From Guidance to Guardrails: Cost & Carbon Policy-as-Code with OPA in CI

Machiko Shinozuka and Kouki Hama

2026/5/19

About us

Machiko Shinozuka

Senior Research Engineer at NTT Computer and Data Science Laboratories

Background

- Joined NTT in 2015
- Ordinally specialized in Environmental Impact Assessment
  - LCA : Life-Cycle Assessment

Interested in

- Green Software
- FinOps
- Sustainability cloud operations

Also enjoy

- Swimming
- Coffee
- Sports manga
- Accounting

About us

Kouki Hama
NTT, Inc
Senior Research Engineer
Japan

Linked In
QR Code

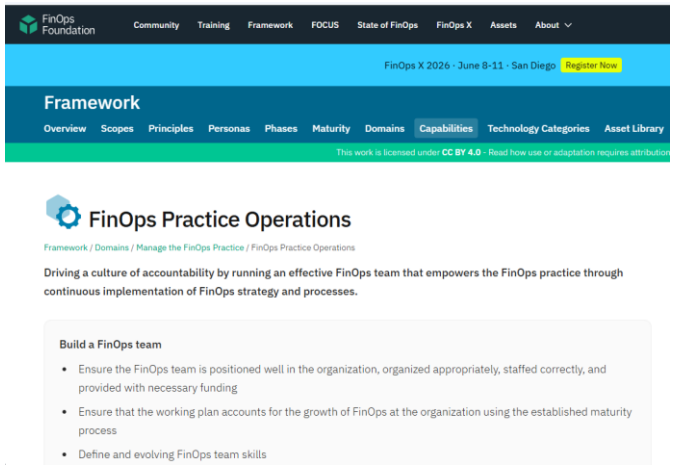
Kouki Hama is a Senior Research Engineer in software engineering at NTT, Inc., Computer & Data Science Laboratories. His research focuses on improving the efficiency, reliability, and governance of CI/CD, with a focus on GreenOps, FinOps, reliability engineering, and software supply chain assurance. In particular, he aims to make build and release artifacts and metadata—such as SBOMs—operationally actionable for decision-making in real-world systems, while enabling green software optimization based on energy efficiency and Software Carbon Intensity. In addition to his professional work, he contributes to the SBOM Sub Working Group in the OpenChain Japan community and serves as a co-leader of Eclipse SW360, an open-source SBOM catalog and management platform.

Agenda

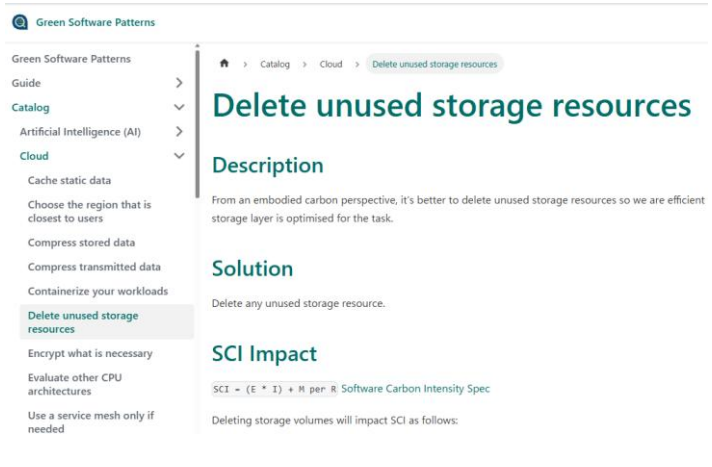
1. Introduction
2. Challenge: The Gap from Guidance to Decision-Making
3. Approach: Policy-as-Code with OPA
4. System Design and Architecture
5. Policy Patterns and Representative Rules
6. Implementation and Operations
7. Demonstration
8. Future Roadmap and Takeaways

Multiple Concerns in Cloud Operation

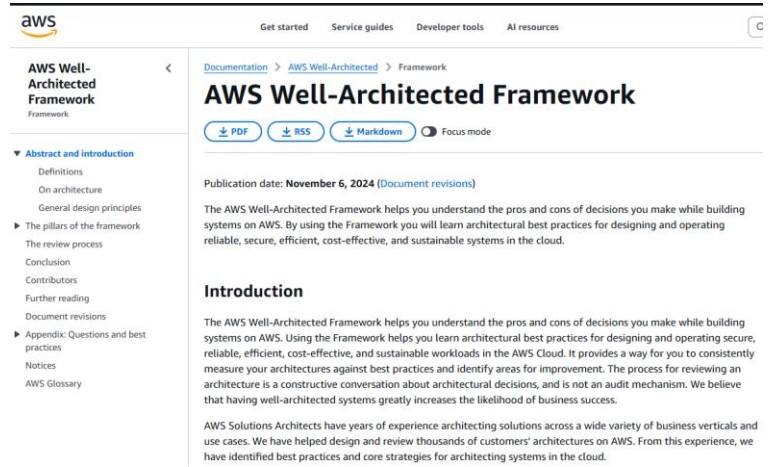
Cloud operations require balancing multiple concerns such as cost, performance, and sustainability. Various guidelines—such as the FinOps Framework and Green Software Patterns—provide principles for improving these aspects.



