Catch Me If You Can: Hunting Misconfigurations Before They Break Prod

Marcel Punselie & Jagadeesh Devaraj - ING

SREcon25 EMEA, Dublin

do your thing

Product Area Lead - Reliability by Design

Global Engineering & Reliability

Marcel Punselie has over 25 years of experience in IT, and his career has always been driven by a passion for reliability. He began as a Cobol and Java programmer—even carrying a beeper (remember those?) while on standby. For the past 15 years, he has worked as a Solution Architect across various companies, consistently focusing on resilience and high availability. More recently, he has also taken on the role of managing several teams responsible for guardrails, including Well-Architected reviews, conformity bots, and static code analysis.

Engineering Lead – Conformity Bots

Global Engineering & Reliability

Jagadeesh Devaraj (**JD**) is a seasoned Cloud Architect with over 17 years of experience in IT, specialising in cloud-native platforms, infrastructure automation, and site reliability. Currently at ING, he focuses on building resilient systems using technologies like Conformity Bots. A long-time VMware vExpert and speaker at VMworld and vForum events, JD is also an active blogger and community contributor.

All references to GitLab, AWS, and ING are for educational purposes only.

No endorsement or liability implied.

Agenda

From Outages to Guardrails: The Journey Ahead

Two major outages and their lessons

From Firefighting to Prevention

ING's Closed Control Loop (CCL) framework

Scaling guardrails

Challenges and lessons learned

Could Policy-as-Code Have Prevented the AWS S3 Outage?

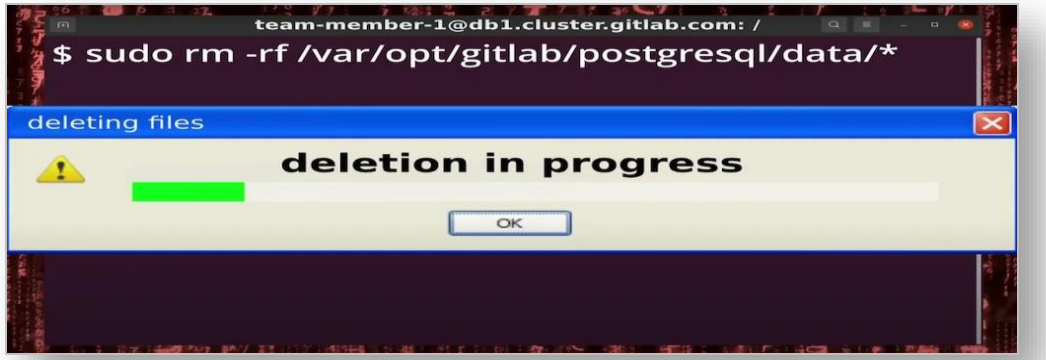
Incident: AWS S3 Outage – Feb 28, 2017

1. An AWS engineer accidentally removed too many servers from a subsystem.
2. This took down the **S3 index subsystem** in **us-east-1**, causing a **global ripple effect**.
3. Many clients had **hardcoded single-region endpoints** and **no failover logic**.
4. The root issue: **no guardrails on operator actions**, and **no resilience in client config**.

```
# 1. Canary required for operator-triggered infra changes
deny[msg] {
  input.actor == "human"
  input.change_type == "infrastructure"
  not input.canary_enabled
  msg := "Operator changes must be canaried before full rollout"
}

# 2. Blast radius control for destructive commands
deny[msg] {
  input.command == "remove_servers"
  count(input.targets) > 3
  not input.approved_by == "senior_ops"
  msg := "High-impact infra changes require senior approval"
}

# 3. Enforce multi-region fallback for S3 clients
deny[msg] {
  input.service == "s3"
  input.endpoint == "us-east-1.amazonaws.com"
  not input.has_fallback
  msg := "S3 clients must define fallback region or CDN"
}
```

Ref: https://youtu.be/tLdRBsuvVKc
https://aws.amazon.com/message/41926/

Could Policy-as-Code Have Prevented the GitLab Data Loss?



