mode in practice is not the absence of security tools but their inconsistent, isolated, or partial integration into the delivery pipeline.

Where the literature diverges. Studies diverge most sharply on two questions: (1) whether the primary barrier to better governance is technical or organisational, and (2) at which pipeline stage enforcement should be prioritised. The organisational perspective (Sánchez-Gordón & Colomo-Palacios 2020; Moyón et al. 2024; Akbar et al. 2022) argues that tooling improvements alone are insufficient without cultural and process change. The technical perspective (Kapetanidou et al. 2025; Rossi et al. 2025; Vijayaraghavan 2025) argues that better-integrated tooling would directly improve outcomes regardless of organisational context. On enforcement stage, the literature is similarly fragmented: no study reviewed here advocates for and empirically evaluates a multi-stage approach covering both CI validation and admission control as complementary mechanisms.

What is missing. Despite the breadth of the reviewed corpus, two specific gaps remain consistently unfilled. No study provides a reproducible empirical evaluation of PaC enforcement across both a CI stage and a Kubernetes admission stage using a shared policy language. And no study examines the governance consequences of CI bypass — the scenario in which a non-compliant workload reaches the cluster without going through a CI pipeline — as a primary research question. These two omissions directly motivate the research questions and architecture of this dissertation.

2.9.1 Identified Research Gaps

Although there is maturity in the individual technologies, five gaps consistently hinder the effective adoption of Policy-as-Code in DevSecOps environments. Table 2.3 summarises these gaps, their supporting evidence, and whether this dissertation addresses them.

Table 2.3: Identified Research Gaps and Thesis Coverage

Gap	Representative Sources	Addressed in This Work?
Fragmented enforcement across pipeline stages — tools operate at isolated points without coordination	Haverinen et al. 2024; Akbar et al. 2022; Moyón et al. 2024	**Yes** — multi-stage CI + admission architecture directly addresses this
No empirical evaluation of multi-stage PaC enforcement in a reproducible setting	Rossi et al. 2025; Kapetanidou et al. 2025	**Partially** — controlled PoC evaluation with 29 manifests and 261 assertions
Absence of policy lifecycle management (versioning, testing, deprecation)	Haverinen et al. 2024; Vakhula et al. 2025; Vijayaraghavan 2025	**No** — explicitly out of scope
Lack of systematic mapping from regulatory requirements (ISO 27001, GDPR) to machine-enforceable policies	Ramaj et al. 2024; Haverinen et al. 2024	**No** — explicitly out of scope
Cross-platform CI/CD coverage — most PaC studies focus on Kubernetes alone	Prates & Pereira 2025; Akbar et al. 2022	**No** — GitHub Actions only

The gap this dissertation most directly addresses is the first: fragmented enforcement across pipeline stages. The second gap is partially addressed through the empirical evaluation. The remaining three gaps are acknowledged but fall outside the scope of this proof-of-concept work, as defined in Section 1.4.

2.9.2 Implications for This Thesis

The state of the art reflects clear recognition of the need for automated, policy-driven governance in DevSecOps environments, supported by a rapidly evolving ecosystem of PaC tools and frameworks. However, as Table 2.3 makes explicit, the literature stops short of delivering an integrated, empirically validated enforcement model that coordinates CI and admission-time policies using a shared policy language. This gap directly motivates the architecture proposed in Chapter 6 and the evaluation presented in Chapter 8. The three contributions of this dissertation — described in Section 1.6 — are positioned specifically to address Gaps 1 and 2 from Table 2.3, while acknowledging that Gaps 3, 4, and 5 remain open problems for future work.

Chapter 3

Research Design and Approach

3.1 Research Design and Approach

This dissertation adopts a design science research methodology Hevner et al. 2004, which is well suited to research that produces and evaluates an artefact intended to solve a practical problem. Design science research proceeds through the creation of a novel artefact — in this case, a multi-stage Policy-as-Code enforcement mechanism — and its rigorous evaluation against defined criteria within a relevant environment. This methodology is appropriate here because the research problem is not purely observational: it requires constructing a solution and assessing its behaviour under controlled conditions.

The research design consists of three primary phases. The first, *exploratory and analysis*, reviews existing knowledge across DevSecOps, cloud-native security, and Policy-as-Code to identify limitations, patterns, and adoption gaps. The second, *constructive*, develops the Policy-as-Code enforcement mechanism and integrates it with CI/CD workflows. The third, *evaluative*, assesses the mechanism using controlled scenarios against the research questions defined in Section 3.2.

This structure ensures alignment between the identified research gaps, the proposed solution, and the evaluation objectives.

3.2 Research Questions

The evaluation of this dissertation is guided by three research questions (RQs), defined below. These questions concern the artefact produced by this research and are answered by the experimental evaluation in Chapter 8. They are distinct in purpose from the thematic review structure of Chapter 2, which organises the literature synthesis rather than framing experimental outcomes.

RQ1: Can Policy-as-Code enforcement detect insecure Kubernetes configurations automatically?

RQ2: Does multi-stage enforcement (CI validation combined with admission control) improve governance coverage compared to single-stage enforcement?

RQ3: Does admission control mitigate the risks associated with CI pipeline bypass?

RQ1 addresses detection capability: whether Rego-based policies, evaluated by an automated engine, can identify known insecure configurations without manual intervention and without generating spurious violations. RQ2 addresses coverage: whether the combination of CI and admission-time enforcement provides governance that neither stage alone can

deliver. RQ3 addresses the most operationally important threat: whether the admission-control layer maintains governance guarantees when the CI pipeline is bypassed entirely, intentionally or otherwise.

These three questions are answered by the experimental evaluation in Chapter 8 and are revisited in the conclusions in Chapter 9. All requirements, scenario designs, and result tables in this dissertation use the exact wording above.

3.3 Proposed Solution Overview

The proposal involves the implementation of the CI/CD-integrated Policy-as-Code enforcement strategy to provide for the early, automated, and repeatable validation of security and compliance requirements. Rather than developing new policy languages or enforcement engines, the strategy leverages mature Policy-as-Code technology, specifically Open Policy Agent, to ensure relevance.

Conceptually, the resolution involves the enforcement of policies at various stages in the CI/CD pipeline, which makes it possible for the violation to be identified before deployment or at runtime. Policies are formulated using machine-readable rules, which are based on the artefacts that are created throughout the pipeline.

Further, the solution has incorporated the governance perspective aligned to GitOps, allowing policies to be treated as versioned artifacts for review and evolution. The solution is able to provide traceability and consistency throughout the development and operational stages.

The solution is presented as a proof-of-concept reference implementation rather than a production-ready platform.

3.4 Evaluation Strategy

The evaluation strategy is designed to assess the effectiveness and practicality of the proposed Policy-as-Code approach within a constrained DevSecOps pipeline setting. The objective of the evaluation is not to provide exhaustive coverage of security or compliance domains, but to empirically validate the feasibility and impact of automated policy enforcement under controlled and well-defined conditions.

3.4.1 Baseline Definition

To enable a meaningful evaluation, the proposed approach is assessed against a baseline that reflects common practice in CI/CD environments without explicit Policy-as-Code enforcement. The baseline consists of pipeline executions in which security and compliance checks are either absent or limited to default tool configurations, without custom, explicitly defined policies integrated into the pipeline.

This comparison allows the evaluation to isolate the contribution of Policy-as-Code mechanisms with respect to policy visibility, enforcement timing, and feedback provided to developers.

3.4.2 Evaluation Metrics

The evaluation relies on a set of quantitative and qualitative metrics selected to reflect both technical effectiveness and practical applicability. The primary metrics include:

- The number of policy violations detected across different pipeline executions.
- The enforcement stage at which violations are identified (e.g., early pipeline stages versus later stages).
- The additional execution overhead introduced by policy evaluation within the CI/CD pipeline.

Where appropriate, qualitative observations regarding the clarity and usefulness of policy feedback are also considered to complement quantitative measurements.

3.4.3 Evaluation Scenarios

The evaluation is conducted using a limited set of controlled scenarios designed to reflect typical usage conditions. These scenarios include pipeline executions with compliant artefacts as well as executions containing intentionally introduced policy violations. By varying the presence and type of violations, the evaluation examines how effectively the proposed approach detects and reports non-compliant configurations.

Multiple executions are performed for each scenario to reduce the influence of incidental variation and to support consistent interpretation of the observed results.

3.4.4 Success Criteria

The proposed approach is considered successful if it consistently detects relevant policy violations at predefined enforcement points within the pipeline, while maintaining acceptable execution overhead. In addition, the evaluation seeks to confirm that policy feedback is generated in a form that supports timely identification and resolution of issues by developers.

Failure to detect violations reliably, excessive pipeline performance degradation, or ambiguous feedback would indicate limitations of the approach and inform potential refinements or future extensions.

3.5 Validity and Limitations

Several limitations are inherent to the scope and design of this research. Firstly, the assessment is done on a limited number of CI/CD tools and configurations, which might have an impact on the generalization of the results obtained. Secondly, the chosen set of policies is representative but not comprehensive, and as such, not all categories of security/security compliance requirements might have been included.

Threats to internal validity may occur due to the test repository and evaluation circumstances chosen, whereas external validity is limited to the controlled experimental setup rather than industrial-scale experiments. These pitfalls are overcome by being transparent about assumptions and maintaining a consistent method of evaluation and conforming to generally available tools and techniques mentioned in the literature.

Despite these challenges, the work provides useful insights into the feasibility, value, and trade-offs of PaC enforcement in the early stages of the CI/CD pipeline, directly addressing the shortcomings identified in the state-of-the-art review.

Chapter 4

Ethical Considerations

4.1 Ethical and Legal Compliance

This study adheres to the guidelines that govern research integrity, research ethics, and associated laws. The study neither involves human participants nor personal data collection, nor has it used survey research or experiments conducted with human subjects. The study, therefore, does not need IRB or ethics committee review.

The study exclusively concentrates on the analysis, design, and evaluation related to the technical techniques for the enforcement of Policy-as-Code. Every testing happening in the study is conducted on open-source repositories or specifically designed test cases. None of the proprietary systems, confidential business information, or restricted infrastructures are used when executing the study.

Legally speaking, the conducted research satisfies the licensing terms regarding software. All the tools, frameworks, and libraries used, such as Open Policy Agent, CI/CD integration, and the supporting tooling, are used in compliance with their respective open-source licenses. The artifacts developed as part of this body of research are original and do not use any unauthorized copyrighted materials.

Moreover, the proposed approach is not intended to evade, undermine, or circumvent security measures. Instead, it aims at enhancing the governance, compliance, and accountability focus throughout the software delivery process. As a matter of fact, the research is therefore based on ethical foundations which encourage secure and responsible software development practices.

4.2 Data Privacy and Confidentiality

Although this research is not working with personal or sensitive data, issues related to data protection and confidentiality still come into play. The scenarios for data evaluation involve configuration files, infrastructure, pipeline metadata, and artificial/synthesized data related to security violations. No personal identifiers, passwords, or sensitive operational data are involved.

All the repositories considered in the evaluation phase can be accessed publicly and chosen to eliminate the risk of exposing personal information. For situations where the test scenarios involve the need for vulnerable or policy-violated configurations, these scenarios are implemented in controlled environments made solely for the purpose of experiments.

The study is guided by the principles of data minimisation and purpose limitation. Data processed during the evaluation is used solely for assessing policy enforcement effectiveness and performance; no data is retained beyond the research scope or shared outside research reporting. On a wider ethical perspective, the study acknowledges the possible social implications of automated governance: while Policy-as-Code offers improvements in security and compliance, poorly designed policies carry risks of over-compliance, reduced developer autonomy, or decision-making opacity. In summary, this research was designed with full consideration of privacy, confidentiality, and responsible automation, operating within the guidelines of ethical research practice and applicable data protection regulations.

Chapter 5

Project Planning and Feasibility

5.1 Project Scope and Constraints

This work proposes to focus on the design, development, and assessment of a Policy-as-Code (PaC) enforcement method that can be implemented as part of CI/CD integration pipelines. This dissertation confines its scope to be feasible within the time constraints of a Master's program. The primary scope of work encompasses the following:

- Identification and formalization of representative security and compliance requirements relevant to modern CI/CD pipelines;
- Development of declarative policies using Policy-as-Code techniques, primarily based on Open Policy Agent and Rego;
- Integration of policy evaluation into CI/CD workflows at selected pipeline stages;
- Experimental evaluation of policy enforcement effectiveness and performance overhead using controlled and open-source scenarios.

The project explicitly excludes:

- Development of a full production-grade security platform;
- Large-scale industrial deployment in proprietary environments;
- Runtime intrusion detection or behavioral anomaly detection mechanisms;
- Legal certification or formal compliance auditing against regulatory standards.

Certain limitations are inherent to the implementation scope. This research was conducted by a single researcher with access limited to publicly available tools and infrastructure. Technological constraints include dependence on open-source CI/CD tools and the inability to replicate large-scale enterprise environments. The overall dissertation timeline further necessitated a controlled experimental scope.

5.2 Risk Analysis and Mitigation

The study has both technical and methodological risks that can either form or undermine the project outcome. All risks should be addressed to ensure the project can be feasible.

A technical risk is related to Policy-as-Code authoring, especially for describing non-trivial policy compliance requirements in declarative languages like Rego. A poorly crafted policy could introduce false positives or false negatives, degrading the outcome of policy evaluation.

While this risk exists, it could be controlled through careful selection of policies, ensuring they are representative of test cases, and checking them for conformance through test cases.

Another is related to the variability of integration of CI/CD tools. Similarities and differences between tools might affect the evaluation of policies, making it a concern for the project. In this case, the project concentrates on the common tools used in the industry, which results in the generality of the concepts.

Risks and challenges associated with methodologies relate to the deficiency of evaluation data and/or uncertain measurements of system behavior. These can be mitigated by ensuring that the experimental designs can be reproduced with similar scenarios and through relative measurements.

Finally, there could be a problem of scope creep, specifically with the wide range of issues related to the scope of DevSecOps. However, care will be taken to strictly focus on the objectives and avoid the issues covered in the exclusion criteria to stay on track with the main research questions.

5.3 Schedule and Milestones

The project is structured into sequential phases aligned with the research methodology and dissertation timeline. The major phases include:

- Literature review and problem analysis;
- Definition of requirements and policy scope;
- Design of the Policy-as-Code enforcement approach;
- Implementation and CI/CD integration;
- Experimental evaluation and data collection;
- Analysis of results and documentation.

These phases are mapped onto a project schedule using a Work Breakdown Structure (WBS) and a Gantt chart, which detail task dependencies, durations, and milestones. The planning ensures that critical activities—such as implementation and evaluation—are allocated sufficient time while maintaining flexibility to accommodate iterative refinement.

Milestones are defined at the completion of each major phase, providing clear checkpoints for progress assessment and alignment with dissertation milestones.

5.4 Monitoring and Control

Project monitoring and control are conducted through continuous progress tracking against the defined schedule and milestones. Task completion is reviewed regularly to identify deviations from the plan and to enable timely corrective actions.

Version control systems are used to manage all artefacts, including policies, CI/CD configurations, and experimental scripts, ensuring traceability and reproducibility. Incremental validation of intermediate results—such as policy correctness and pipeline integration—serves as an additional control mechanism to reduce late-stage rework.

Risks are revisited periodically throughout the project lifecycle, and mitigation strategies are adjusted as necessary. This lightweight but systematic monitoring approach is appropriate for the scale of the project and ensures that feasibility is maintained without introducing unnecessary management overhead.

Overall, the project plan demonstrates that the proposed work is realistic, well-scoped, and achievable within the available time and resources, satisfying the feasibility requirements of this dissertation.

Chapter 6

Solution Design and Architecture

This chapter details the proposed solution architecture and the design decisions made for the proof-of-concept implementation. The goal is to achieve multi-stage Policy-as-Code (PaC) enforcement within a Kubernetes-oriented DevSecOps pipeline, integrating CI-stage validation with Kubernetes admission control.

The objective is to address the security and governance challenges associated with Kubernetes workload configuration without compromising pipeline automation or developer usability.

6.1 Requirements

The solution proposed was created according to a set of functional and non-functional requirements formulated on the basis of the research questions and governance issues found in the literature review.