https://www.finops.org/framework/capabilities/finops-practice-operations/



https://patterns.greensoftware.foundation/catalog/cloud/delete-unused-storage-resources



https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html

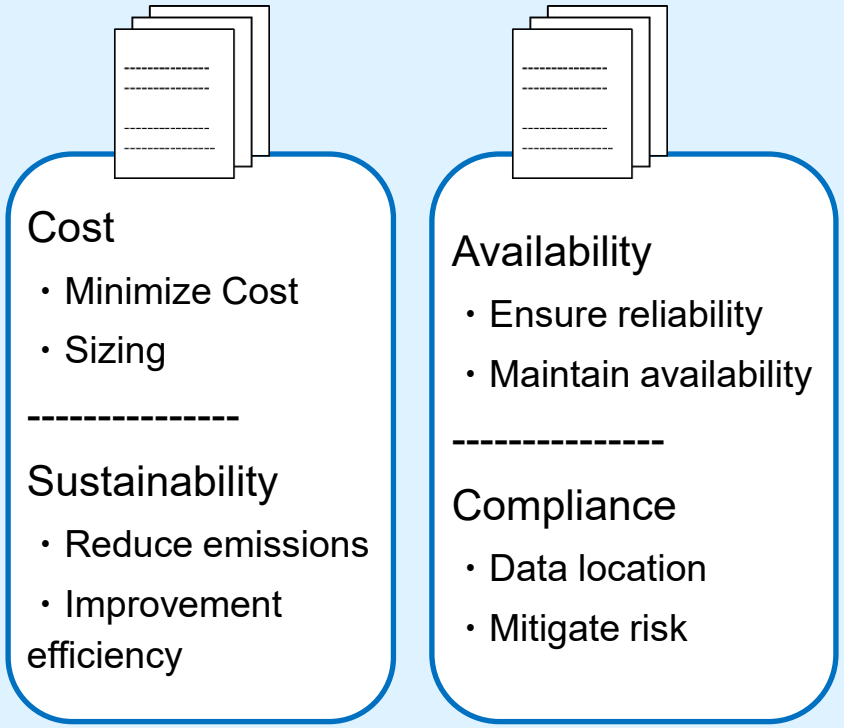
Agenda

1. Introduction
2. Challenge: The Gap from Guidance to Decision-Making
3. Approach: Policy-as-Code with OPA
4. System Design and Architecture
5. Policy Patterns and Representative Rules
6. Implementation and Operations
7. Demonstration
8. Future Roadmap and Takeaways

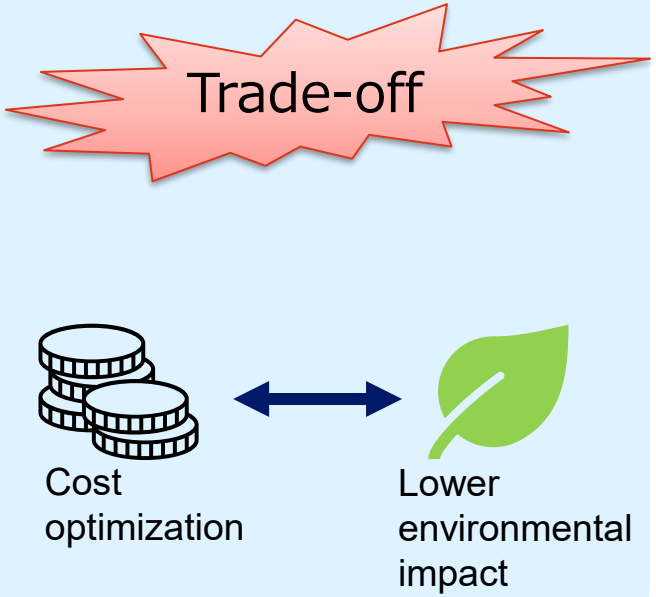
From Guidance to Decisions is Not Straightforward

Translating guidelines into concrete decisions is not straightforward.
They often mix abstract principles with practical details, making consistent decision-making difficult in manual review processes.