Incident: GitLab – Jan 31, 2017

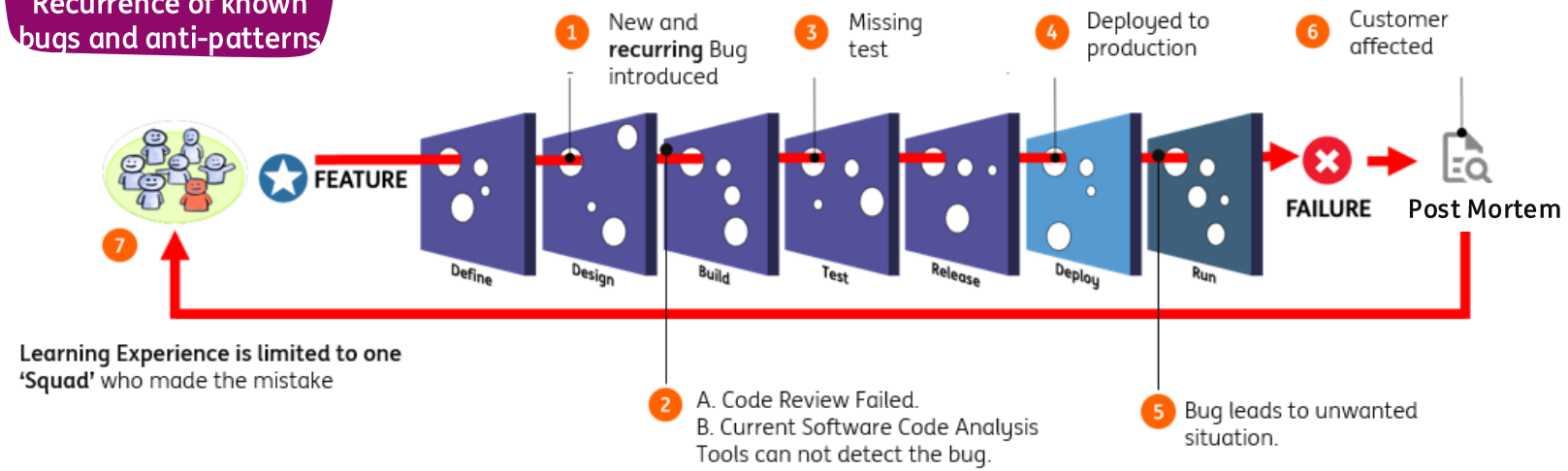
1. An engineer accidentally ran a destructive command (rm -rf) on the production database.
2. Backups were either **corrupted**, **incomplete**, or **not restorable**.
3. GitLab lost **6 hours of production data**.
4. Root causes: **manual DB access**, **no enforced snapshot policy**, and **untested backup restore paths**.

```
# 1. Block destructive DB operations unless a verified snapshot exists
deny[msg] {
  input.action == "db_delete"
  not input.snapshot_created
  msg := "Snapshot required before destructive DB operation"
}

# 2. Require backup restore test to pass before trusting backup
deny[msg] {
  input.backup_verified != true
  msg := "Backup must pass restore test before use"
}

# 3. Restrict manual DB access to emergency-only workflows
deny[msg] {
  input.actor == "human"
  input.resource == "production_db"
  not input.is_emergency
  msg := "Manual access to production DB is restricted to emergency workflows"
}
```