6.1.1 Functional Requirements

Table 6.1 presents the functional requirements defined for the proposed proof of concept and their traceability to the research questions.

Table 6.1: Functional Requirements

ID	Requirement	Traceability
FR-01	The system shall automatically assess Kubernetes manifests against a predefined collection of security policies, eliminating the need for manual review.	RQ1
FR-02	The system shall cause CI pipeline failure for non-compliant manifests during validation workflows.	RQ1, RQ2
FR-03	The system shall accept compliant manifests at the CI stage within the evaluated dataset.	RQ1
FR-04	The system shall reject non-compliant manifests at the Kubernetes admission stage, preventing resource creation in the cluster.	RQ2, RQ3
FR-05	The system shall accept compliant manifests at the admission stage within the evaluated dataset.	RQ2
FR-06	The system shall report multiple applicable policy violations in a single evaluation pass, with per-container attribution.	RQ1
FR-07	The system shall enforce policies at the admission stage independently of the CI pipeline, blocking resources that bypass CI validation.	RQ3
FR-08	The system shall produce structured, timestamped log output for every evaluation run to support traceability and reproducibility.	Scope
FR-09	The system shall support at least five distinct security policies covering container security context, resource constraints, and container image version control.	Scope
FR-10	The system shall enable a baseline comparison by evaluating the same manifests without policy enforcement.	Scope
FR-11	The evaluation dataset shall contain both compliant and non-compliant Kubernetes manifests representing multiple violation categories and enforcement scenarios.	RQ1, RQ2, RQ3

6.1.2 Non-Functional Requirements

Table 6.2 presents the non-functional requirements considered during the design of the proof of concept.

Table 6.2: Non-Functional Requirements

ID	Requirement	Traceability
NFR-01	Policy evaluation at the CI stage shall introduce limited execution overhead relative to baseline YAML validation (target overhead factor $\leq 1.5\times$).	Scope
NFR-02	The complete implementation shall be executable from a single Git repository clone, given documented prerequisite tools.	Scope
NFR-03	CI enforcement shall run on GitHub Actions (`ubuntu-latest`) without platform-specific dependencies. Local execution shall be supported on Windows (PowerShell 5.1).	Scope
NFR-04	All policies shall be version-controlled alongside the manifests they govern, enabling change-level traceability.	Scope
NFR-05	The policy architecture shall support the addition of new policies without requiring modification of existing policy definitions.	Scope
NFR-06	The proof of concept shall depend only on open-source, freely available tools, including Conftest, OPA, Gatekeeper, kind, and kubectl.	Scope
NFR-07	The same policy intent shall produce the same enforcement outcome at both CI and admission stages for the same manifest input.	RQ2
NFR-08	Admission-control tests shall use `-dry-run=server` to avoid persisting state in the cluster, ensuring repeatable evaluation.	Scope

6.2 Tool Selection Rationale

The architecture integrates Conftest for CI-stage validation and OPA Gatekeeper for Kubernetes admission control. This section justifies those choices against the main alternatives.

6.2.1 CI Validation — Conftest vs. Alternatives

Table 6.3 compares Conftest against three commonly used CI-stage manifest validation tools.

Table 6.3: CI-Stage Manifest Validation Tool Comparison

Criterion	Conftest	Checkov	Datree	Polaris
Policy language	Rego (custom)	Python/YAML	YAML schemas	Go-based rules
Any CI via CLI				
Gatekeeper-compatible policy reuse		×	×	×
Custom rule extensibility	Full Rego	Partial	Limited	Limited
Admission control support	Via Gatekeeper	×	×	Via webhook only

Conftest was selected over Checkov, Datree, and Polaris because it uses OPA/Rego as its evaluation engine — the same engine that underpins Gatekeeper (Kapetanidou, Nizamis, and Votis 2025; Rossi et al. 2025). This allows the same policy files to be executed at both the CI stage and the admission stage without maintaining separate implementations in different languages. Checkov uses a Python-based rule engine and YAML-defined policies that have no semantic equivalence to Rego (Kapetanidou, Nizamis, and Votis 2025). Datree enforces predefined YAML schemas and provides no mechanism for custom policy logic. Polaris uses Go-based rules with no OPA compatibility. None of the three alternative tools offers Gatekeeper-compatible cross-stage reuse, which is the property required by this architecture.

6.2.2 Admission Controller — Gatekeeper vs. Alternatives

Table 6.4 compares OPA Gatekeeper against the two most widely discussed alternatives.

Table 6.4: Kubernetes Admission Controller Comparison

Criterion	**OPA Gatekeeper**	**Kyverno**	**Kubewarden**
Policy language	Rego	YAML / Common Expression Language (CEL)	WebAssembly (any)
CI-stage tool with same language	Conftest	None	None
CNCF graduation status	Graduated	Incubating	Sandbox
Learning curve	High (Rego)	Low (YAML)	High (WebAssembly)
Cross-stage policy reuse	Native	×Requires separate impl.	×Requires separate impl.

Kyverno was considered but rejected because it uses a YAML/CEL-based policy language for which there is no equivalent CI-stage validation tool (Kapetanidou, Nizamis, and Votis 2025). Using Kyverno for admission control would require maintaining a separate set of policy definitions for CI validation — exactly the enforcement fragmentation this architecture is designed to avoid. Kubewarden was excluded due to its CNCF Sandbox maturity status and the additional complexity introduced by WebAssembly compilation for policy authoring; both factors increase adoption risk relative to the scope of this proof of concept. Gatekeeper, as a CNCF Graduated project (Rossi et al. 2025), offers the strongest maturity guarantee and the only native compatibility with the Conftest toolchain.

6.3 Architecture Overview

The proposed architecture introduces a multi-tier Policy-as-Code enforcement approach combining CI validation and Kubernetes admission control enforcement.

The proposed architecture overview can be seen on Figure 6.1.

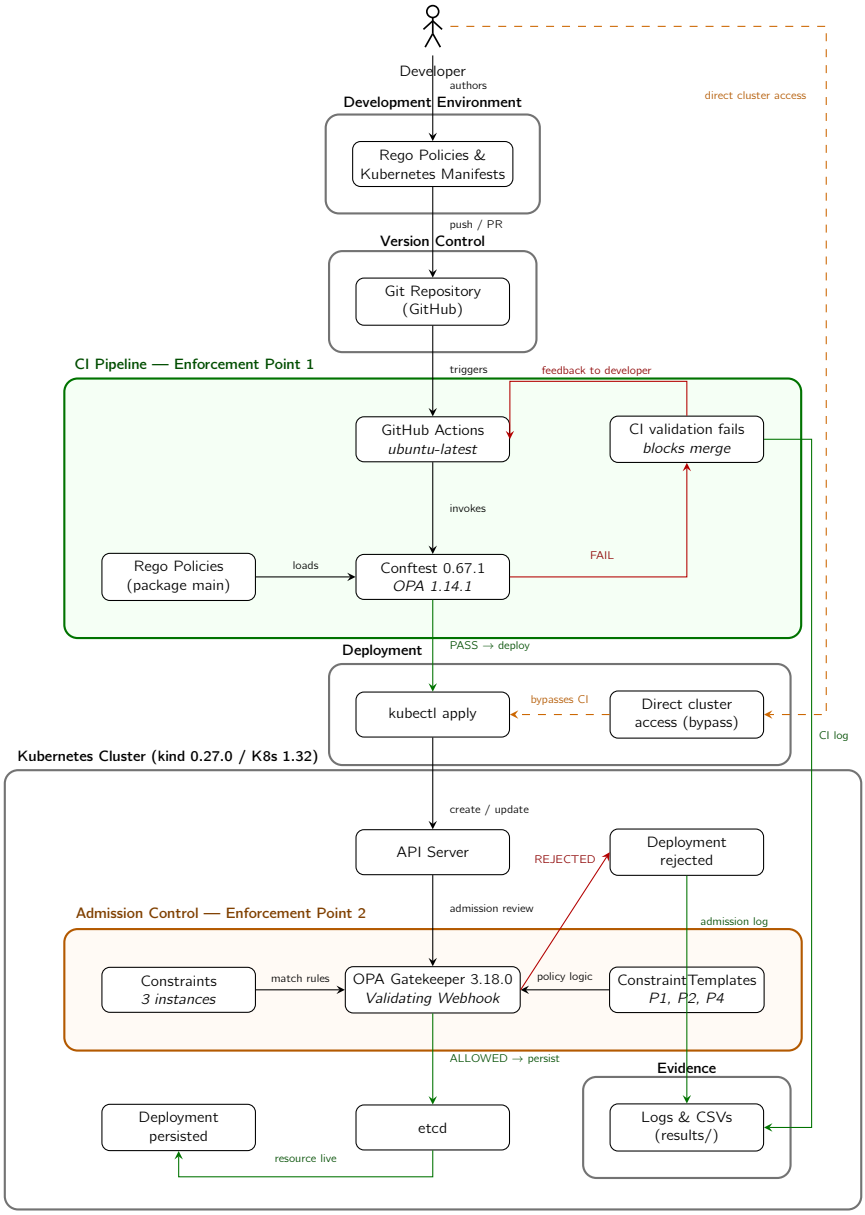

Figure 6.1: Multi-Stage Policy-as-Code Enforcement Architecture

The flow begins when a developer submits Kubernetes manifests to a GitHub repository. This GitHub repository contains GitHub Actions workflow jobs that handle automated CI execution and validation.

As part of the CI process, the Kubernetes manifests are evaluated using Conftest and Rego policies, allowing it to detect insecure configurations even before the deployment attempt takes place. In case of policy violations, the pipeline will fail, and respective violations messages will be delivered to the developer.

The CI validation layer allows the shift-left security paradigm through governance enforcement at an early stage within the software delivery process. However, CI validation alone cannot provide complete governance enforcement because deployments could bypass CI pipelines in favor of direct communication with the Kubernetes cluster.

For this purpose, an additional enforcement layer based on Kubernetes admission control with the usage of OPA Gatekeeper is introduced. OPA Gatekeeper is a validating admission webhook plugin integrated with Kubernetes API server.

In the process of deployment request processing, Gatekeeper runs the manifest validations against pre-defined ConstraintTemplates and Constraints, which are responsible for implementing Policy-as-Code rules. Non-conformant manifests are blocked prior to Kubernetes resources creation, hence offering obligatory enforcement on the deployment step.

The proposed architecture includes two main layers of enforcement:

- **CI Validation Layer** — responsible for early detection and fast developer feedback.
- **Admission Control Layer** — responsible for mandatory deployment-time enforcement and CI bypass mitigation.

By incorporating two separate enforcement layers, governance coverage can be maximized. While CI validation helps detect insecure configurations early, the admission control layer ensures no deployment of non-compliant manifests into the cluster takes place.

It should be emphasized that the architecture uses only open-source tools to facilitate the development and operation of Kubernetes-based DevSecOps pipelines. The specific technologies employed in the proposed solution include:

- **GitHub Actions** for CI/CD workflow orchestration;
- **Conftest** for static manifest validation;
- **Rego** as the Policy-as-Code language;
- **Kubernetes** as the deployment platform;
- **OPA Gatekeeper** for admission control enforcement.

The architecture does not incorporate run-time behavioral monitoring or dynamic workload analysis as it falls outside the scope of the current proof of concept.

6.4 Policy Model

The PoC uses a collection of Policy-as-Code statements designed to detect and prevent any instances of insecure Kubernetes configurations that usually give rise to misconfigurations. These policies were selected according to the following criteria:

- relevance to Kubernetes security and governance practices;
- suitability for declarative validation using Rego policies;
- compatibility with both CI-based validation and Kubernetes admission control enforcement.

For this reason, these policies focus on configuration-oriented security measures, namely those associated with granting the execution privileges, guaranteeing the reproduction of deployments, and controlling resources. Despite the fact that the PoC seeks not to provide comprehensive coverage of Kubernetes security policies, the chosen set was specifically chosen to include different types of enforcement and deployment risks.

The policy model combines two enforcement stages:

- **CI Enforcement** using Conftest and Rego policies executed during GitHub Actions workflows;
- **Admission Enforcement** using OPA Gatekeeper ConstraintTemplates and Constraints integrated with the Kubernetes admission control mechanism.

Some of the policies were not enforced during both stages of implementation. Some policies that needed to be enforced were enforced during admission for proving blocking during deployment time and bypass of CI. The whole set of policies was enforced during CI validation.

Table 6.5 summarizes the implemented policies and their enforcement stages.

Table 6.5: Policy Summary and Enforcement Stages

Policy	**CI Enforcement**	**Admission Enforcement**	**Primary Objective**
No Root Execution	Yes	Yes	Restrict privileged execution
No Privilege Escalation	Yes	Yes	Prevent privilege abuse
No Latest Tag	Yes	Yes	Improve deployment reproducibility
Resource Constraints	Yes	No	Prevent resource exhaustion
No Privileged Containers	Yes	No	Restrict privileged workloads

6.4.1 No Root Execution Policy

The *No Root Execution* policy requires containers to explicitly define `runAsNonRoot: true` within the Kubernetes security context configuration.

Rationale

When containers run with the root user, the vulnerability scope of Kubernetes workloads increases, and there is potential for privilege abuse in case a container becomes vulnerable. Using root privileges for running containers is not compliant with the least privilege principle and is considered a frequent misconfiguration practice in Kubernetes.

This policy seeks to implement enforcement of non-root execution as part of deployment validation processes.

Security Impact

The policy reduces any threats that are associated with the execution of privileged containers and minimizes the impact of an attack involving container escape or privilege escalation. The policy promotes workload isolation and improves the security baseline of the container.

Enforcement Stage

The policy is enforced at both stages of the proposed architecture:

- **CI Enforcement** using Rego validation rules executed through Conftest;
- **Admission Enforcement** using Gatekeeper ConstraintTemplates and Constraints.

At the admission stage, non-compliant manifests are rejected before resource creation occurs within the Kubernetes cluster.

6.4.2 No Privilege Escalation Policy

The *No Privilege Escalation* policy requires containers to explicitly define `allowPrivilegeEscalation: false` within the security context configuration.

Rationale

Privilege escalation methods allow containerized applications to escalate their privileges while running. For Kubernetes platforms, the presence of unlimited privilege escalation increases the chances of lateral movements and compromises of containers.

The rule was put into practice to ensure that workloads prevent privilege escalation.

Security Impact

This policy will reduce the likelihood of unauthorized privilege escalation in running containers and improve workload isolation. As a result, the effect of an application process being exploited is minimized, as well as the attack surface available for exploitation by malicious workloads.

Enforcement Stage

The policy is enforced during:

- CI validation using Conftest and Rego;
- Kubernetes admission control using Gatekeeper.

The admission enforcement layer ensures that manifests violating the policy cannot be deployed directly into the cluster, even when CI validation is bypassed.

6.4.3 No Latest Tag Policy

The *No Latest Tag* policy prevents the use of mutable container image tags such as `:latest`.

Rationale

Mutable image tags negatively affect deployable reproducibility and configurational traceability. Inconsistency during deployment may occur due to the usage of the `:latest` image tag because its content could be modified without any change in its manifest.

This policy is created with the intention of requiring explicit versioning of images and increasing deployment determinism.

Security Impact

Limiting the mutability of image tags improves traceability within the supply chain and facilitates consistency in deployments. The policy ensures reproducibility of the infrastructure and reduces operational risks due to uncontrolled image changes.

Enforcement Stage

The policy is enforced during:

- CI validation using Rego policies executed through Conftest;
- admission control enforcement through Gatekeeper.

Non-compliant deployment requests are rejected at the Kubernetes API level during admission validation.

6.4.4 Resource Constraints Policy

The *Resource Constraints* policy validates the presence of Kubernetes CPU and memory requests and limits within workload manifests.

Rationale

Kubernetes workloads without any specified resource requirements may result in resource starvation, scheduling problems, and overall unpredictability in the cluster's behaviour. Explicitly specifying the resource requests and limits has been considered best practice in governance for the operation of the Kubernetes clusters in multi-tenancy.

This policy was implemented in order to ensure that workloads behave predictably.

Security Impact

Despite its operational nature, the policy indirectly helps improve the resilience of the platform by reducing the chances of abusing resources and excessive workload consumption.