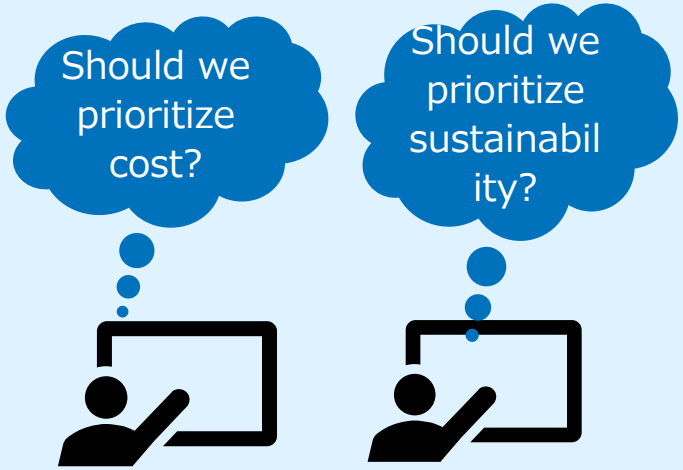
Guidelines are multi-dimensional and abstract



Principle create trade-offs



Decision depends on context



Agenda

1. Introduction
2. Challenge: The Gap from Guidance to Decision-Making
3. Approach: Policy-as-Code with OPA
4. System Design and Architecture
5. Policy Patterns and Representative Rules
6. Implementation and Operations
7. Demonstration
8. Future Roadmap and Takeaways

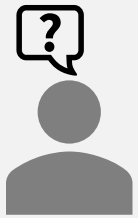
Our Approach: Policy-as-Code with OPA

We translate guidance into machine-readable rules and evaluate them systematically using Open Policy Agent (OPA).
This allows decisions to be made consistently based on the same inputs.

Before: Human Judgement

Decisions vary depending on who reviews them.

Operational Guidance
Policies, Rules, Standards, etc



Manual Interpretation
Each reviewer interprets the guidance

Review Decisions
Decisions may vary by reviewer

Inconsistent Decisions
Outcomes differ across teams and time

After: Policy-as-Code with OPA

Consistent decision through machine-evaluated policies.

Operational Guidance
Policies, Rules, Standards, etc

Translate to Machine-Readable Rules
Encode guidance as executable policies

OPA evaluation
Evaluate policies systematically

Consistent Decisions
Same inputs lead to the same outcomes

Turn guidance into code. Evaluate with OPA. Achieve consistent, repeatable decisions.

Why OPA?

- Works across tools and platforms:
  OPA evaluates structured inputs from IaC, configuration files, and APIs without being tied to a specific system.

- Policy as code for consistent decisions:
  Rego enables declarative policies that are testable and can be evaluated consistently in CI.

- Decouples policy from implementation:
  Policies can be applied consistently across different environments without changing application logic.