We need to shift from firefighting to prevention

Guardrails That Drive 99.9% Uptime

Challenge:
Recurrence of known bugs and anti-patterns



Solution:
Reliability by Design

Guardrails

Well-architectured Review

An interview to identify improvements in the architecture

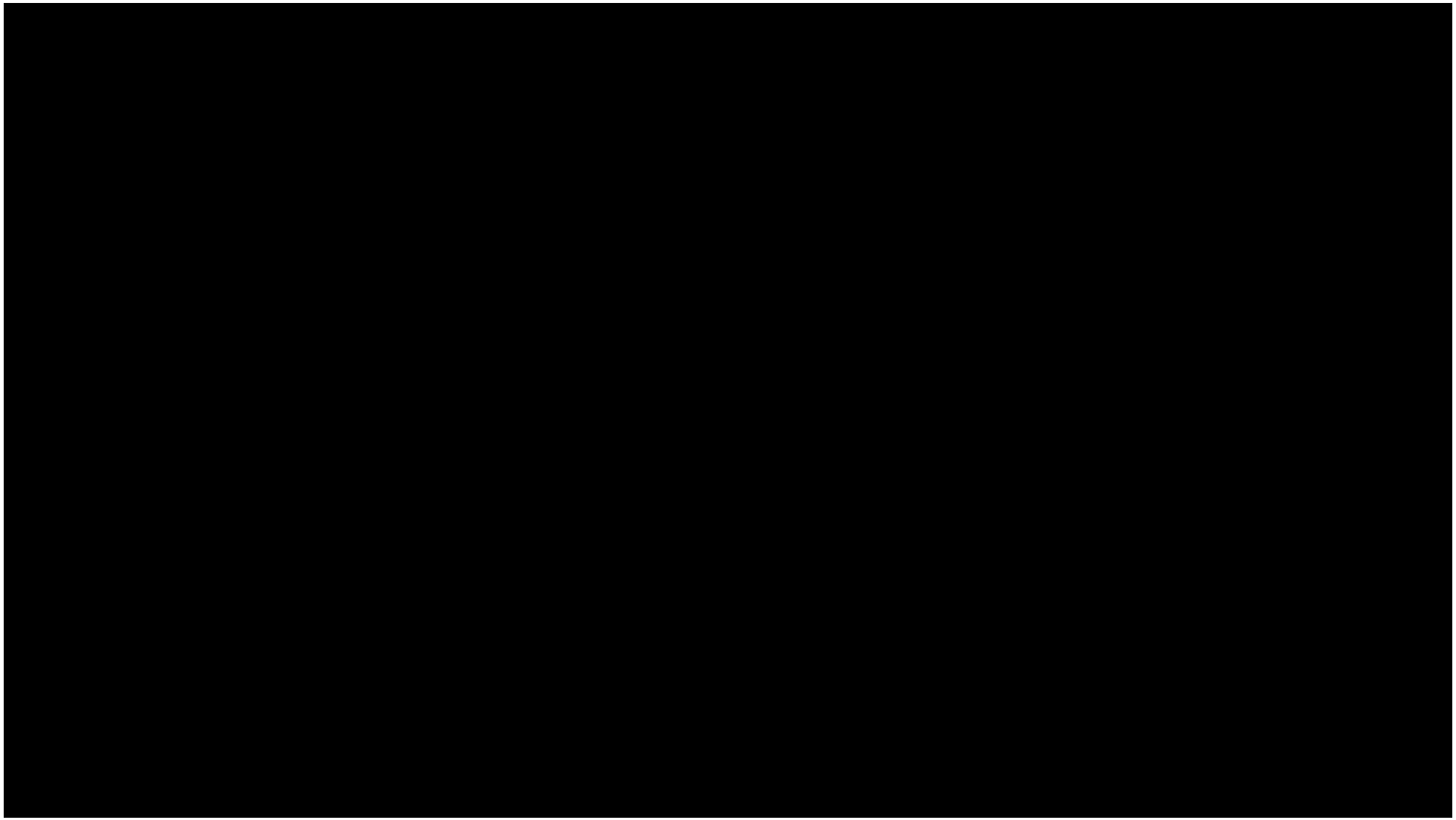
Conformity Bots

Enforce conformity to best-practice reliability patterns

Code Scanning

Detecting bugs and anti-patterns early with Static Code Analysis

Catch, Correct, Continue: Guardrails in Motion



Big Code, Small Blast Radius: ING's Reliability Story

Over 20 ING entities (countries) and over 20 CIOs

Over 1M Pipeline runs every month

15K engineers, of which 12K active committers

30K active Repos

Over 50 programming languages