Enforcement Stage

The enforcement of this policy is done only during the CI validation phase through Conftest and Rego.

It should be noted that the policy was intentionally omitted from the admission policy enforcement process to maintain a small scope of admission policies.

6.4.5 No Privileged Containers Policy

The *No Privileged Containers* policy prevents workloads from enabling Kubernetes privileged container mode.

Rationale

Privileged containers provide enhanced access to host resources, increasing the consequences of a breach on the container. The execution in a privileged manner reduces the separation

between workloads and introduces greater security threats to the Kubernetes node. This policy is designed to prevent such a scenario.

Security Impact

It mitigates the potential dangers related to host access and avoids threats related to container escape and privilege escalation.

Enforcement Stage

This policy is enforced during the CI verification phase using Rego policies through Conftest.

Following the same principle as the Resource Constraints policy, this policy was limited only to CI verification due to its admission control nature in the proof-of-concept.

The selected policy set was designed specifically to include different types of governance policies such as privilege management, reproducible deployment, and operational stability. The inclusion of various types of governance policies allows the proof-of-concept to evaluate the effectiveness of the Policy-as-Code concept in enforcing different aspects of security and governance.

6.5 CI Enforcement Layer

The first level of enforcement that the suggested architecture consists of is integrated into the CI flow. The layer performs an automatic Policy-as-Code validation before deployment is initiated in order to be able to detect insecure Kubernetes configurations as early as possible.

The enforcement system in CI uses GitHub Actions, Conftest, and Rego policies to perform Kubernetes manifests validation upon submission to the repository.

6.5.1 GitHub Actions Integration

GitHub Actions was selected for orchestrating CI operations because of its inherent capability to work with GitHub repositories, its widespread use among DevOps practices, and its ability to operate within automated validation pipelines.

The workflow is invoked whenever changes are made to Kubernetes manifest files or policies through commit or pull request operations on the repository. Once the workflow is executed, it establishes a run-time environment based on Ubuntu and performs policy validation.

The CI pipeline performs the following functions:

1. repository checkout;
2. Conftest installation;
3. loading of Rego policy definitions;
4. evaluation of Kubernetes manifests;
5. generation of validation results;
6. pipeline success or failure reporting.

The workflow was deliberately designed to be up-to-date and replicable. The validation code runs purely within the context of the CI environment without having to invoke any external tools or integrations.

6.5.2 Conftest Validation Engine

Conftest acts as the primary validation tool that will run in the CI environment during policy evaluation. It performs the declarative validation of configuration using Open Policy Agent (OPA) Rego policies.

In the suggested architecture, Conftest will validate Kubernetes manifest files before deploying them in the cloud by running a set of policy rules against them. The entire process will be automated within the CI pipeline.

During execution, Conftest performs the following operations:

- loads Kubernetes manifest files;
- loads Rego policy definitions;
- evaluates manifest fields against policy conditions;
- produces validation results and policy violation messages.

This allows identification of several violations at once in one manifest. This behaviour is especially important in terms of governance assessment, where there are several infringements present in a workload.

Conftest was selected because it provides a simple API that is customized for testing configurations and can be easily combined with GitHub Actions workflows. It also works with reproducible command-line instructions.

6.5.3 Rego Policy Definitions

Regulatory aspects of the CI enforcement layer include Rego as the Policy-as-Code language that checks Kubernetes manifests for compliance. In essence, Rego policies contain the conditions that define whether the given Kubernetes manifests comply or not.

Policies were written for each specific Kubernetes configuration field that corresponded to some security or governance aspect. These policies are built according to the deny-rule semantic approach, all policy violations result in clear rejection notifications.

The implemented Rego policies evaluate the following conditions:

- enforcement of non-root container execution;
- prevention of privilege escalation;
- restriction of privileged container mode;
- validation of CPU and memory resource constraints;
- restriction of mutable container image tags.

The policies operate over Kubernetes Deployment manifests and inspect declarative workload definitions contained within the YAML specification.

The use of Rego provides several advantages within the proposed architecture:

- declarative policy definition;
- separation between policy logic and application code;
- reusable validation logic;
- compatibility with both Conftest and Gatekeeper.

This compatibility was particularly important because it enabled partial policy reuse between the CI validation stage and the Kubernetes admission-control stage.

6.5.4 Validation Flow

Figure 6.2 presents the high-level validation flow implemented within the CI enforcement layer.

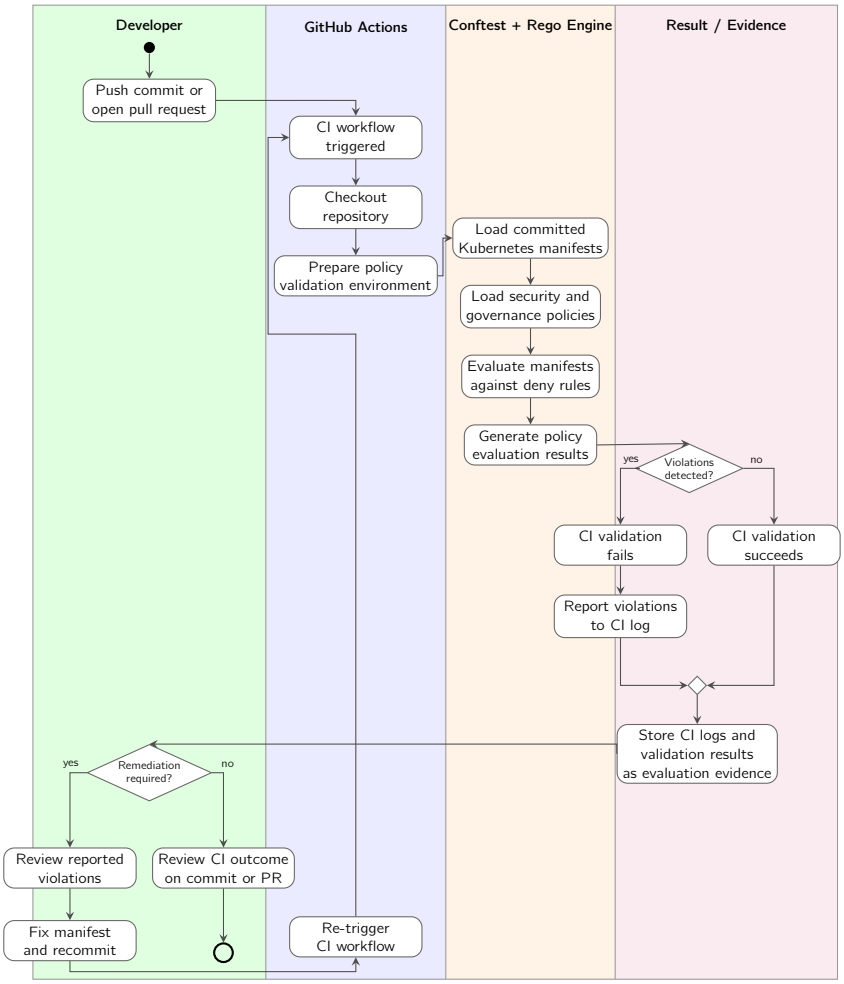

Figure 6.2: CI Validation Flow

Validation begins when a developer commits or raises a pull request, which triggers automatic activation of the CI workflow responsible for fetching committed Kubernetes manifests along with their associated Policy-as-Code definitions from the repository.

Following that, the enforcement layer for CI ensures that a policy evaluation is done through creation of an environment for validating and assessing the conformance of the policies

chosen. For instance, Kubernetes manifests will be evaluated on the basis of security and governance policies related to workload privileges, resource limitations, and container image versions.

In case the validation is successful, the workflow will complete without any errors, which means that all Kubernetes manifests are compliant to defined policies.

However, if there are one or multiple violations detected during validation, the workflow will produce validation failure reports, which will highlight violations of specified policies along with the container configurations that were affected.

The CI workflow will fail at this step, revealing validation results directly in the GitHub Actions interface, where developers will find violations reported with specific details on the violated policies.

At the same time, the validation workflow provides support for an iterative approach. Once developers review reported violations, they can update the affected manifests and re-submit the modified Kubernetes manifest files using additional commits or pull request updates, which again, is consistent with the shift-left security approach.

Finally, after completing validation checks, the CI workflow also stores collected validation logs as evidence artifacts used for supporting traceability, reproducibility, and experimental evaluation activities presented in other chapters of this dissertation.

6.5.5 Pipeline Failure Behavior

CI Enforcement layer was intentionally designed in such a way that it fails independently whenever there is a case of policy violations. This would ensure that any non-conforming manifests would not get beyond the stage of being validated.

Upon occurrence of violations, the process generates well-formatted output in terms of:

- violated policy identifiers;
- affected manifest files;
- container attribution information;
- human-readable violation messages.

The independent failure allows instant feedback in case of policy violations by exposing them in CI process execution logs. The strategy follows shift-left security concept and therefore ensures early discovery of governance issues during the software delivery cycle.

However, CI enforcement layer alone cannot guarantee mandatory deployments of Kubernetes resources. The resource deployment in the cluster can happen independently using techniques that bypass the CI process altogether.

It was for this reason that another layer of enforcement based on Kubernetes Admission Control was added. The mechanism is discussed in the next section.

6.6 Admission Control Layer

Although CI-based policy validation provides early detection of insecure configurations, it does not guarantee mandatory enforcement at deployment time. Kubernetes resources may

still be applied directly to the cluster through mechanisms that bypass the CI workflow entirely. For this reason, the proposed architecture includes a second enforcement stage based on Kubernetes admission control.

The admission-control layer was implemented using Open Policy Agent (OPA) Gatekeeper integrated with the Kubernetes admission webhook mechanism. The purpose of this layer is to enforce mandatory deployment policies at the time of workload creation within the cluster.

6.6.1 OPA Gatekeeper

OPA Gatekeeper uses Kubernetes admission controllers to enforce policies using the OPA and Rego engines. Gatekeeper functions as a validating admission webhook that is linked with the Kubernetes API server.

Whenever there is a request to deploy anything in the cluster, the Kubernetes API server sends an admission review request to Gatekeeper. The admission review request will be checked by Gatekeeper according to the policy constraints specified before creating or modifying any resources.

Within the proposed proof of concept, Gatekeeper was responsible for enforcing a subset of the implemented Policy-as-Code rules directly at the cluster boundary. This enforcement stage represents the authoritative governance layer of the architecture.

The admission-control layer was intentionally limited to three policies:

- non-root container execution;
- privilege escalation restriction;
- mutable container image tag restriction.

These policies were chosen as they define deployment scenarios that can be enforced using runtime constraints.

6.6.2 ConstraintTemplates

Gatekeeper policies are created with the help of custom resources in Kubernetes named ConstraintTemplates. The ConstraintTemplates contain reusable policy logic defined in Rego.

Each ConstraintTemplate defines:

- the policy evaluation logic;
- the custom constraint schema;
- the Kubernetes resource types to which the policy may be applied.

As part of the architectural approach described above, different ConstraintTemplates have been created for admission policies. These constraint templates assess Kubernetes Deployments based on the configuration of security policies on the container level.

The benefit of using ConstraintTemplates is that policy separation and modularity can be achieved.

6.6.3 Constraints

Constraints are concrete policy enforcement rules that arise from ConstraintTemplates. Whereas ConstraintTemplates provide reusable policy logic, Constraints outline the context and policy enforcement for Kubernetes.

Each constraint specifies the following:

- the target Kubernetes resource scope;
- the enforcement action;
- the policy matching conditions.

All developed constraints in the proposed proof of concept enforced policies using a deny approach. This meant that deployment requests violating the configured policies would be denied at admission review time.

The use of both ConstraintTemplates and Constraints allows for reusable policy logic with flexible enforcement.

6.6.4 Webhook Validation Process

Figure 6.3 presents the high-level admission-control validation process implemented within the Kubernetes cluster.

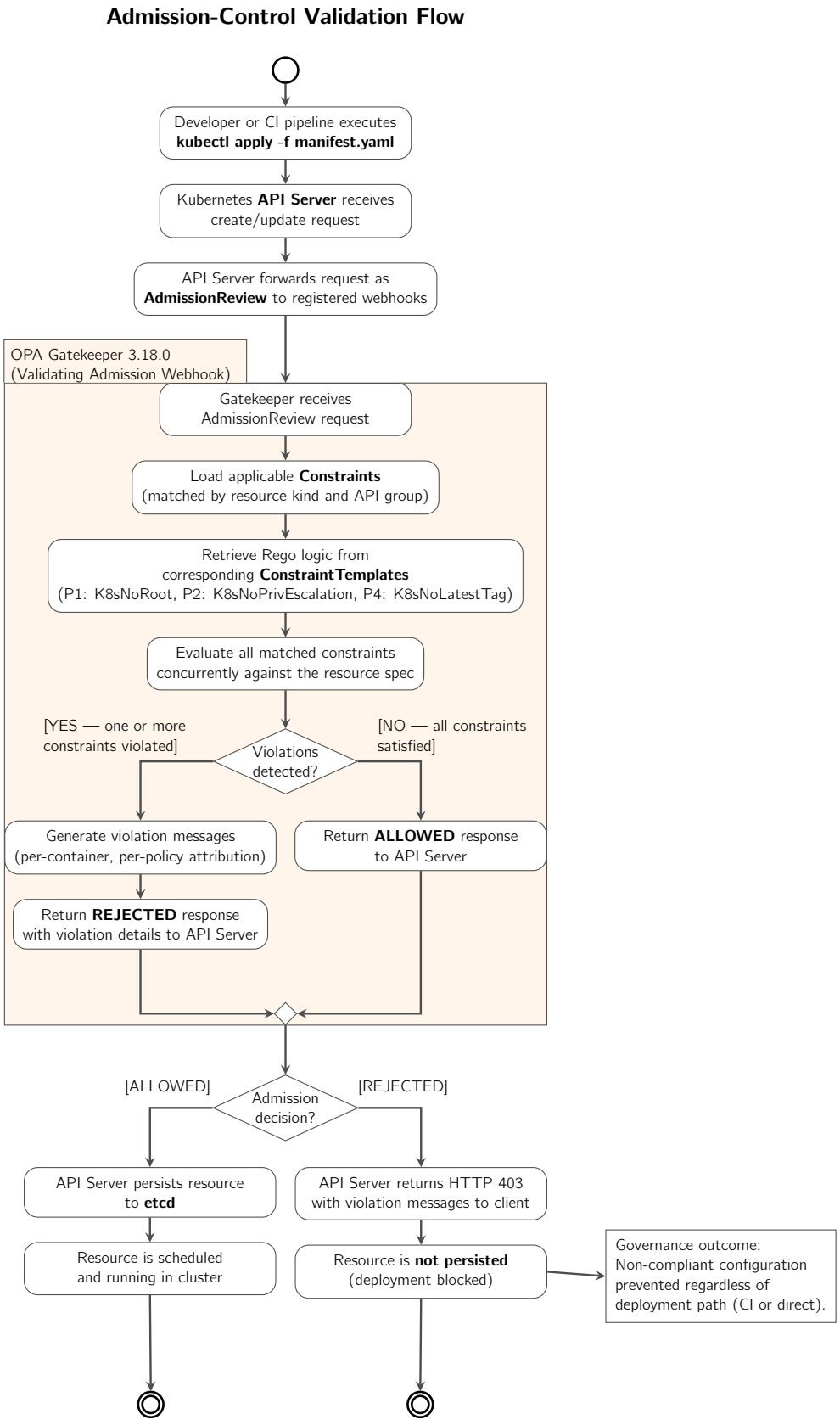

Figure 6.3: Admission Control Validation Flow

The admission-control flow starts when a request for deployment is sent to the Kubernetes cluster using `kubectl apply`. Once the request gets to the Kubernetes API Server, the admission control process is triggered even before the resource gets persisted.

The review request is then processed by the Gatekeeper validating webhook to validate the resource according to the specified Constraints and Constraint Templates. This step involves the evaluation of the Policies-as-Code rules in relation to the Kubernetes resource.

In case there are no violations, Gatekeeper will allow the Kubernetes API Server to persist the resource in the cluster's state.

However, if violations are detected, Gatekeeper will return the Kubernetes API Server with a refusal and messages explaining the violated policies. The Kubernetes API Server will then refuse to process the deployment request.