400K policy checks per month

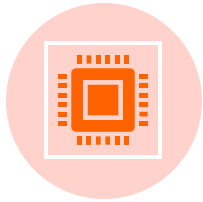
Over 100M lines of code

17K changes each month

Reliability by Default: ING's Vision for Continuous Prevention

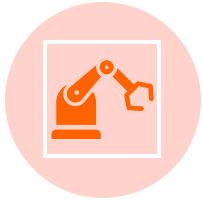
Reliability is built in—not bolted on

Every change is treated as a signal



Sensors across CI/CD and runtime

Codified wisdom via Policy-as-Code



Safe, reversible actuators

Logged decisions for audit and learning

Closed Control Loop: Our Misconfig Radar

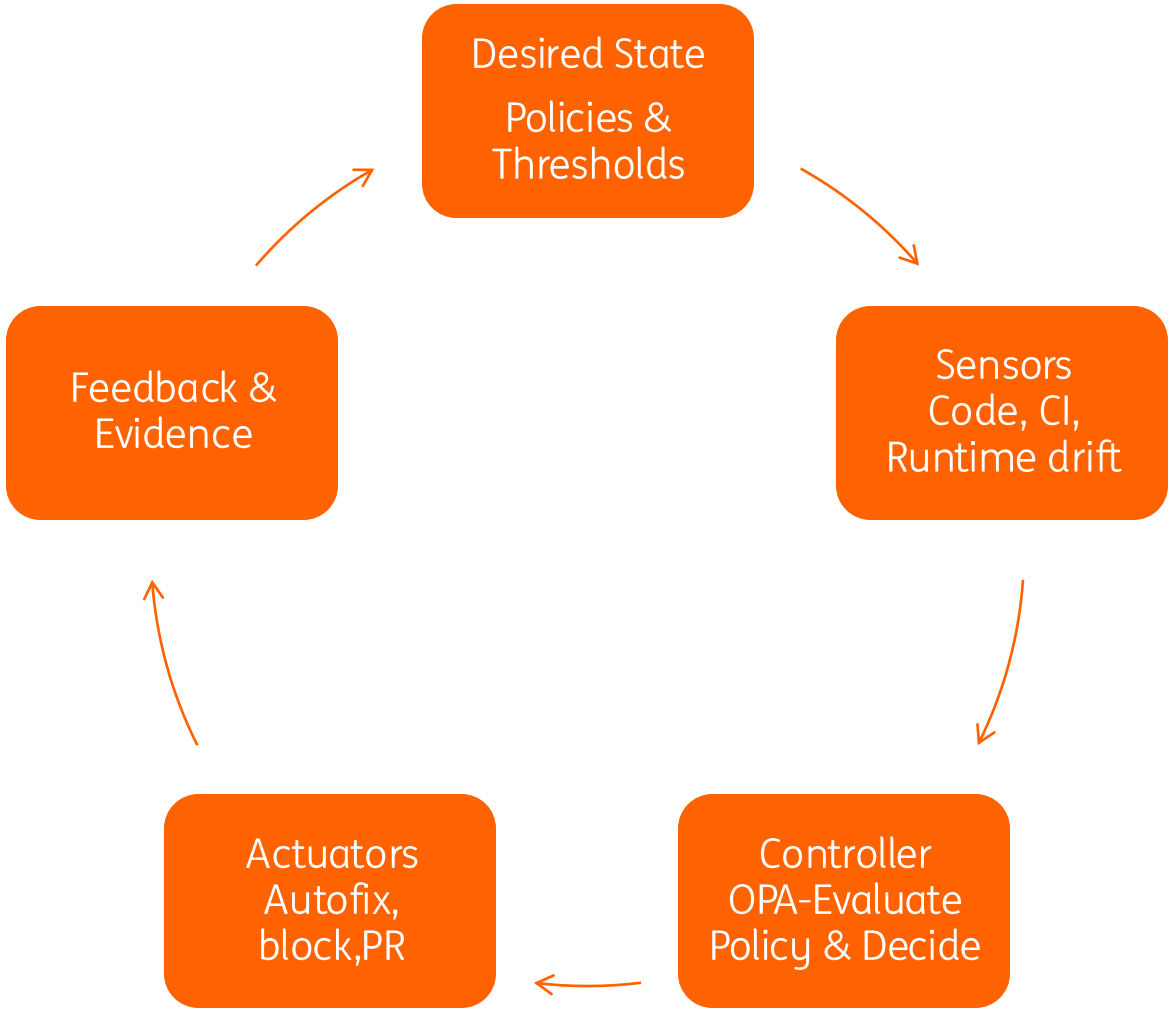
A Framework for Proactive Management

Purpose:

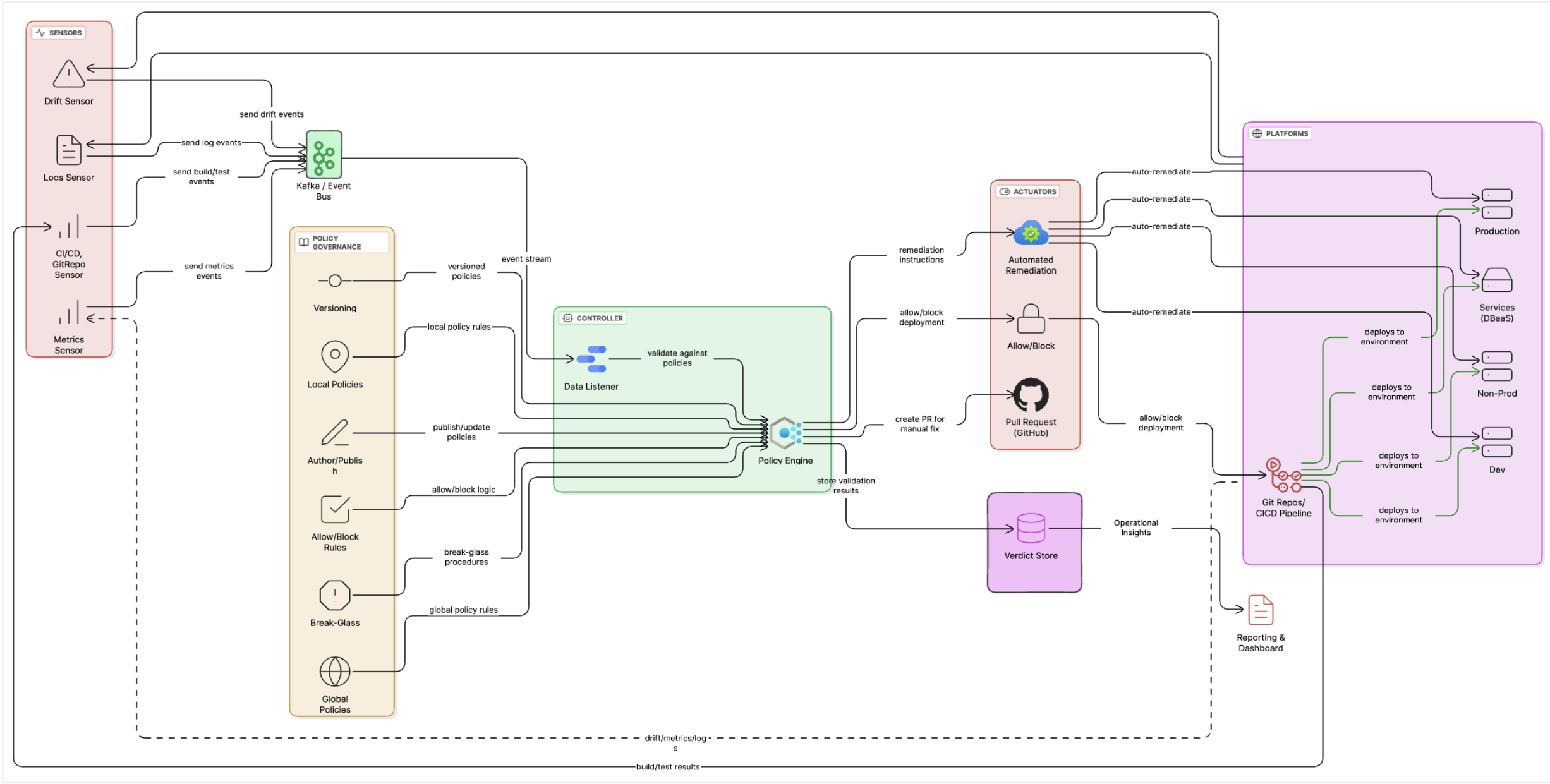
- Prevent misconfigurations before production
- CCL **continuously monitors** the sensors against the desired state.

Core Components:

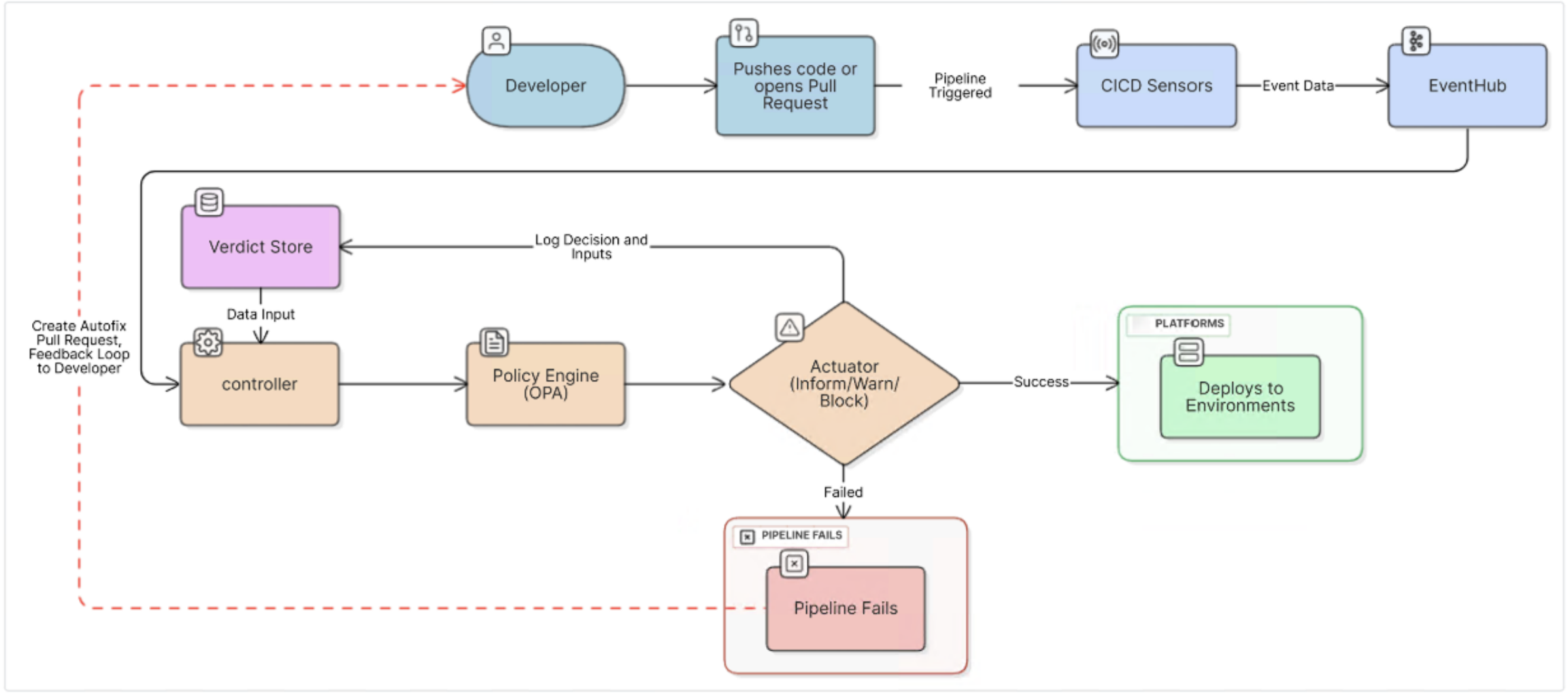
- **Setpoints/Desired State**: Policies & thresholds
- **Sensors**: Monitor code, CI/CD, runtime
- **Controller**: Evaluate policy & decide
- **Actuators**: Fix, block, or PR
- **Feedback**: Evidence & learning