It can be observed that such behaviour enforces mandatory governance irrespective of the CI process.

6.6.5 Deployment Blocking Behavior

The admission-control layer was intentionally designed to provide authoritative deployment enforcement. Unlike the CI enforcement layer, which primarily provides early validation feedback, admission control operates directly at the Kubernetes cluster boundary.

Therefore, any workload violating admission control policies will not be able to be deployed regardless of whether or not the CI validation phase is skipped entirely.

This deployment-blocking capability is particularly important in scenarios involving:

- direct cluster access;
- manual deployment operations;
- misconfigured delivery workflows;
- incomplete CI enforcement integration.

The admission-control layer therefore acts as the final governance enforcement point within the proposed architecture.

6.7 Multi-Stage Enforcement Model

The proposed proof of concept adopts a multi-stage Policy-as-Code enforcement model that combines CI-based validation with Kubernetes admission control. This layered approach was designed to provide both early detection and mandatory deployment enforcement within Kubernetes-oriented DevSecOps workflows.

The architecture deliberately separates enforcement responsibilities across two complementary governance stages:

- CI-based policy validation using GitHub Actions, Conftest, and Rego;
- deployment-time admission enforcement using OPA Gatekeeper.

Together, these stages form the core governance model evaluated in this dissertation.

6.7.1 CI-Based Early Detection

The first enforcement stage operates during the software integration process. By evaluating Kubernetes manifests before any deployment attempt, CI validation enables early identification of insecure or non-compliant configurations — catching problems at a point where they are cheapest to fix.

This stage primarily supports:

- rapid developer feedback;
- shift-left security practices;
- early governance validation;
- configuration quality improvement.

Detecting violations at the pull request or commit stage reduces the likelihood of insecure manifests progressing further through the delivery pipeline. That said, CI validation alone cannot guarantee that no insecure resource ever reaches the cluster — manifests can still be deployed through mechanisms that bypass the CI workflow entirely.

6.7.2 Admission-Based Mandatory Enforcement

The second enforcement stage operates directly at the Kubernetes cluster boundary through admission control. Unlike CI validation, which can be circumvented, admission enforcement evaluates every deployment request immediately before the resource is persisted — making it the authoritative line of defence regardless of how a deployment was initiated.

This stage primarily supports:

- mandatory policy enforcement;
- deployment-time governance validation;
- CI bypass mitigation;
- cluster-level security protection.

Because enforcement occurs within the Kubernetes control plane itself, workloads violating critical security policies are automatically rejected before deployment completes — whether the CI workflow ran or not.

6.7.3 Complementary Protection Model

Rather than treating CI validation and admission control as alternatives, the proposed architecture deliberately combines them as complementary mechanisms. The CI layer provides early visibility and developer-facing feedback; the admission-control layer provides a hard enforcement boundary that cannot be sidestepped. Together, they establish layered governance protection across multiple points of the software delivery lifecycle.

This combination addresses different categories of operational risk:

- configuration errors caught during active development;
- insecure manifests submitted through standard CI workflows;
- direct deployment attempts that bypass CI entirely;

- inconsistent governance practices across teams or environments.

Figure 6.4 illustrates the relationship between the two enforcement stages, including the CI bypass path that motivates the admission-control layer.

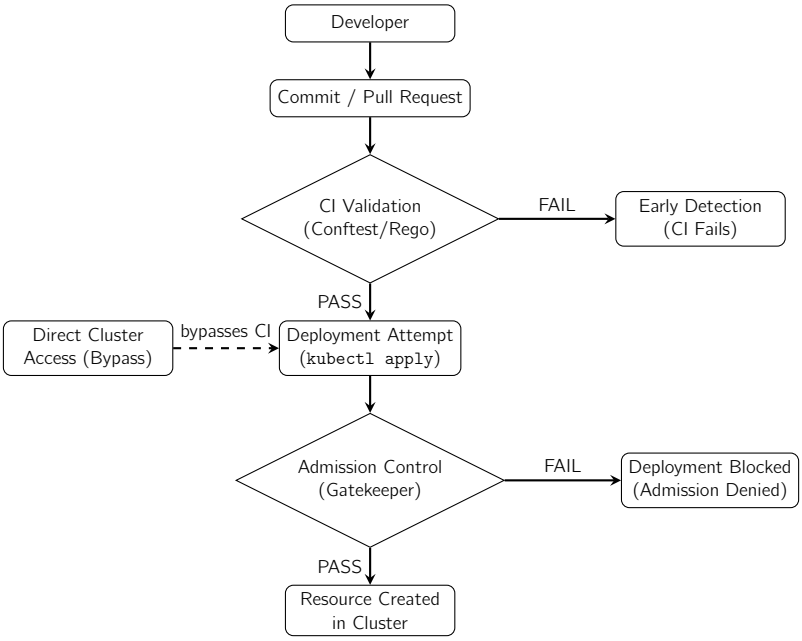

Figure 6.4: Multi-stage enforcement model showing CI validation, the CI bypass path, and Gatekeeper admission control as the final enforcement boundary

The multi-stage enforcement model is the central architectural argument of this dissertation. Rather than relying exclusively on preventive CI validation or deployment-time enforcement in isolation, combining both mechanisms produces a more consistent governance posture and materially reduces the likelihood of insecure Kubernetes resources reaching a running cluster.

6.8 Limitations and Scope

The proof of concept was scoped deliberately to keep the evaluation focused on Policy-as-Code enforcement mechanisms within Kubernetes-oriented DevSecOps workflows. While the implementation successfully demonstrates multi-stage governance enforcement, several limitations must be acknowledged before drawing broader conclusions.

6.8.1 Synthetic Evaluation Dataset

The evaluation dataset consists of synthetic Kubernetes manifests created specifically for controlled experimental validation. It covers compliant manifests, single-violation manifests, multi-violation manifests, negative-control scenarios, and CI-bypass scenarios — each constructed to exercise a known policy condition.

As a result, the dataset is policy-aware by design. The evaluation therefore demonstrates policy correctness and enforcement consistency rather than the ability to generalise across arbitrary real-world Kubernetes configurations.

Although the dataset spans multiple workload types and violation combinations, it does not claim comprehensive detection capability over the full diversity of production Kubernetes manifests. In particular, the evaluation excludes malformed YAML documents, adversarial configuration patterns, and repositories sourced from live production environments. The results should be read with these boundaries in mind.

6.8.2 Partial Gatekeeper Policy Enforcement

The admission-control layer enforces only a subset of the policies implemented at the CI stage. Of the five security rules validated by Conftest, only three were carried through to Gatekeeper:

- non-root execution;
- privilege escalation restriction;
- mutable image tag restriction.

This was a deliberate trade-off to keep the implementation feasible within the scope and time-frame of the proof of concept while still providing a meaningful demonstration of mandatory deployment-time enforcement. The admission-control layer should therefore be understood as a targeted demonstration rather than a full mirror of the CI policy set.

6.8.3 Deployment-Focused Admission Enforcement

The admission-control implementation is scoped exclusively to Kubernetes workload deployment validation — specifically Deployment, Job, CronJob, and StatefulSet resources relevant to containerised workload execution.

Resource categories such as networking policies, persistent storage configuration, RBAC definitions, and cluster-wide administrative resources fall outside the evaluated scope. The Gatekeeper Constraints were also designed for admission-time validation of create and update operations only, with no attempt to address broader cluster governance. This focus is intentional: it keeps the evaluation centred on the relationship between Policy-as-Code enforcement and Kubernetes workload governance within DevSecOps pipelines.

6.8.4 Absence of Runtime Protection

Policy validation occurs exclusively at CI execution time and during Kubernetes admission review. No runtime protection or continuous workload monitoring was implemented.

Consequently, the proof of concept does not address:

- runtime behavioural analysis;
- post-deployment policy drift;
- container compromise detection;
- anomaly detection;
- runtime intrusion prevention.

Once a workload clears admission control and is running in the cluster, the proposed solution takes no further enforcement action. It is best understood as a preventive governance mechanism rather than a complete runtime security platform.

6.8.5 Evaluation Scope Boundaries

The implementation is a proof of concept — designed to evaluate the feasibility and behaviour of multi-stage Policy-as-Code enforcement, not to deliver production-ready enterprise governance coverage. The evaluation therefore focuses on:

- policy enforcement correctness;
- governance layering behaviour;
- CI bypass mitigation;
- operational enforcement feasibility;
- enforcement overhead characteristics.

Topics such as large-scale production deployment, multi-cluster governance federation, policy lifecycle management, and organisational compliance integration are explicitly out of scope. These boundaries define the context within which the presented results should be interpreted.

Chapter 7

Proof of Concept Implementation

This chapter describes the implementation of the proof of concept (PoC) for multi-stage Policy-as-Code enforcement within Kubernetes-oriented DevSecOps pipelines. Where the previous chapter focused on architectural design and the governance model, this chapter focuses on the practical details: how the environment was built, how the tooling fits together, and what configuration decisions were made to support the experimental evaluation.

The implementation integrates GitHub Actions, Conftest, Rego policies, Kubernetes, and OPA Gatekeeper to provide automated policy validation across CI and admission-control stages. Throughout, the emphasis is on clarity and reproducibility — every component is open-source, and the full environment can be recreated from the repository alone.

The complete proof-of-concept implementation, including policy definitions, evaluation manifests, Gatekeeper configurations, CI workflow definitions, automation scripts, and experimental artefacts, is available in a public GitHub repository to support reproducibility and independent verification of the results: `https://github.com/aliceresende/dimei-pac-devsecops-pipeline-k8s`.

7.1 Development Environment

The proof of concept was built using a hybrid local and cloud-based environment: a Windows workstation for development and local testing, GitHub-hosted runners for CI execution, and a local Kubernetes cluster for admission-control validation.

This setup was chosen to support:

- reproducible CI validation workflows;
- local Kubernetes admission-control testing;
- isolated policy evaluation scenarios;
- repeatable experimental execution.

7.1.1 Local Development Workstation

Implementation and local testing were performed on a Windows workstation using Visual Studio Code. PowerShell 5.1 handled local automation tasks, including:

- local Conftest execution;
- Gatekeeper validation testing;

- Kubernetes resource evaluation;
- automated scenario execution;
- evidence and log generation.

The Windows environment reflects the primary development platform used throughout the project. However, all CI validation workflows ran on Linux-based GitHub Actions runners to maintain compatibility with standard cloud-native CI/CD execution environments.

7.1.2 GitHub Actions Environment

CI validation was implemented using GitHub Actions workflows on GitHub-hosted Ubuntu runners. The CI environment was responsible for:

- automated workflow execution;
- Conftest-based policy validation;
- policy violation reporting;
- generation of CI execution evidence.

Workflows ran on the `ubuntu-latest` runner image, ensuring consistent validation behaviour independent of the local development platform. The CI environment pulled directly from the GitHub repository, which contained:

- Kubernetes manifests;
- Rego policy definitions;
- evaluation scripts;
- Gatekeeper configuration resources.

7.1.3 Kubernetes Cluster Environment

The admission-control layer was implemented and tested on a local Kubernetes cluster provisioned with `kind` (Kubernetes in Docker). The choice of `kind` offered several practical advantages for this type of proof of concept:

- lightweight and fast cluster provisioning;
- reproducible Kubernetes environments;
- full compatibility with Gatekeeper admission control;
- isolated experimental execution.

The cluster was used exclusively for admission-control validation and deployment enforcement scenarios, hosting:

- the Kubernetes API server;
- OPA Gatekeeper components;
- ConstraintTemplates and Constraints;
- admission-control evaluation resources.

A single-node control-plane configuration was sufficient for the experimental objectives of the proof of concept. Table 7.1 summarises the cluster configuration.

Table 7.1: Kubernetes Environment Configuration

Component	**Description**
Kubernetes Distribution	kind (Kubernetes in Docker)
Cluster Type	Single-node control-plane cluster
Admission Controller	OPA Gatekeeper
Deployment Validation	Kubernetes Validating Webhooks
Execution Environment	Local Docker-based cluster

7.1.4 Software Components and Tooling

The proof of concept integrates several open-source technologies covering CI validation, Policy-as-Code enforcement, and Kubernetes admission control. Table 7.2 summarises the primary tools used throughout the implementation.

Table 7.2: Software Components Used in the Proof of Concept

Component	**Version**	**Category**	**Purpose**
GitHub Actions	ubuntu-latest	CI/CD Platform	Automated workflow execution
Conftest	0.67.1	Policy Validation Engine	CI-based manifest validation
OPA/Rego	1.14.1	Policy-as-Code Framework	Declarative policy definition
OPA Gatekeeper	3.18.0	Admission Controller	Kubernetes admission enforcement
kind	0.27.0	Kubernetes Distribution	Local cluster provisioning
Kubernetes	1.32	Orchestration Platform	Deployment target
kubectl	1.32	Kubernetes CLI	Cluster interaction and testing
PowerShell	5.1	Automation Environment	Local scripting and evaluation
Visual Studio Code	Latest	Development Environment	Policy and manifest development

The tooling stack was deliberately limited to open-source technologies to maximise reproducibility and avoid external dependencies.

7.1.5 Implementation Characteristics

This proof of concept is a controlled experimental environment rather than a production-ready deployment platform. Several implementation decisions were simplified to stay focused on the research objectives:

- limited admission-control policy scope;
- synthetic evaluation dataset;
- single-cluster deployment model;
- absence of runtime monitoring;
- locally controlled execution environment.

These constraints keep the focus squarely on evaluating the behaviour and effectiveness of multi-stage Policy-as-Code enforcement, rather than on operational production scalability. The following sections describe the repository structure, Rego policies, CI workflows, and admission-control configuration in detail.

7.2 Repository Structure

The proof of concept lives in a structured Git repository containing all the artefacts needed for Policy-as-Code validation, Kubernetes admission enforcement, evaluation execution, and experimental reproducibility.

The repository separates policy definitions, Kubernetes manifests, Gatekeeper configuration resources, evaluation scripts, and supporting documentation into distinct directories. This organisation improves maintainability and makes it easier to understand where each piece of the implementation fits.

Figure 7.1 presents a simplified view of the repository layout.

```
repository/

 policies/
 tests/
 gatekeeper/
 scripts/
 docs/
 results/
 .github/workflows/
```

Figure 7.1: Simplified repository structure of the proof of concept

7.2.1 Policy Definitions

The `policies/` directory contains the Rego Policy-as-Code definitions used during CI validation through Conftest. Each file implements a single governance rule associated with a Kubernetes security or operational best practice, covering:

- non-root container execution;
- privilege escalation restriction;

- privileged container prevention;
- resource constraint validation;
- mutable image tag restriction.

Keeping each policy in an independent file makes the rules easier to reason about and extend without touching unrelated logic.

7.2.2 Evaluation Manifests

The `tests/` directory contains the Kubernetes manifests used during policy evaluation. Manifests are grouped by scenario category:

- compliant manifests;
- single-violation manifests;
- multi-violation manifests;
- negative-control manifests;
- CI-bypass evaluation scenarios.

Compliant and non-compliant manifests are stored separately to simplify automated validation execution and result interpretation. All manifests are synthetic Kubernetes Deployment workloads constructed to exercise the implemented policies.

7.2.3 Gatekeeper Configuration

The `gatekeeper/` directory contains the Kubernetes resources needed for admission-control enforcement, including:

- ConstraintTemplates;
- Constraints;
- Gatekeeper deployment configuration;
- admission-control validation resources.

Separating these resources from the CI policy definitions makes the boundary between cluster-level enforcement and CI validation artefacts explicit, and allows each layer to be modified independently.

7.2.4 Automation Scripts

The `scripts/` directory contains PowerShell scripts for local validation execution, Gatekeeper testing, and evidence generation. These scripts automate repetitive evaluation tasks including:

- Conftest validation execution;
- Gatekeeper deployment testing;
- automated scenario execution;
- result logging and evidence collection.

Automating these tasks reduces manual effort during evaluation and helps ensure each run produces consistent, comparable results.

7.2.5 Documentation and Results

The `docs/` directory holds implementation documentation and supporting material for the proof of concept. Generated execution evidence — validation logs, evaluation outputs, screenshots, and overhead measurements — is stored in the `results/` directory, kept separate from the implementation artefacts to preserve a clean distinction between what was built and what was observed.