Inside the Drift Radar: ING's CCL Architecture



CCL Architecutre For Policy as Code in Pipelines



Policy-as-Code: Writing the Rules of the Chase

```
package reliability.ha

# Rule 1: Require at least 2 regions
violation[msg] {
  count(input.spec.regions) < 2
  msg := "Deployment must span at least 2 regions for HA"
}

# Rule 2: Require at least 2 DB connection strings
violation[msg] {
  count(input.spec.dbConfig.connectionStrings) < 2
  msg := "Database must define >= 2 connection strings for HA"
}
```

```
# pod-deployment-drift.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payments-api
spec:
  replicas: 1                    # ❌ single replica
  regions:
    - eu-west-1                  # ❌ only one region
  dbConfig:
    connectionStrings:
      - postgres://db1.example.com   # ❌ only one DB endpoint
```

```
# pod-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payments-api
spec:
  replicas: 3
  regions:
    - eu-west-1
    - eu-central-1               # ✅ multi-region deployment
  dbConfig:
    connectionStrings:
      - postgres://db1.example.com
      - postgres://db2.example.com  # ✅ dual endpoints for HA
```

Governance: How We Approve the Guardrails

1.Propose

- Codified & documented form (entry level for policy as code)
- **Assess impact with stakeholders** (testing involved)
- Define Rollout plan (enforcement, targets)
- **A policy must be approved by the Governance Board**

Process owners present proposal

2.Deploy

- **Managing enforcement levels (inform, warn, block)**
- Deployment strategies are applied
- Deploy policies to platforms (First non-prod, then prod)

Policy deployed to the execution platform- CCL

3.Review

- **Policy execution**
- Collect feedback
- Monitor impact
- Policy continuous review

4.Retire

- **Archive policy for auditing (ex. SoX)**
- Document outcome
- Goes through the update cycle

Sent back for refinement

Updating enforcement level

Sent back for updates

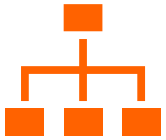
Waivers: Temporary exceptions approved by board
Break-the-Glass: Emergency override by Incident Manager & Platform Owner

Scaling Guardrails

Implementing CCL Across the Org

Ambitious Targets Goal:

500+ Policies/Guardrails deployed across platforms and services by 2026—embedding reliability into Org's delivery DNA.



Governance Framework

A central governance board oversees policy lifecycle, risk alignment, and rollout of policies across the Org.

Global Standards

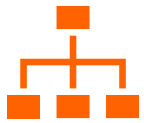
Encryption, HA configurations, and RBAC are standardized across environments to ensure consistency and compliance.

Local Adaptations

Teams implement domain-specific guardrails tailored to their platform, service, and operational context.

SRE Playbook

Engineering Reliability Before It Breaks



Turn experience into code:

Automate best practices as Policy-as-Code

Look beyond pipelines:

Detect drift in runtime, not just CI

Guardrails help, not hinder:

Start with guidance, then enforce

Close the loop:

Detect → Decide → Act → Learn

Scale smart:

Use governance to grow without slowing teams

Shift everywhere:

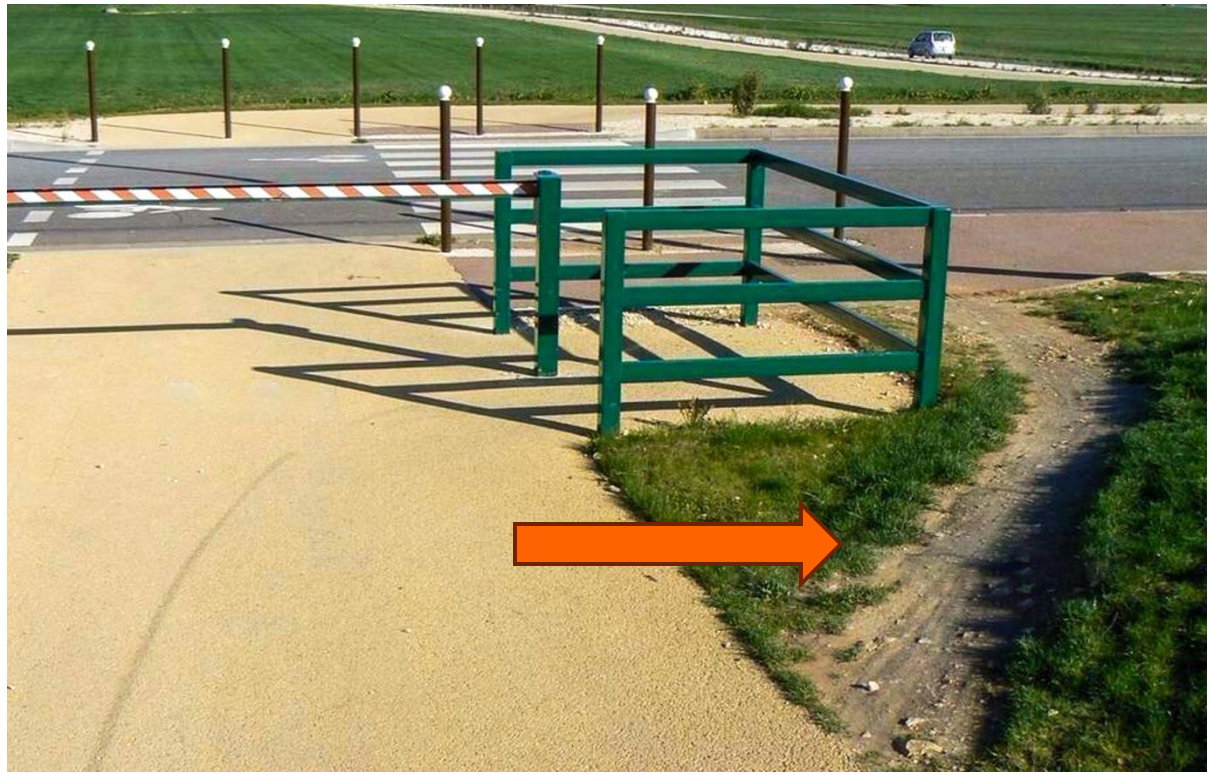
Catch issues early (left) and monitor in production (right)