7.2.6 CI Workflow Configuration

The `.github/workflows/` directory contains the GitHub Actions workflow definitions that drive CI-based Policy-as-Code enforcement. These workflow files automate Conftest execution, policy validation, failure handling, and result reporting.

Taken together, the repository structure reflects a clear separation of concerns across:

- policy definitions;
- evaluation datasets;
- admission-control resources;
- automation logic;
- generated evidence.

This organisation makes the proof of concept easier to navigate, extend, and reproduce.

7.3 Rego Policies

Policy-as-Code validation in this proof of concept is implemented using Rego, the declarative policy language provided by Open Policy Agent (OPA). Rego was selected because it is natively compatible with both Conftest and Gatekeeper, which means the same policy logic can be used across CI validation and Kubernetes admission-control enforcement without duplication.

The implemented policies evaluate Kubernetes workload manifests and identify configuration patterns associated with insecure or non-compliant deployment behaviour.

7.3.1 Policy Structure

Each policy is implemented as an independent Rego module using deny-rule semantics. Under this model, a violation causes the policy to emit an explicit rejection message describing the problem — making it straightforward to trace exactly which container triggered which rule.

The policies primarily inspect fields within Kubernetes Deployment specifications:

- `securityContext`;
- container image definitions;
- resource requests and limits;

- privilege-related configuration fields.

A simplified example of the Rego structure is presented in Listing 7.1.

All policies use **OPA v1/Rego v1 syntax**, which requires `import rego.v1` and uses `deny contains msg if {...}` rather than the legacy `deny[msg] {...}` form. This migration was required by Conftest 0.67.1 (OPA 1.14.1), which enforces OPA v1 grammar by default. The updated syntax also enables compatibility with the Gatekeeper ConstraintTemplate wrapper, which uses the same evaluation engine.

```
deny[msg] {
    container := input.spec.template.spec.containers[_]
    not container.securityContext.runAsNonRoot

    msg := sprintf(
        "Container '%s' must set runAsNonRoot to true",
        [container.name]
    )
}
```

Listing 7.1: Simplified Rego deny-rule structure

The deny-rule approach produces explicit, human-readable violation messages while keeping the evaluation logic simple enough to work consistently across both enforcement layers.

7.3.2 No Root Execution Policy

This policy verifies that every container in a workload explicitly sets `runAsNonRoot: true` in its security context. If the field is absent or set to any other value, the manifest is considered non-compliant. Violation messages are generated per container, making it easy to pinpoint exactly which containers in a multi-container pod need attention.

Listing 7.2 shows a simplified excerpt of the policy logic.

```
container := input.spec.template.spec.containers[_]

not container.securityContext.runAsNonRoot

msg := sprintf(
    "Container '%s' must set runAsNonRoot to true",
    [container.name]
)
```

Listing 7.2: Simplified non-root execution policy logic

7.3.3 Privilege Escalation Restriction Policy

This policy checks that the `allowPrivilegeEscalation` field is explicitly set to `false` for every container. Explicit boolean validation was chosen to avoid accepting configurations that omit the field entirely, which would otherwise produce an ambiguous result.

```
container := input.spec.template.spec.containers[_]

container.securityContext.allowPrivilegeEscalation != false

msg := sprintf(
    "Container '%s' must disable privilege escalation",
```

```
    [container.name]
)
```

Listing 7.3: Simplified privilege escalation policy logic

7.3.4 Mutable Image Tag Restriction Policy

Using the `:latest` tag means a deployment may silently pull a different image on each run, making behaviour difficult to reproduce and audit. This policy detects any container image ending in `:latest` and rejects the workload.

```
endswith(container.image, ":latest")

msg := sprintf(
    "Container '%s' uses mutable image tag",
    [container.name]
)
```

Listing 7.4: Simplified mutable image tag policy logic

7.3.5 Resource Constraint Policy

Unlike the security-oriented rules above, this policy addresses operational governance. It validates that CPU and memory requests and limits are present in every container specification. Workloads that omit resource constraints are considered non-compliant, since unconstrained containers can starve neighbouring workloads of cluster resources.

7.3.6 Privileged Container Restriction Policy

This policy checks whether any container attempts to enable Kubernetes privileged execution mode via the `privileged` field in its security context. Privileged containers have broad access to the host kernel, making them a significant attack surface; the policy ensures such workloads are rejected before they can be scheduled.

7.3.7 Policy Reuse Across Enforcement Stages

A notable characteristic of the proof of concept is that the same Rego logic underpins both enforcement stages. Conftest consumes the policy files directly during CI validation; Gatekeeper wraps the same logic inside a ConstraintTemplate for admission-control enforcement. Although the wrapping introduces some Kubernetes-specific boilerplate, the core semantics remain consistent between layers.

This reuse has a practical benefit: when a policy rule is updated, both the CI and admission-control layers reflect the change without requiring separate, independently maintained copies of the logic.

The implemented policies prioritise clarity and reproducibility over complexity. The goal is to demonstrate multi-stage enforcement behaviour, not to produce an exhaustive Kubernetes governance ruleset.

7.4 GitHub Actions Workflow

The CI enforcement layer was implemented as a GitHub Actions workflow integrated directly with the repository containing the Kubernetes manifests and Policy-as-Code definitions. The workflow runs Conftest-based validation automatically on every repository update, embedding policy enforcement into the standard development process rather than treating it as a separate manual activity.

7.4.1 Workflow Trigger Configuration

The workflow fires automatically on:

- direct commits and push operations;
- pull request updates.

This means that any modification to a manifest or policy file triggers validation, giving developers immediate feedback without any manual steps.

A simplified trigger configuration is shown in Listing 7.5.

```
on:
  push:
  pull_request:
```

Listing 7.5: Simplified GitHub Actions workflow trigger configuration

7.4.2 Validation Workflow Execution

Once triggered, the workflow initialises a GitHub-hosted Ubuntu runner and executes the following stages in sequence:

1. repository checkout;
2. Conftest installation;
3. loading of Kubernetes manifests and Rego policy definitions;
4. policy evaluation;
5. validation result generation.

Compliant and non-compliant manifest directories are evaluated separately to support controlled experimental validation. A simplified excerpt of the validation step is shown in Listing 7.6.

```
- name: Validate manifests
  run: |
    conftest test tests/good --policy policies
    conftest test tests/bad --policy policies
```

Listing 7.6: Simplified Conftest validation workflow step

Violations appear directly in the GitHub Actions execution log, giving developers a clear view of which manifests failed and why.

7.4.3 Validation Result Processing

For each detected violation, the workflow reports:

- the affected manifest;
- the violated policy condition;
- the affected container;
- a human-readable validation message.

Multiple simultaneous violations within the same manifest are all reported in a single evaluation pass, which is particularly important for the multi-violation scenarios in the proof of concept dataset. The generated logs also serve as experimental evidence for the evaluation presented later in this dissertation.

7.4.4 Failure Handling Behaviour

When Conftest detects non-compliant manifests, it exits with a non-zero status code, causing the GitHub Actions workflow to fail. This immediate failure signal prevents insecure manifests from passing the CI gate silently.

However, a failed CI workflow does not prevent someone with cluster access from deploying a non-compliant resource directly. This gap motivates the admission-control layer described in the following section.

7.4.5 Workflow Evidence Generation

Beyond its enforcement role, the CI workflow also acts as an evidence-generation mechanism for the experimental evaluation. Workflow execution logs capture successful validation results, policy violation outputs, multi-violation detection behaviour, and timing information — all of which were collected and preserved as part of the evaluation dataset.

7.5 Gatekeeper Implementation

The admission-control layer was implemented using Open Policy Agent (OPA) Gatekeeper integrated with the Kubernetes admission webhook mechanism. Gatekeeper allows Policy-as-Code enforcement to be applied directly within the Kubernetes control plane using Rego-based definitions, making it a natural fit for this proof of concept.

The implementation concentrates on mandatory deployment-time enforcement of a selected set of security-critical policies.

7.5.1 Gatekeeper Deployment

Gatekeeper was deployed into the local `kind` cluster. The deployment includes the controller manager, audit components, and the validating admission webhook that integrates with the Kubernetes API server. All Gatekeeper components run in the `gatekeeper-system` namespace.

Once deployed, Gatekeeper intercepts every resource creation and update request before it is persisted in the cluster — forming the hard enforcement boundary that CI validation alone cannot provide.

7.5.2 ConstraintTemplates

Admission-control policies are implemented as Gatekeeper ConstraintTemplates, which encapsulate reusable Rego policy logic and define the schema for enforcement. Separate templates were created for:

- non-root execution validation;
- privilege escalation restriction;
- mutable image tag restriction.

Each template inspects container-level security configuration within Kubernetes Deployment manifests. Listing 7.7 shows a simplified ConstraintTemplate definition.

```
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: k8snoroot
spec:
  crd:
    spec:
      names:
        kind: K8sNoRoot
```

Listing 7.7: Simplified Gatekeeper ConstraintTemplate structure

The template structure separates reusable policy logic from environment-specific enforcement configuration, keeping each concern modular and independently maintainable.

7.5.3 Constraints

Gatekeeper Constraints instantiate enforcement behaviour from a ConstraintTemplate. While templates define the validation logic, Constraints specify:

- the enforcement scope;
- the resources to target;
- matching conditions.

All Constraints in this proof of concept use deny-based enforcement: any workload that violates a configured policy is rejected before it is created. Listing 7.8 shows a simplified Constraint definition.

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sNoRoot
metadata:
  name: no-root
spec:
  match:
    kinds:
      - apiGroups: ["apps"]
        kinds: ["Deployment"]
```

Listing 7.8: Simplified Gatekeeper Constraint definition

This two-level structure — templates for reusable logic, Constraints for targeted enforcement — makes it straightforward to apply the same policy logic across different resource types or namespaces without duplicating the underlying rules.

7.5.4 Admission-Control Enforcement Flow

When a deployment request arrives at the Kubernetes API server, the API server forwards it to the Gatekeeper validating webhook for policy evaluation. Gatekeeper checks the submitted resource against every applicable Constraint and its associated ConstraintTemplate.

If no violations are found, the webhook returns an approval and the resource is persisted. If any violations are detected, Gatekeeper returns a rejection response with explicit violation messages, and the Kubernetes API server blocks the request before the resource is created.

This flow ensures mandatory governance enforcement at the cluster boundary, independently of whether any CI validation was run beforehand.

7.5.5 Admission-Control Validation Scenarios

The Gatekeeper implementation was evaluated against controlled scenarios covering:

- compliant workloads;
- single-policy violations;
- multiple simultaneous violations;
- CI-bypass deployment attempts.

All scenarios used `kubectl apply -dry-run=server` to trigger authentic admission-review behaviour without leaving persistent changes in the cluster, preserving a clean environment for repeated test runs.

7.5.6 Implementation Scope

The Gatekeeper layer focuses on deployment-time enforcement of selected security-critical policies rather than attempting exhaustive Kubernetes governance coverage. This scope is appropriate for the proof of concept: the goal is to demonstrate that mandatory Policy-as-Code enforcement at the admission layer is feasible and effective, not to implement a complete enterprise governance platform.

7.6 Evaluation Dataset

The proof of concept was evaluated against a controlled dataset of Kubernetes workload manifests created specifically for Policy-as-Code validation and admission-control testing. The dataset was designed to cover:

- CI validation scenarios;
- admission-control enforcement scenarios;
- multi-policy violation detection;
- CI bypass testing;

- controlled false-positive analysis.

In total, the dataset contains 29 Kubernetes manifests, spanning eight resource types (Deployment, StatefulSet, Job, CronJob, ConfigMap, Service, Namespace, and bare Pod) across three origin categories: synthetic manifests purpose-built to trigger or avoid a specific policy; adapted realistic manifests modelled on real workload patterns such as database StatefulSets, migration Jobs, and sidecar Deployments; and realistic manifests representing plausible production resources outside the policy evaluation scope. This provenance taxonomy makes the generalisability boundaries of the evaluation explicit.

7.6.1 Dataset Characteristics

Table 7.3 summarises the dataset composition.

Table 7.3: Evaluation Dataset Categories

Category	**Count**	**Purpose**	**Origin**
Compliant manifests	9	Confirm no false positives across structurally diverse compliant workloads	Mixed (synthetic + adapted realistic)
Negative-control manifests	5	Confirm policies do not over-match on out-of-scope resource types	Mixed (synthetic + realistic)
Single-violation manifests	9	Isolate each policy with exactly one violation; verify per-rule attribution	Synthetic
Multi-violation manifests	6	Test concurrent detection across two to eight simultaneous violations	Mixed (synthetic + adapted realistic)
Total	**29**		

The manifests cover the full range of conditions targeted by the implemented policies:

- root container execution;
- privilege escalation;
- privileged container mode;
- missing resource constraints;
- mutable image tags.

The multi-violation manifests combine several of these conditions within a single workload definition, reflecting the reality that real-world Kubernetes misconfigurations often involve more than one governance failure at a time.

7.6.2 Synthetic and Policy-Aware Dataset

The dataset is synthetic and policy-aware: every manifest was created with prior knowledge of the policy conditions being evaluated. This means the results demonstrate enforcement correctness and consistency rather than generalisation to arbitrary production workloads.

The evaluation does not include malformed YAML, adversarial policy evasion techniques, or manifests drawn from live production repositories. The results should therefore be read as evidence of correct enforcement behaviour within a controlled scope, not as a claim about detection capability in the wild.

7.6.3 Manifest Reuse Across Evaluation Scenarios

The same manifests were used across multiple evaluation scenarios. A non-compliant manifest might be evaluated through the CI workflow in one scenario and submitted directly to the admission layer in another, allowing direct comparison of enforcement behaviour between the two stages without introducing variation from different input data.

As a result, the total number of evaluation scenarios exceeds the number of unique manifests in the dataset.

7.6.4 Dataset Organisation

Manifests are organised within the repository by evaluation category:

```
tests/

 good/        (compliant manifests + negative controls)
 bad/         (single-violation + multi-violation manifests)
```

Figure 7.2: Evaluation dataset organisation within the repository

Compliant manifests and negative-control resources are stored together under `tests/good/`, since both categories pass all policy checks. Non-compliant manifests — both single-violation and multi-violation — are stored under `tests/bad/`. Scenario categories are distinguished by filename convention rather than directory structure, keeping the layout simple and easy to navigate.

7.7 Scenario Design

The evaluation was structured around a set of controlled scenarios designed to exercise the implemented Policy-as-Code enforcement mechanisms under a range of governance conditions. The scenarios were designed to evaluate:

- compliant workload behaviour;
- isolated policy violations;
- simultaneous policy violations;
- CI bypass conditions;
- policy scoping behaviour.

7.7.1 Compliant Scenarios

Compliant scenarios verify that properly configured workloads pass both CI validation and admission control without generating false violations. These scenarios act as positive controls: if a compliant manifest produces a violation, it indicates a problem with the policy logic rather than the workload.

The compliant manifests include correctly configured security contexts, pinned image tags, and valid resource constraints.

7.7.2 Single-Violation Scenarios

Single-violation scenarios test each policy rule in isolation. Each manifest contains exactly one intentional violation — no more — allowing the evaluation to confirm that the relevant rule fires correctly without interference from other configuration issues.

These scenarios also assess the quality of violation attribution: a well-written policy should identify not just that a violation occurred, but which container caused it and why.

7.7.3 Multi-Violation Scenarios

Multi-violation scenarios evaluate manifests containing several simultaneous policy violations. These are among the most practically relevant scenarios in the dataset, since real-world misconfigurations rarely occur in isolation. A workload might combine root execution with privilege escalation and a mutable image tag, all at once.

These scenarios test whether the policy engine reports all violations in a single evaluation pass, and whether the resulting output remains readable when multiple issues are flagged simultaneously.

7.7.4 CI Bypass Scenarios

CI bypass scenarios are perhaps the most important evaluation condition in the proof of concept. They simulate deployment attempts in which a non-compliant manifest skips the CI workflow entirely and is submitted directly to the Kubernetes API server — the sort of thing that might happen through manual `kubectl` access, an emergency deployment, or an incomplete pipeline integration.

The objective is to determine whether admission control holds firm regardless of CI status. During these scenarios:

- non-compliant manifests bypass CI validation;
- deployment requests are submitted directly to the Kubernetes API server;
- Gatekeeper evaluates the resources during admission review;
- non-compliant workloads are rejected before deployment completes.

These scenarios directly support the evaluation of RQ3 and provide the clearest evidence of the admission-control layer's value.

7.7.5 Negative-Control Scenarios

Negative-control scenarios test policy scoping rather than semantic correctness. The five negative-control resources confirm that the policies do not over-match on inputs outside their intended evaluation scope.

Three resources (ConfigMap, Service, Namespace) have no container specification at all and pass trivially. The bare Pod (`pod-negative-control.yaml`) uses `busybox:latest` with no security context — configurations that would trigger P1, P2, and P4 if the resource were a Deployment. It passes because the policy evaluation path (`input.spec.template.spec.containers[_]`) does not match bare-Pod structure. The CronJob (`cronjob-good.yaml`) also uses `busybox:latest` with no security context and passes because CronJobs nest containers at `spec.jobTemplate.spec.template.spec.containers` — one level deeper than the path the policies evaluate.

Together, these two cases document two distinct scope boundaries: the bare Pod tests the absence of `spec.template`; the CronJob tests an alternative nesting depth. Both limitations are deliberate and traceable: the policies were designed for `spec.template.spec.containers` (Deployment, StatefulSet, Job), and extending them to cover bare Pods and CronJobs would require additional path evaluation in each Rego file. No false positives were observed across all five negative-control resources.

7.7.6 Scenario Reuse and Enforcement Comparison

Several scenarios were run under multiple enforcement conditions to support direct comparison between CI validation and admission-control behaviour. The same non-compliant manifest might be evaluated through Conftest, through Gatekeeper, and through a CI bypass attempt — yielding consistent, comparable evidence across all three paths.

This design supports evaluation of:

- detection consistency across enforcement stages;
- deployment-blocking capability;
- governance coverage under different operational conditions.

Chapter 8

Evaluation and Results

This chapter presents the experimental evaluation of the proposed multi-stage Policy-as-Code enforcement architecture. The evaluation examines CI-based validation and Kubernetes admission-control enforcement when applied to Kubernetes workload manifests within controlled DevSecOps scenarios.

The three research questions guiding the evaluation are:

- **RQ1**: Can Policy-as-Code enforcement detect insecure Kubernetes configurations automatically?
- **RQ2**: Does multi-stage enforcement (CI validation combined with admission control) improve governance coverage compared to single-stage enforcement?
- **RQ3**: Does admission control mitigate CI bypass risks?

The evaluation also examines operational characteristics of the implementation, including execution overhead and enforcement consistency. All results were obtained using the controlled dataset and environment described in Chapter 7. All manifests, policy definitions, enforcement configurations, and evaluation artefacts used during the experiments are available in the project repository.

8.1 Evaluation Methodology

The proof of concept was evaluated using a controlled experimental methodology covering:

- CI validation testing;
- Kubernetes admission-control testing;
- multi-stage enforcement comparison;
- CI bypass evaluation;
- execution overhead measurement.

The methodology prioritises reproducible enforcement behaviour over large-scale performance benchmarking.

8.1.1 Controlled Evaluation Environment

All evaluation activities ran within the reproducible environment described in Chapter 7, combining GitHub Actions CI workflows, Conftest-based policy validation, Gatekeeper admission control, a local `kind` Kubernetes cluster, and PowerShell automation scripts. The synthetic

dataset of 29 manifests was evaluated in a total of 34 scenarios: 29 CI-stage scenarios (one per manifest) plus five admission-stage scenarios that reuse existing manifests under a different execution context (`kubectl apply -dry-run=server`, bypassing the CI pipeline). This distinction between manifests and scenarios is important: the same manifest evaluated at the CI stage and at the admission stage constitutes two distinct scenarios with different enforcement conditions.

8.1.2 CI Validation Testing

CI validation testing measured the ability of Conftest and Rego policies to detect insecure configurations during GitHub Actions workflow execution. Each manifest was classified as either passing or producing a policy violation, and the evaluation verified that:

- compliant manifests were accepted without spurious violations;
- non-compliant manifests produced the expected violation messages;
- manifests containing multiple violations reported all of them in a single pass.

Workflow execution logs were preserved as experimental evidence.

8.1.3 Admission-Control Testing

Admission-control testing evaluated Gatekeeper-based deployment enforcement against compliant, non-compliant, multi-violation, and CI-bypass scenarios. All scenarios used:

```
1 kubectl apply --dry-run=server
```

Listing 8.1: Admission-control evaluation execution approach

This mode triggers genuine admission-review behaviour without leaving persistent changes in the cluster, allowing the same scenarios to be run repeatedly in a clean environment.

8.1.4 Overhead Measurement Methodology

The evaluation also measured the execution overhead introduced by Policy-as-Code validation by comparing baseline execution against CI validation runs with enforcement enabled. Measurements were repeated multiple times to reduce variance. For each group, the evaluation calculated the average execution time, standard deviation, and relative overhead factor.

This analysis is intended to give an indicative sense of the operational cost of enforcement, not to serve as a production-scale performance benchmark.

8.2 RQ1 — Detection Capability

RQ1 asks whether the proposed Policy-as-Code implementation can automatically detect insecure Kubernetes configurations. The evaluation covers isolated violations, simultaneous multi-policy violations, compliant workload acceptance, and overall validation consistency.

8.2.1 Detection Results

The CI validation layer correctly detected the expected policy violations across all evaluated non-compliant manifests. Table 8.1 summarises the results.

Table 8.1: RQ1 Detection Results

Scenario Category	Expected Result	Observed Result
Compliant manifests	Accepted	Accepted
Single-violation manifests	Violation detected	Violation detected
Multi-violation manifests	Multiple violations detected	Multiple violations detected
Negative-control manifests	Accepted	Accepted

The multi-violation scenarios confirmed that the policy engine reports all simultaneous violations within a single evaluation pass — a practically important property, since forcing developers to fix one violation at a time creates unnecessary friction. Manifest D-20, the maximum-violation reference, triggered all five policies simultaneously, producing eight distinct deny messages in one pass.

Across the full dataset, Conftest evaluated 29 manifests against 9 deny rules, producing a total of 261 assertions. Of these, 221 passed and 40 violations were detected. Zero false positives and zero false negatives were observed. Every violation was correctly attributed to the specific container and field that triggered the deny rule, including D-16 where only the second container in a two-container pod violated P4 — confirming that the policy logic correctly iterates over the container list without false attribution to non-violating containers.

Each policy was triggered by at least two independent manifests: P1 by 7 manifests, P2 by 5, P3 by 5, P4 by 8, and P5 by 2. This ensures that no policy's detection capability depends on a single test case.

The compliant set was designed to stress the boundary conditions of the policy logic, not merely to repeat a single passing structure. Four edge cases are particularly relevant to the false-positive claim. D-30 omits the `privileged` field entirely rather than setting it to `false`, confirming that P5 does not treat absence as a violation. D-31 references a container image by SHA-256 digest (`nginx@sha256:...`) rather than by tag, confirming that P4 handles digest-based references without false-positive. D-32 uses a hardened `securityContext` containing additional fields (`readOnlyRootFilesystem`, `capabilities.drop`, `runAsUser`) beyond those evaluated by the policies, confirming that `object.get` is unaffected by extra keys. D-34 pairs a non-compliant init container — using no tag and no security context — with a fully compliant application container, confirming that all five policies scope to `containers[_]` only and do not evaluate `initContainers`.

The 261-assertion result should be interpreted carefully. The dataset was constructed with prior knowledge of the policies: every violation was intentionally placed, and every compliant manifest was intentionally clean. This is not a weakness of the evaluation — it is appropriate for a proof of concept that aims to demonstrate enforcement correctness, not detection capability on unseen inputs. What the result does support is that the Rego deny-rule model, applied consistently across five policies and nine deny rules, produces deterministic and unambiguous output on the evaluated input space.

8.2.2 False-Positive Analysis

The negative-control manifests were accepted cleanly during every validation run. No false positives were observed. That said, this result should be interpreted carefully: the dataset

is synthetic and policy-aware, so it is not a strong test of false-positive behaviour across arbitrary real-world workloads. These results demonstrate enforcement correctness within the controlled evaluation scope, not broad generalisation.

8.3 RQ2 — Governance Coverage

RQ2 asks whether multi-stage enforcement improves governance coverage compared to relying on a single enforcement stage. The evaluation compares CI-only, admission-only, and combined enforcement approaches.

8.3.1 Enforcement Comparison

CI validation provides early governance detection but cannot prevent direct deployment attempts that bypass the pipeline. Admission control provides a hard deployment boundary but offers no developer-facing feedback during manifest development. The combined architecture addresses both gaps.

Table 8.2 summarises the observed capabilities of each approach.

Table 8.2: RQ2 Multi-Stage Enforcement Comparison

Capability	**CI Only**	**Admission Only**	**Combined**
Early violation detection	Yes	No	Yes
Mandatory deployment blocking	No	Yes	Yes
Developer feedback	Yes	Limited	Yes
CI bypass protection	No	Yes	Yes
Multi-stage governance coverage	Partial	Partial	Improved

8.3.2 Governance Layering

The comparison in Table 8.2 reveals a structural property that goes beyond the individual capabilities of each stage: the two enforcement mechanisms are not simply additive — they are complementary in a way that produces a governance guarantee neither can provide alone.

CI validation operates within the developer's workflow. Its authority depends entirely on the pipeline being in the deployment path: if a developer has direct cluster access, or if a new repository has not yet been configured with Conftest, CI validation is simply absent. It provides no protection against what it cannot see. Admission control, by contrast, operates inside the Kubernetes API server itself. Every resource creation or update request passes through it regardless of how it was initiated. Its authority does not depend on developer compliance with the workflow — it is architecturally mandatory.

This distinction has a concrete implication for DevSecOps governance: organisations that rely exclusively on CI validation are accepting a residual risk that scales with the number of people who have direct cluster access. In regulated environments or multi-team contexts where `kubectl` access is not tightly controlled, this risk is not theoretical. The bypass scenarios in Section 8.4 demonstrate exactly this: every non-compliant manifest submitted

directly to the cluster was caught by Gatekeeper, while CI was never involved. CI validation had zero influence over the outcome.

The governance implication of the combined architecture is therefore not simply "two checks are better than one." It is that the two checks cover categorically different failure modes. CI validation addresses the developer experience dimension of governance — catching problems early, providing fast feedback, reducing the cost of remediation. Admission control addresses the enforcement dimension — ensuring that no non-compliant resource can reach the cluster regardless of how the delivery process was followed. A DevSecOps governance model that includes only one of these dimensions is incomplete; a model that includes both achieves defence-in-depth.

This finding aligns with the gap identified in the literature review: Haverinen et al. 2024 and Akbar, Smolander, et al. 2022 both identify single-stage enforcement focus as a structural limitation of current practice, but neither provides empirical evidence of what multi-stage enforcement looks like in practice. The comparison in Table 8.2 provides that evidence for the specific Conftest–Gatekeeper toolchain.

8.4 RQ3 — CI Bypass Mitigation

RQ3 is arguably the most important research question evaluated in this dissertation, because it tests whether the admission-control layer provides genuine mandatory enforcement or merely advisory guidance.

8.4.1 CI Bypass Scenario

The evaluation submitted non-compliant manifests directly to the Kubernetes cluster, bypassing the CI workflow entirely. In practice, this simulates several realistic operational scenarios:

- a developer or operator running `kubectl apply` directly from a local workstation;
- an emergency hotfix pushed to the cluster without waiting for CI completion;
- an automated process with cluster credentials that deploys without triggering a pipeline;
- a new repository or branch where the Conftest validation step has not yet been configured.

In all cases, the manifest reaches the Kubernetes API server without having been evaluated by Conftest. The admission-control evaluation tests whether Gatekeeper's webhook — operating inside the API server's admission chain — independently detects and rejects the same violations that CI would have caught.

Deployment was attempted using:

```
kubectl apply --dry-run=server
```

Listing 8.2: CI bypass evaluation scenario

8.4.2 Admission-Control Enforcement Results

Gatekeeper rejected every non-compliant deployment attempt during the bypass scenarios. The validating webhook intercepted each admission-review request and blocked resource

creation before the manifest could be persisted, returning explicit violation messages in every case.

Table 8.3 summarises the observed outcomes.

Table 8.3: RQ3 CI Bypass Evaluation Results

Scenario	**CI Validation**	**Admission Control**	**Outcome**
Compliant workload	Bypassed	Accepted	Deployment permitted
Non-compliant workload	Bypassed	Rejected	Deployment blocked
Multi-violation workload	Bypassed	Rejected	Deployment blocked

These results confirm that admission control provides a reliable enforcement guarantee regardless of CI status — exactly the property that motivates the multi-stage architecture.

The significance of mandatory enforcement. The bypass scenario is not merely a robustness test. It demonstrates a governance property that is qualitatively different from anything CI validation can offer: *mandatory* enforcement. CI validation is voluntary in the sense that its authority depends on the pipeline being present in the deployment path. Admission control is mandatory in the sense that any resource creation or update request must pass through it regardless of how the request was submitted. This distinction matters because in practice, CI bypass is not a rare edge case — it occurs through direct cluster access, emergency deployments, automated scripts with credentials, and onboarding of new pipelines that have not yet been configured with CI validation. For every one of these scenarios, the admission layer holds firm.

The broader governance implication is *resilience*: the architecture maintains enforcement guarantees even when a prior mechanism fails or is deliberately circumvented. This is the defining property of a defence-in-depth model as opposed to a single-point-of-control model. A governance programme that relies exclusively on CI validation assumes that developers always deploy through the pipeline — an assumption that breaks the moment a production incident triggers an emergency hotfix. A programme that includes admission control does not need that assumption.

However, the bypass protection is selective rather than universal. The Gatekeeper layer enforces only three of the five implemented policies (P1 — non-root execution, P2 — privilege escalation restriction, P4 — mutable image tag). Policies P3 (resource constraints) and P5 (privileged containers) are enforced at the CI stage but not at admission. A workload that violates only P3 or P5 — and bypasses CI — would not be blocked by Gatekeeper. This is a deliberate design choice to keep the admission scope manageable within the proof of concept, but it means the defence-in-depth guarantee is incomplete for those two policies. In a production deployment, extending Gatekeeper to all five policies would close this gap.

The bypass scenario also highlights a distinction that the dry-run testing approach makes necessary: all admission-control tests used `kubectl apply -dry-run=server`, which triggers genuine admission-webhook evaluation but does not persist resources. The enforcement logic is identical to a live deployment — the API server forwards the request to Gatekeeper

and enforces the webhook response — but no resources were created or left in the cluster. This is appropriate for a repeatable evaluation environment, though it means the results reflect intended enforcement behaviour rather than long-running cluster state.

8.5 Baseline and Execution Overhead Analysis

The evaluation compared Policy-as-Code enforcement behaviour against a governance baseline and measured the execution-time cost of Conftest validation.

8.5.1 Enforcement Comparison Against Baseline

The governance baseline reflects common practice in CI/CD environments without explicit Policy-as-Code enforcement: standard `kubectl apply -dry-run=client` execution, which validates Kubernetes API schema compliance but applies no security policies.

Table 8.4: Baseline vs. Policy-as-Code Enforcement Comparison

Enforcement Stage	**Good Manifests (14)**	**Bad Manifests (15)**	**Governance Contribution**
Baseline (`kubectl` schema only)	Accepted (14/14)	Accepted (0/15 blocked)	No security enforcement; all violations pass silently
CI — Conftest (all 5 policies)	Accepted (14/14)	Rejected (15/15 blocked, 40 violations)	Shift-left detection; per-rule attribution
Admission — Gatekeeper (P1, P2, P4)	Accepted (1/1 tested)	Rejected (4/4 tested)	Runtime gate; enforces independently of CI

The baseline result is the most operationally significant finding in this table. Every non-compliant manifest — including those with critical misconfigurations such as root execution, privilege escalation, and mutable image tags — was accepted by Kubernetes without any warning. This confirms that standard API schema validation provides no security governance; all enforcement value comes from the explicitly defined Rego policies.

8.5.2 Execution Overhead Measurements

Execution-time overhead was measured using `System.Diagnostics.Stopwatch` in PowerShell 5.1. Each operation was executed 20 times; the first run was discarded as a warm-up to exclude one-time costs. Statistics were computed over the remaining 19 runs.

Table 8.5: Policy Enforcement Execution Overhead (19 runs after warm-up)

Operation	**Avg (ms)**	**Min (ms)**	**Max (ms)**	**SD (ms)**
M1 — `kubectl -dry-run=client` (baseline, no policy)	56.8	52.4	61.6	1.9
M2 — `conftest test` (all 29 manifests, 9 rules)	56.3	53.7	59.6	1.7

Conftest evaluation of all 29 manifests against 9 deny rules completed in an average of 56.3 ms (SD 1.7 ms), compared to 56.8 ms (SD 1.9 ms) for the baseline — a difference of -0.5 ms, yielding an overhead factor of $1.0\times$. The negative delta and overlapping standard deviations confirm that policy evaluation time is indistinguishable from the baseline within measurement noise, yielding an overhead factor of $1.0\times$ and comfortably satisfying NFR-01 (target $\leq 1.5\times$).

This result carries a specific significance for DevSecOps adoption that goes beyond satisfying a technical requirement. One of the most frequently cited barriers to introducing security controls into CI/CD pipelines is developer and team resistance to any mechanism perceived as slowing down the delivery process (Akbar, Smolander, et al. 2022; Moyón, Angermeir, and Mendez 2024). Even controls that are technically sound are often deprioritised or disabled when they are perceived as adding friction to feedback loops. The overhead measurement directly addresses this concern: Conftest evaluation of all 29 manifests against 9 deny rules adds less than one millisecond to a pipeline step that already takes tens of seconds for repository checkout, dependency installation, and test execution. The governance cost is, in practical terms, zero. This means that the adoption argument for CI-stage PaC validation does not require trading pipeline speed for security — the two are compatible at this scale.

Admission-stage overhead measurements (M3/M4) were not completed within the measurement window; these remain a direction for future evaluation.

8.6 Discussion

The evaluation results demonstrate that the proposed multi-stage architecture successfully implements Policy-as-Code enforcement within Kubernetes-oriented DevSecOps workflows. The more significant question — what these results mean for governance practice — is addressed in this section.

8.6.1 What the Multi-Stage Architecture Adds

The results in Table 8.2 confirm that neither enforcement stage alone covers all governance risks. CI validation blocks all 15 non-compliant manifests but cannot prevent direct cluster access. Gatekeeper blocks bypass attempts but provides no early feedback during development and does not enforce P3 and P5. Only the combined architecture covers both.

The key insight is that the two stages are not redundant — they are complementary in a specific and non-obvious way. CI validation is *preventive*: it catches problems before they reach the cluster, at the cheapest point in the delivery lifecycle to fix them. Admission

control is *authoritative*: it provides a hard boundary that remains active regardless of how the delivery process was followed. The combination creates a property that neither layer alone can provide — defence-in-depth where the failure of one enforcement stage does not eliminate governance protection.

Relation to identified research gaps. The synthesis in Chapter 2 identified fragmented enforcement across pipeline stages as the primary gap in the reviewed literature — the observation that tools operate at isolated points without coordination (Akbar, Smolander, et al. 2022; Haverinen et al. 2024). The multi-stage architecture directly addresses this gap: CI and admission control are not two independent tools applied at separate times, but two coordinated enforcement points that share the same policy definitions and collectively provide coverage that neither provides alone. The second gap — the absence of reproducible empirical evaluation of multi-stage enforcement — is partially addressed by the experimental results in this chapter. The remaining gaps (policy lifecycle management, regulatory mapping, cross-platform CI/CD coverage) are confirmed as open and are discussed in the future work section.

Comparison with related work. Kapetanidou, Nizamis, and Votis 2025 evaluate scanning tools for Kubernetes misconfigurations but focus on single-stage admission-time analysis. Rossi et al. 2025 demonstrate OPA-based admission enforcement for scheduling but do not evaluate CI integration. Vijayaraghavan 2025 advocate for shift-left enforcement at commit time but do not empirically test the interaction between CI and admission stages. The results here extend beyond each of these studies by demonstrating that the two stages can be coordinated under a single policy language and that this coordination produces measurable governance properties — specifically, the bypass-resistance demonstrated in RQ3 — that the individual stages cannot claim independently.

8.6.2 The Policy Reuse Property and Its Significance

Of all the results in this evaluation, the policy reuse property is the most architecturally significant — and the least visible in the results tables. The tables show that policies detect violations and that Gatekeeper blocks bypass attempts. What the tables do not show is that all of this happens from a single source of truth: five Rego files, authored once, evaluated by two independent enforcement engines at two separate points in the delivery lifecycle.

What reuse means in practice. Conftest reads the policy files directly and evaluates them against manifests during CI. Gatekeeper wraps the same Rego logic inside ConstraintTemplates and evaluates it at the Kubernetes API server boundary. The underlying deny-rule semantics — the conditions checked, the messages produced, the containers evaluated — are identical. When a policy changes, the change propagates to both enforcement stages automatically. There is no second implementation to update, no translation layer to maintain, and no risk of the CI check and the admission check diverging over time.

Why divergence is a real risk. In organisations that use separate tools and languages for CI and admission — for example, Checkov at CI and Kyverno at admission — the same governance intent must be expressed twice in two different policy languages with different semantics, different test frameworks, and different maintenance cycles. Over time, policy updates in one layer frequently fail to propagate to the other, particularly when the teams responsible for CI tooling and cluster governance are different. The result is that a configuration blocked by the CI check may pass admission control, or vice versa — a governance inconsistency that is difficult to detect and undermines the audit trail. The

Conftest–Gatekeeper architecture eliminates this class of failure by construction: single-source policy definitions cannot diverge because there is only one definition.

Implications for governance consistency. The reuse property has a direct implication for governance assurance that extends beyond the technical implementation. An organisation that can demonstrate that its CI validation and its admission control evaluate the same policy logic from the same source is in a significantly stronger position from a compliance and auditability perspective than one that maintains two separate policy repositories. The former can point to a single, version-controlled policy definition and demonstrate that it is enforced at every stage of the delivery process. The latter must prove consistency across two independently maintained systems — a much harder audit burden.

Relation to the literature. Few studies in the reviewed literature explicitly evaluate policy reuse across both enforcement stages (Rossi et al. 2025; Vijayaraghavan 2025). The shift-left literature (Haverinen et al. 2024; Vijayaraghavan 2025) emphasises early enforcement but does not address what happens when early enforcement fails or is bypassed, or how to ensure that later enforcement applies the same rules. The admission-control literature (Kapetanidou, Nizamis, and Votis 2025; Rossi et al. 2025) demonstrates cluster-boundary enforcement but treats it as a standalone mechanism. This dissertation demonstrates that the two perspectives are not alternatives but can be unified under a single policy language, and that this unification preserves governance consistency across the full delivery path.

8.6.3 What the Results Do Not Prove

Acknowledging the boundaries of the evidence is as important as reporting the evidence itself. The results in this chapter demonstrate two things clearly: functional correctness (the policies enforce the intended rules without false positives or false negatives on the evaluated dataset) and architectural viability (the Conftest–Gatekeeper combination operates as a coordinated two-stage enforcement mechanism with negligible execution overhead). These are meaningful findings. They are not sufficient, however, to draw conclusions about several questions that practitioners would need to answer before adopting this architecture at scale.

The evaluation does not demonstrate **detection accuracy on unseen inputs**. The dataset was constructed by the same researcher who implemented the policies, creating a known risk of unconscious alignment between test cases and policy logic. The zero false-positive result holds within this controlled scope; it cannot be extrapolated to independently authored manifests or production-sourced workload configurations without further evaluation.

The evaluation does not demonstrate **scalability under production conditions**. The dataset contains 29 manifests evaluated against 5 policies. Real Kubernetes governance baselines typically require 20–50 controls covering network policies, RBAC, secret management, and image provenance. High-frequency deployment pipelines may evaluate hundreds of manifests per hour. Whether the architecture maintains acceptable performance and coverage at that scale is an open empirical question.

The evaluation does not demonstrate **organisational adoption feasibility**. The sub-millisecond tooling overhead confirms that the technical cost of adoption is negligible, but organisational adoption depends predominantly on non-technical factors: team agreement on policy rules, exemption management processes, onboarding of new engineers to the governance model, and the cultural willingness to treat a failed CI check as a governance signal rather than a workflow obstacle. These factors are not addressed here and represent a distinct research programme.

The evaluation does not demonstrate **policy lifecycle management**. The five policies were authored, tested, and deployed in a single-researcher environment with full visibility of the evaluation conditions. In practice, policies must be versioned, reviewed, approved, tested against realistic workloads, and deprecated when requirements change. None of these processes were evaluated.

In summary: the results demonstrate that the proposed architecture is *correct* and *viable*. They do not demonstrate that it is *complete*, *scalable*, or *organisationally effective* in the way that a production governance programme must be. These distinctions should be kept clearly in mind when interpreting the contributions of this work.

8.6.4 Chapter Summary: What These Results Teach

A reader finishing this chapter might conclude that the main result is: five policies were implemented and they detected the violations they were designed to detect. That reading is technically accurate but misses what the evaluation actually demonstrates.

The more important result is this: *the same governance intent, expressed once in Rego, was enforced consistently at two structurally different points in the delivery process — before the cluster and at the cluster boundary — with zero semantic divergence between them*. The 261 assertions and the zero false-positive result are evidence that this works correctly on the evaluated inputs. The overhead measurement is evidence that it works without pipeline cost. The bypass scenarios are evidence that the admission layer maintains governance guarantees independently of whether the CI stage ran at all.

What this collection of results teaches about Policy-as-Code in DevSecOps is not primarily about which violations were detected. It is about governance architecture: that a model in which policies are defined once and enforced everywhere is both technically feasible and operationally viable with mature open-source tooling. This is the property that organisations building on this work should focus on — not the specific five policies, which are a starting baseline, but the architecture that allows any set of governance rules to be applied consistently across the full delivery path without duplication, divergence, or dependency on developer workflow compliance.

8.7 Threats to Validity

Several limitations should be considered when interpreting these results.

8.7.1 Synthetic and Policy-Aware Dataset

The evaluation dataset was created with prior knowledge of the implemented policies. This means it primarily demonstrates enforcement correctness under known conditions, not the ability to generalise to the diversity of real-world Kubernetes workloads.

8.7.2 Limited Policy Scope

Only five policies were implemented at the CI stage, and only three were enforced at the admission layer. The evaluation does not represent comprehensive Kubernetes governance coverage, and the selected policies were chosen to demonstrate enforcement behaviour rather than to cover the full attack surface.

8.7.3 Deployment-Focused Admission Enforcement

The admission-control evaluation is scoped to Kubernetes workload deployment validation. RBAC enforcement, networking policies, storage configuration, and cluster-wide administrative governance are outside the evaluated scope. Results should be interpreted accordingly.

8.7.4 Controlled Experimental Environment

All evaluation activities took place in a local, single-cluster environment using a synthetic dataset. The proof of concept does not evaluate production-scale CI/CD systems, high-volume deployment pipelines, distributed multi-cluster governance, or organisational policy lifecycle management. The results demonstrate implementation feasibility and enforcement behaviour, not enterprise-scale operational performance.

8.7.5 Absence of Runtime Security Evaluation

The proof of concept focuses on preventive configuration governance. Runtime workload protection, behavioural monitoring, and post-deployment security analysis are outside scope. The architecture should be understood as a deployment-time governance enforcement mechanism, not a complete Kubernetes security platform.

8.7.6 Known Policy Scope Limitations

The implemented policies target `spec.template.spec.containers`, the path used by Deployments, StatefulSets, and Jobs. Three resource categories fall outside this scope and are documented by dedicated negative-control scenarios.

Bare Pods (using `spec.containers` directly) do not have a `spec.template` wrapper and are therefore invisible to all five policies. The negative-control dataset includes a bare Pod (`pod-negative-control.yaml`) using `busybox:latest` with no security context — configurations that would violate P1, P2, and P4 if the resource were a Deployment. The Pod passes correctly, documenting the scope boundary. In production environments, bare Pods deployed by operators or system components would not be governed by this policy set.

CronJobs nest containers at `spec.jobTemplate.spec.template.spec.containers` — one level deeper than the path the policies evaluate. The negative-control CronJob (`cronjob-good.yaml`) also uses `busybox:latest` with no security context and passes all policies, documenting a second scope boundary. Extending coverage to CronJobs would require adding `spec.jobTemplate.spec.template.spec.containers` as a second evaluation path in each policy.

Init containers (`spec.template.spec.initContainers`) are not evaluated by the policies, which iterate over `containers[_]` only. Manifest D-34 pairs a non-compliant init container (no tag, no security context) with a compliant application container; the manifest passes all policies correctly, confirming that init containers are out of scope. In practice, a malicious or misconfigured init container could run as root or use a mutable image tag without being detected by the current policy set.

Chapter 9

Conclusion and Future Work

This dissertation presented the design, implementation, and evaluation of a multi-stage Policy-as-Code enforcement architecture for Kubernetes-oriented DevSecOps pipelines.

The proof of concept combined CI-based validation — using GitHub Actions, Conftest, and Rego policies — with Kubernetes admission-control enforcement using OPA Gatekeeper. The goal was to determine whether layered Policy-as-Code enforcement could meaningfully improve governance consistency and deployment security within Kubernetes delivery workflows.