Lessons from the Field: What We Learned Hunting Misconfigs

Culture is the First Guardrail: Awareness Before Action

- Create a **single source of truth** for reliability principles and patterns
- Be **easy to find** in the **developer portal**
- Establish a **Core Team** of established, skilled experts
- Foster a **community of practice**
- Drive reliability forward with **gamification,** targeted trainings, etc.

Reliability isn't our target – It's our default



An unaware engineer will always find the gap

Fast Pipelines, Smart Policies: No Bottlenecks in the Chase

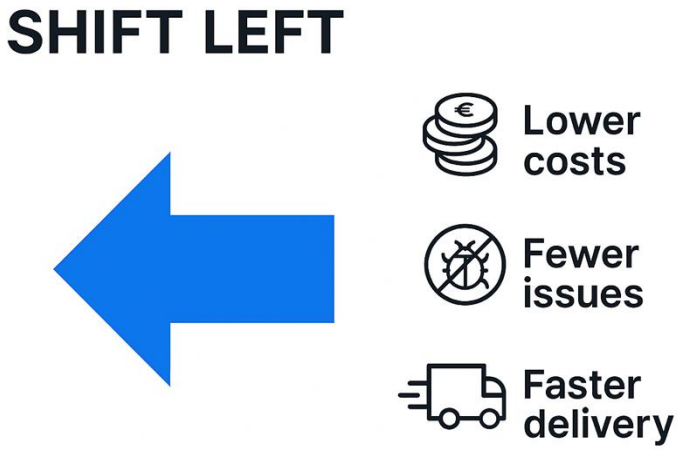
CI/CD pipelines are the backbone of a secure and reliable environment. Embedding synchronous policy checks directly in the pipeline can turn them into bottlenecks.

Policy validation should be designed to avoid becoming a performance choke point in the CI/CD pipeline.

Lesson learned:

Aim for an **asynchronous** Policy validation.

Shift Left or Pay Later: Catch Misconfigs Early



Shifting left:

- **Accelerates delivery** and prevents **firefighting**
- Boosts developer **confidence**
- Bug fixing in pre-prod saves **14x the cost** of solving after implementation [1] [2]

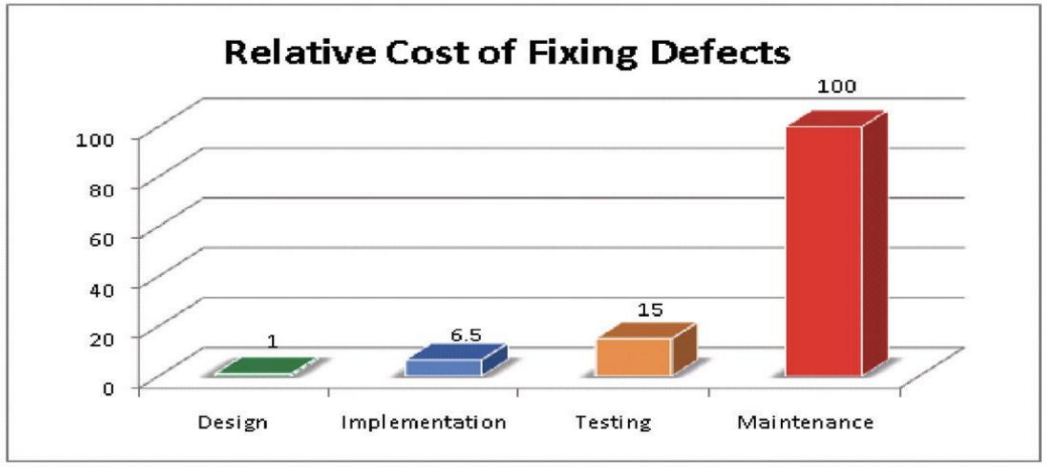


Figure 3: *IBM System Science Institute Relative Cost of Fixing Defects*

We run policy checks across every stage of the SDLC within our pipelines.

[1] https://www.researchgate.net/publication/255965523_Integrating_Software_Assurance_into_the_Software_Development_Life_Cycle_SDLC
[2] https://github.blog/news-insights/product-news/secure-code-more-than-three-times-faster-with-copilot-autofix

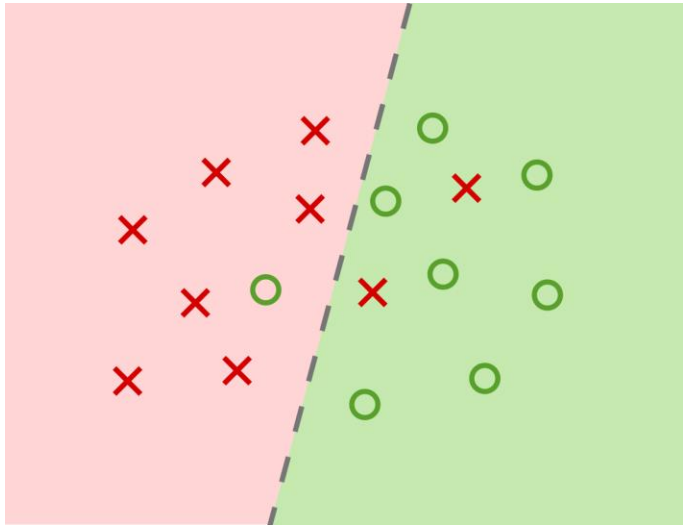
False Positives: The Hidden Cost of Misconfig Hunting

Controller precision is critical. Too many false positives create noise, frustrate teams, and waste resources.

False positives can occur due to:

Inconsistent repo structure
Hard to distinguish prod vs. non-prod code

Non-prod ≠ Prod
Missing resilience patterns in non-prod environments

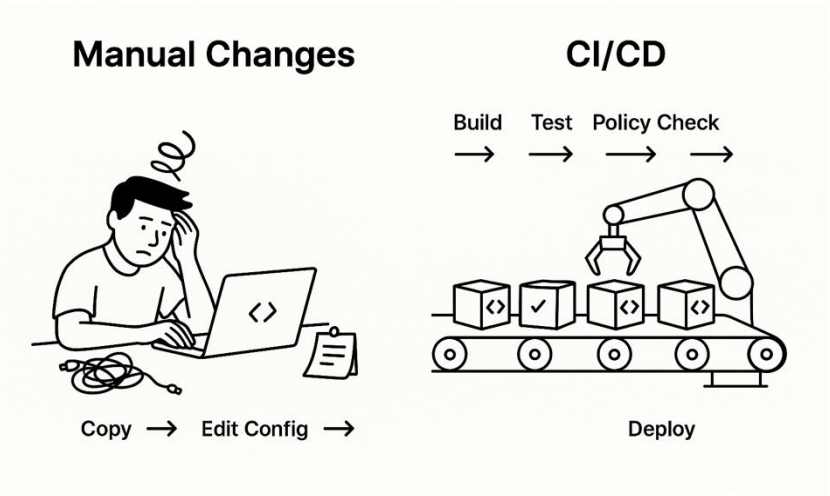


<u>Solution:</u>

- **Standardize repo structure**
- Make pre-prod behave like prod - **Ephemeral, production-like environments**: spin up with the *same* images, configurations, and policies as prod; tear down after tests.

Drift Detection: Catching What Pipelines Miss

Not all changes flow through pipelines, and not every platform is truly immutable. That's why we need Drift Detection — to catch what slips through.



Solution:

Drift Detection: Policy validation is triggered by sensors on platforms, services and repositories.

Nudge First, Block Later: Smart Policy Rollouts

With policies, we can either act like the traffic light — blocking you until you comply — or like the speed sign, raising awareness and letting you self-correct.

VS

Adopt a phased rollout: start in audit/warning mode, then switch to blocking mode after three months

When Reliability Meets Security: A Tactical Collision

Example:
From a reliability perspective, we want progressive delivery with at least a 4-hour delay before full rollout. But this might also mean privileged accounts stay active longer…

Establish clear **Policy Governance**, approved by all stakeholders, with the CISO actively represented on the approval board.

Thanks for Joining the Hunt!

Let's Build Reliability Together

Slack us @ 25emea-day1-track2

ing.com

@ING_News

LinkedIn.com/company/ING

Medium.com/ing-blog

Facebook.com/ING

YouTube.com/ING

Flickr.com/INGGroup

SlideShare.net/ING