The evaluation addressed three research questions: automated detection of insecure Kubernetes configurations, governance coverage across multiple enforcement stages, and mitigation of CI bypass risks through admission-control enforcement. Across all three, the results support the value of the multi-stage approach — not because either layer is perfect in isolation, but because their combination covers risks that no single enforcement mechanism can address alone.

9.1 Conclusion

The implemented proof of concept successfully demonstrated the feasibility of integrating Policy-as-Code enforcement across both CI validation and Kubernetes admission-control stages.

The implementation achieved the following objectives:

- automated validation of Kubernetes manifests using Rego-based policies;
- enforcement of security and governance controls during CI execution;
- deployment-time admission-control enforcement using Gatekeeper;
- rejection of non-compliant workloads before cluster deployment;
- detection of multiple simultaneous policy violations;
- reproducible experimental evaluation within a controlled environment.

The proof of concept additionally demonstrated that Policy-as-Code enforcement can be integrated into Kubernetes-oriented DevSecOps workflows using open-source technologies without requiring proprietary governance platforms.

The evaluation results indicate that combining CI validation with admission-control enforcement provides stronger governance protection than relying exclusively on a single enforcement stage.

Specifically, the admission-control layer demonstrated the ability to enforce deployment restrictions independently of CI validation status, reducing exposure to risks associated with CI bypass scenarios.

The implemented architecture therefore supports a defence-in-depth governance approach in which:

- CI validation provides early detection and developer feedback;
- admission control provides authoritative deployment enforcement at the Kubernetes cluster boundary.

Although the proof of concept intentionally maintains a limited implementation scope, the evaluation demonstrates the practical feasibility of multi-stage Policy-as-Code enforcement within Kubernetes deployment environments.

9.2 Research Questions Revisited

This section revisits the three research questions defined in Section 3.2, interprets the experimental results in their context, relates them to the reviewed literature, and discusses their implications and limitations.

9.2.1 RQ1 — Detection Capability

RQ1: Can Policy-as-Code enforcement detect insecure Kubernetes configurations automatically?

Observation. Across 261 assertions applied to 29 manifests, the five Rego policies detected all 40 expected violations with zero false positives and zero false negatives. All violations were correctly attributed to the specific policy rule and container that triggered them, including cases where only one container in a multi-container pod was non-compliant.

Interpretation. The result confirms that declarative Rego deny-rules, evaluated by an automated engine at CI time, can replace manual manifest review for the class of configuration errors covered by the five policies. The fact that the per-container attribution is correct even in mixed-compliance pods suggests that the OPA evaluation model handles container iteration correctly without requiring policy authors to write iteration guards explicitly.

Relation to literature. This result is consistent with the findings of Kapetanidou, Nizamis, and Votis 2025, who demonstrate that static analysis tools can detect misconfigurations in the pre-deployment phase, and with Rossi et al. 2025, who report that OPA-based enforcement produces reliable admission decisions. However, the reviewed literature does not provide a comparable end-to-end evaluation of Conftest across a structurally diverse 29-manifest dataset; the baseline established here provides a reference point that prior work lacks.

Implications. For organisations considering PaC adoption, the result suggests that even a small, carefully authored policy set (five rules, nine deny conditions) can provide meaningful

automation coverage at negligible execution overhead. The sub-millisecond CI overhead makes adoption cost-free from a pipeline performance perspective.

Limitations. The zero false-positive and zero false-negative result is bounded by the controlled, policy-aware dataset. The manifest authors had prior knowledge of the rules being tested. Evaluating the same policies against independently authored or production-sourced manifests would produce a different — and more informative — false-positive rate. The result should be read as demonstrating enforcement correctness, not as a claim about detection accuracy on arbitrary inputs.

9.2.2 RQ2 — Governance Coverage

RQ2: Does multi-stage enforcement (CI validation combined with admission control) improve governance coverage compared to single-stage enforcement?

Observation. CI-only enforcement covers all five policies but cannot prevent deployments that bypass the pipeline. Admission-only enforcement covers three of the five policies and blocks all deployment paths, but misses P3 and P5 entirely. The combined architecture inherits both properties: full policy coverage at CI time and deployment-blocking capability at admission time for the three enforced policies.

Interpretation. The complementarity is structural rather than incidental. The two stages operate at different points in the delivery process and are subject to different bypass vectors. CI enforcement depends on the pipeline being in the deployment path; admission enforcement depends on Gatekeeper being installed and its constraints being active. Neither assumption is universally safe in real environments. The value of combining both is precisely that the failure of one assumption does not eliminate governance protection — a defence-in-depth property.

Relation to literature. The finding aligns with the synthesis in Chapter 2: no study in the reviewed corpus evaluates multi-stage enforcement empirically, and several identify single-stage focus as a gap (Haverinen et al. 2024; Akbar, Smolander, et al. 2022). The result provides the first concrete evidence that the two stages are complementary rather than redundant for this specific toolchain.

Implications. The most actionable implication is that teams should not treat CI validation and admission control as alternatives and choose one. Each covers risks the other cannot. For organisations already using Gatekeeper or another admission controller, adding Conftest-based CI validation is low-cost (sub-millisecond overhead) and extends coverage to policies not implemented at admission. For organisations with CI validation only, adding three Gatekeeper constraints for the most security-critical policies (P1, P2, P4) would close the bypass gap for those policies at a defined implementation cost.

Limitations. The governance improvement demonstrated is bounded to the five implemented policies. P3 and P5 remain CI-only in this proof of concept, meaning the defence-in-depth property does not hold for resource limit and privileged-mode violations. Extending admission coverage to all five policies is a straightforward engineering task but was outside the proof-of-concept scope.

9.2.3 RQ3 — CI Bypass Mitigation

RQ3: Does admission control mitigate the risks associated with CI pipeline bypass?

Observation. All four non-compliant manifests applied directly to the cluster via `kubectl apply -dry-run=server` — bypassing the CI pipeline entirely — were rejected by the Gatekeeper admission webhook. The compliant manifest was accepted without false rejection. The multi-violation scenario (S29) confirmed that Gatekeeper evaluates all applicable constraints concurrently, reporting all three violations in a single rejection.

Interpretation. This result is the most operationally significant finding of the dissertation. The bypass scenarios simulate realistic conditions — direct `kubectl apply` from a developer workstation, an emergency hotfix, or a misconfigured new repository — in which CI validation is not in the deployment path. The result demonstrates empirically that Gatekeeper's webhook, operating inside the Kubernetes API server's admission chain, provides enforcement guarantees that are independent of the deployment mechanism. This is the key property that distinguishes a two-stage architecture from a CI-only approach.

The result also reveals a boundary: the mitigation is selective. Violations of P3 and P5 — policies without Gatekeeper ConstraintTemplates — would not be blocked on the bypass path. This is a documented scope decision, not a fundamental constraint of the pattern.

Relation to literature. The bypass threat is identified but rarely quantified in the reviewed literature. Vijayaraghavan 2025 identifies CI bypass as a risk in GitOps environments; Rossi et al. 2025 demonstrates Gatekeeper's scheduling enforcement but does not test bypass scenarios. The bypass scenario results in this dissertation provide empirical evidence for a claim that previous work has asserted conceptually but not tested.

Implications. For security architects, this result supports the use of admission control as a last line of defence that does not depend on developers following the correct deployment path. It also motivates extending Gatekeeper coverage to P3 and P5 in any production deployment of this architecture, since the bypass gap for those two policies remains open in the current proof of concept.

9.3 Industrial Applicability

The proof-of-concept results have direct implications for organisations considering Policy-as-Code adoption at scale, though these implications must be interpreted conservatively given the controlled evaluation scope.

Enterprise adoption context. The architecture demonstrated here — Conftest for CI, Gatekeeper for admission control, Rego as the shared policy language — uses tools that are production-grade and widely adopted. Gatekeeper is a Cloud Native Computing Foundation (CNCF) Graduated project used in production by major cloud providers; Conftest is maintained as a CNCF Sandbox project with active enterprise adoption. The technical barrier to adopting this architecture is therefore low: the tooling is freely available, well-documented, and does not require proprietary platforms.

Policy lifecycle management. One challenge not addressed by the proof of concept is policy lifecycle management at scale. In this evaluation, five policies were authored, tested, and deployed by a single researcher with full knowledge of the evaluation conditions. In an enterprise environment, policies must be authored by security teams, reviewed and approved by compliance teams, tested against realistic manifests, versioned alongside the infrastructure they govern, and deprecated when requirements change. None of these processes are automated or supported by the current implementation. This represents the most significant gap between the proof-of-concept scope and enterprise readiness.

Multi-team and multi-cluster governance. The architecture was evaluated in a single-cluster, single-repository environment. Real enterprise deployments typically involve multiple teams, each with their own repositories and pipelines, deploying to multiple clusters across different environments (development, staging, production). Scaling this architecture to that context would require a centralised policy distribution mechanism — for example, a policy OCI registry — and a mechanism for applying different policy sets to different clusters or namespaces based on environment classification. These are solvable engineering problems, but they were outside the scope of this investigation.

Developer friction and organisational adoption. The execution overhead measurement ($1.0\times$ factor, -0.5 ms delta) suggests that the tooling cost of adding Conftest validation is operationally negligible. However, governance programmes frequently fail not because of tooling costs but because of developer friction: failed pipelines that block deployments, unclear violation messages, or exemption processes that are slow or non-existent. The proof of concept produces human-readable violation messages with per-container attribution, which addresses the clarity concern, but does not evaluate how developers respond to policy failures in practice. Organisational adoption research would be needed to assess this dimension.

What the results suggest for practitioners. The proof of concept provides evidence that the Conftest–Gatekeeper combination is technically viable, operationally lightweight, and produces deterministic enforcement outcomes. For organisations at an early stage of PaC adoption, the five policies implemented here (non-root execution, privilege escalation restriction, mutable image tag restriction, resource constraints, privileged container restriction) represent a sensible starting baseline aligned with the CIS Kubernetes Benchmark. The architecture can be adopted incrementally: starting with CI validation only, adding Gatekeeper for the most critical policies, and extending coverage as the policy programme matures.

9.4 Limitations

A clear account of what was and was not demonstrated is essential for correctly scoping the contributions of this dissertation.

What was demonstrated:

- That five Rego deny-rules, evaluated by Conftest, correctly detect all known violations in a purpose-built dataset of 29 Kubernetes manifests with zero false positives and zero false negatives;
- That Gatekeeper correctly blocks direct deployment attempts for the three policies implemented as ConstraintTemplates, independently of the CI pipeline;
- That the execution overhead of Conftest validation (56.3 ms average) is statistically indistinguishable from a baseline without policy evaluation (56.8 ms);
- That a single Rego policy definition can serve both CI validation and admission enforcement without duplication.

What was not demonstrated:

- Detection accuracy on independently authored or production-sourced manifests — the dataset was constructed with prior knowledge of the policies, creating a risk of unconscious alignment between test cases and policy logic;

- False-positive behaviour on the full diversity of Kubernetes configurations, including CronJobs, DaemonSets, bare Pods, and init containers, which fall outside the policy evaluation path;
- Scalability of the enforcement mechanism to large policy sets, high manifest volumes, or concurrent deployment pipelines;
- Policy lifecycle management, including authoring, review, versioning, testing, and deprecation processes;
- Organisational adoption dynamics, developer friction, or compliance integration in real enterprise environments;
- Admission-stage overhead (M3/M4 measurements were not completed), leaving the latency cost of Gatekeeper webhook evaluation uncharacterised;
- Defence-in-depth guarantees for P3 (resource constraints) and P5 (privileged mode), which remain CI-only due to the absence of corresponding ConstraintTemplates.

External validity. The results should not be generalised beyond Kubernetes workload governance using the OPA/Rego ecosystem and the specific tools evaluated. Different policy languages (Kyverno, Polaris), different CI platforms (GitLab CI, Jenkins), or different Kubernetes distributions may produce different enforcement behaviours and overhead characteristics. Generalisation requires separate evaluation in those contexts.

9.5 Future Work

Several directions for future work emerge from the results and limitations of the proposed proof of concept.

One possible extension involves the integration of runtime security enforcement mechanisms capable of monitoring workload behaviour after deployment. This would extend the proposed architecture beyond preventive governance and enable continuous runtime protection.

Future research could additionally evaluate the proposed approach using larger and more diverse Kubernetes manifest datasets, including production-derived repositories and real-world deployment configurations.

Another relevant direction involves enterprise-scale evaluation within:

- high-volume CI/CD environments;
- multi-team governance models;
- distributed Kubernetes infrastructures;
- large-scale policy libraries.

The implementation could also be extended toward multi-cluster governance architectures supporting centralised policy management across multiple Kubernetes environments.

Additional future work may include:

- automated policy generation;
- policy recommendation systems;
- policy severity classification;

- policy lifecycle management;
- integration with supply-chain security frameworks;
- runtime drift detection mechanisms.

Finally, future research could evaluate the operational impact of Policy-as-Code governance on developer productivity, workflow usability, and organisational compliance processes within production DevSecOps environments.

Although the proposed proof of concept intentionally maintains a limited experimental scope, the evaluation results suggest that multi-stage Policy-as-Code enforcement represents a viable and promising approach for improving Kubernetes deployment governance within DevSecOps workflows.

9.6 Reproducibility and Dissemination

To facilitate future research and replication, all artefacts developed during this work have been made available in a public repository: `https://github.com/aliceresende/dimei-pac-devsecops-pipeline-k8s`. This includes the complete set of Rego policy definitions, the evaluation dataset of 29 Kubernetes manifests, the Gatekeeper ConstraintTemplates and Constraints, the GitHub Actions workflow configuration, the PowerShell automation scripts used for evidence generation and overhead measurement, and the experimental logs and screenshots referenced throughout Chapters 7 and 8.

The results of this dissertation have been further developed and submitted for publication in a peer-reviewed scientific journal, extending the investigation of multi-stage Policy-as-Code enforcement in DevSecOps environments.

Bibliography

Akbar, Muhammad Azeem and Abeer Abdulaziz Alsanad (2025). "Empirical Investigation of Key Enablers for Secure DevOps Practices". In: *Information and Software Technology*.

Akbar, Muhammad Azeem, Kari Smolander, et al. (2022). "Toward Successful DevSecOps in Software Development Organizations: A Decision-Making Framework". In: *Information and Software Technology*.

Alnafessah, Ahmad et al. (2021). "Quality-Aware DevOps Research: Where Do We Stand?" In: *IEEE Software*.

Alonso, Juncal, Radoslaw Piliszek, and Matija Cankar (2023). "Embracing IaC Through the DevSecOps Philosophy: Concepts, Challenges, and a Reference Framework". In: *Software: Practice and Experience*.

Berardinelli, Luca et al. (2025). "Model-Driven Engineering, AI, and DevOps: A Systematic Mapping Study". In: *Information and Software Technology*.

Billawa, Priyanka et al. (2022). "SoK: Security of Microservice Applications: A Practitioners' Perspective on Challenges and Best Practices". In: *ACM Computing Surveys*.

Delaitre, Sabine and José Maria Pulgar Gutiérrez (2024). "Vulnerability Detection Tool in Source Code by Leveraging Semantic Code Graphs". In: *SoftwareX*.

Dinh, Nghia et al. (2025). "Enhancing Smart Contract Security Through DevSecOps". In: *Computers & Security*.

Gama, Diego et al. (2024). "Supporting Continuous Vulnerability Compliance Through Automated Identity Provisioning". In: *IEEE Transactions on Software Engineering*.

Haverinen, Henry et al. (2024). "Automating Cybersecurity Compliance in DevSecOps with Open Information Model for Security as Code". In: *Computers & Security*.

Hevner, Alan R. et al. (2004). "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1, pp. 75–105.

Jansen, Matthijs et al. (2025). "Columbo: A Reasoning Framework for Kubernetes Configuration Space". In: *Proceedings of the VLDB Endowment*.

Kapetanidou, Ioanna Angeliki, Alexandros Nizamis, and Konstantinos Votis (2025). "An Evaluation of Commonly Used Kubernetes Security Scanning Tools". In: *Computers & Security*.

Kosińska, Joanna and Krzysztof Zieliński (2023). "Experimental Evaluation of Rule-Based Autonomic Computing Management Framework for Cloud-Native Applications". In: *Future Generation Computer Systems*.

Mittal, Akshay (2025). "AI-Augmented DevSecOps Pipelines for Secure and Scalable Cloud-Native Systems". In: *Future Generation Computer Systems*.

Mittal, Akshay and Vivek Venkatesan (2025). "Practical Integration of Large Language Models into Enterprise CI/CD Pipelines for Security Policy Validation". In: *IEEE Software*.

Moyón, Fabiola, Florian Angermeir, and Daniel Mendez (2024). "Industrial Challenges in Secure Continuous Development". In: *Empirical Software Engineering*.

Nagasundari, S. et al. (2025). "Extensive Review of Threat Models for DevSecOps". In: *ACM Computing Surveys*.

Port, Dan, Bill Taber, and Parisa Emkani (2024). "Investigating Effectiveness and Compliance to DevOps Policies". In: *Journal of Systems and Software*.

Prates, Luís and Rúben Pereira (2025). "DevSecOps Practices and Tools". In: *Software Quality Journal*.

Ramaj, Xhesika (2022). "A DevSecOps-Enabled Framework for Risk Management of Critical Infrastructures". In: *Computers & Security*.

Ramaj, Xhesika et al. (2024). "On DevSecOps and Risk Management in Critical Infrastructures". In: *Computers & Security*.

Romeo, Francesco et al. (2025). "ARPaCCino: An Agentic-RAG for Policy as Code Compliance". In: *IEEE Software*.

Rossi, Matthew et al. (2025). "Policy-Driven Security-Aware Scheduling in Kubernetes". In: *Proceedings of the IEEE Conference*.

Sánchez-Gordón, Mary and Ricardo Colomo-Palacios (2020). "Security as Culture: A Systematic Literature Review of DevSecOps". In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW)*, pp. 266–269. doi: 10.1145/3387940.3392233.

Sedrakyan, Gayane, Maria Eugenia Iacob, and Jos Hillegersberg (2024). "Towards LowDevSecOps Framework for Low-Code Development". In: *Journal of Systems and Software*.

Vakhula, Oleksandr and Ivan Opirskyy (2025). "AI-Driven Security as Code Using Multi-Agent Systems". In: *Journal of Systems and Software*.

Vakhula, Oleksandr, Ivan Opirskyy, et al. (2025). "Research on Policy-as-Code for Role-Based and Attribute-Based Access Control". In: *Computers & Security*.

Vijayaraghavan, Sarathe Krisshnan Jutoo (2025). "Policy as Code: A Paradigm Shift in Infrastructure Security and Governance". In: *IEEE Security & Privacy*.