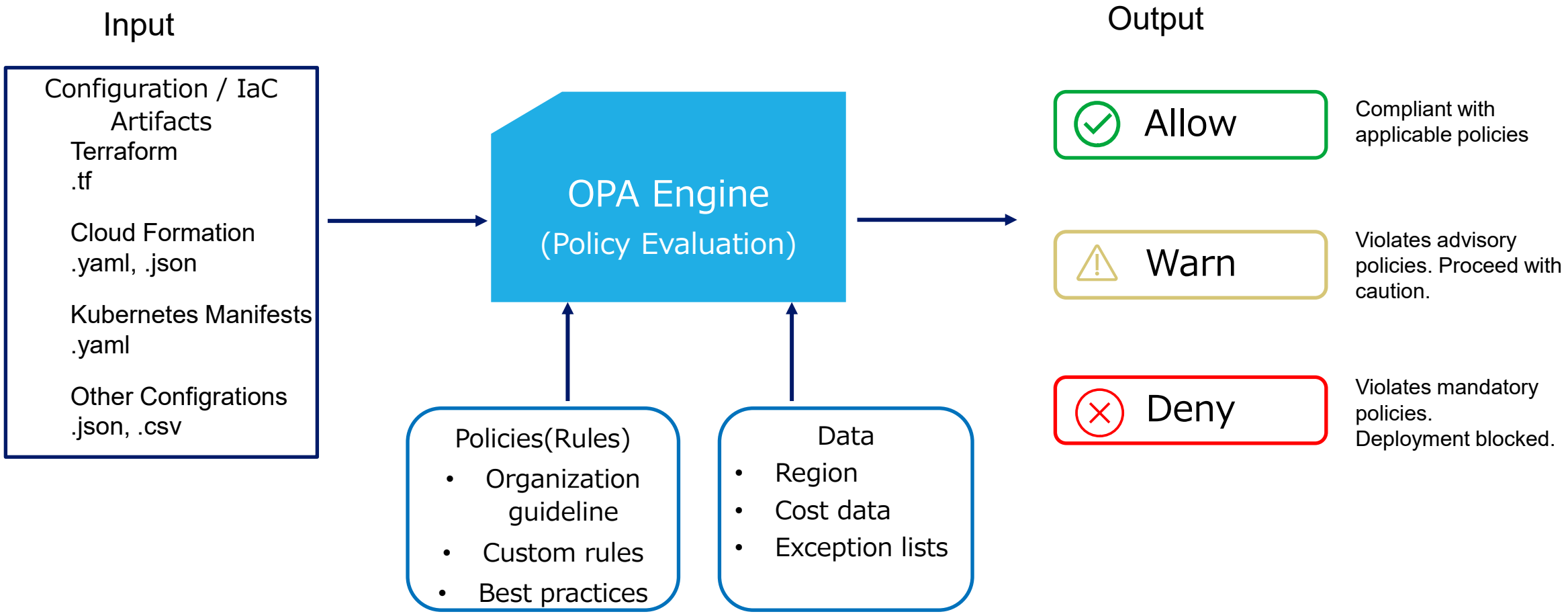
Open Policy Agent

Agenda

1. Introduction
2. Challenge: The Gap from Guidance to Decision-Making
3. Approach: Policy-as-Code with OPA
4. System Design and Architecture
5. Policy Patterns and Representative Rules
6. Implementation and Operations
7. Demonstration
8. Future Roadmap and Takeaways

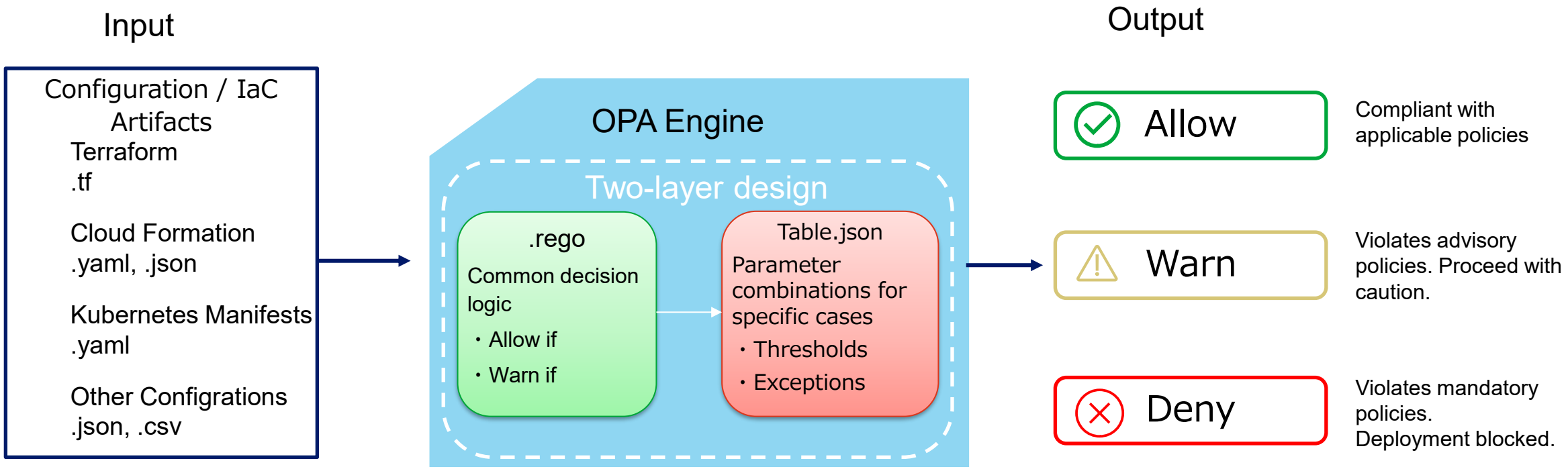
System Overview

At a high level, the system takes configuration or IaC artifacts as input, evaluates them through a policy engine, and produces decisions with explanations.
This enables consistent and repeatable evaluation within CI workflows.



Key Design Goal for Practice

The system must handle multiple concerns without breaking evaluation and remain maintainable even for contributors without deep Rego expertise.



Key Points of this design

1. Safety handle conflicts
- Evaluate all concerns in parallel
- Use defined rules to resolves conflicts
- Don't stop evaluation – provide outcomes with reasons

2. Two-layer design: Rego + JSON
- Non-Rego experts can update thresholds and exceptions in JSON
- Rego does not need changes for every threshold/exception update
- JSON Schema validation catches malformed decision tables in CI

Agenda

1. Introduction
2. Challenge: The Gap from Guidance to Decision-Making
3. Approach: Policy-as-Code with OPA
4. System Design and Architecture
5. Policy Patterns and Representative Rules
6. Implementation and Operations
7. Demonstration
8. Future Roadmap and Takeaways

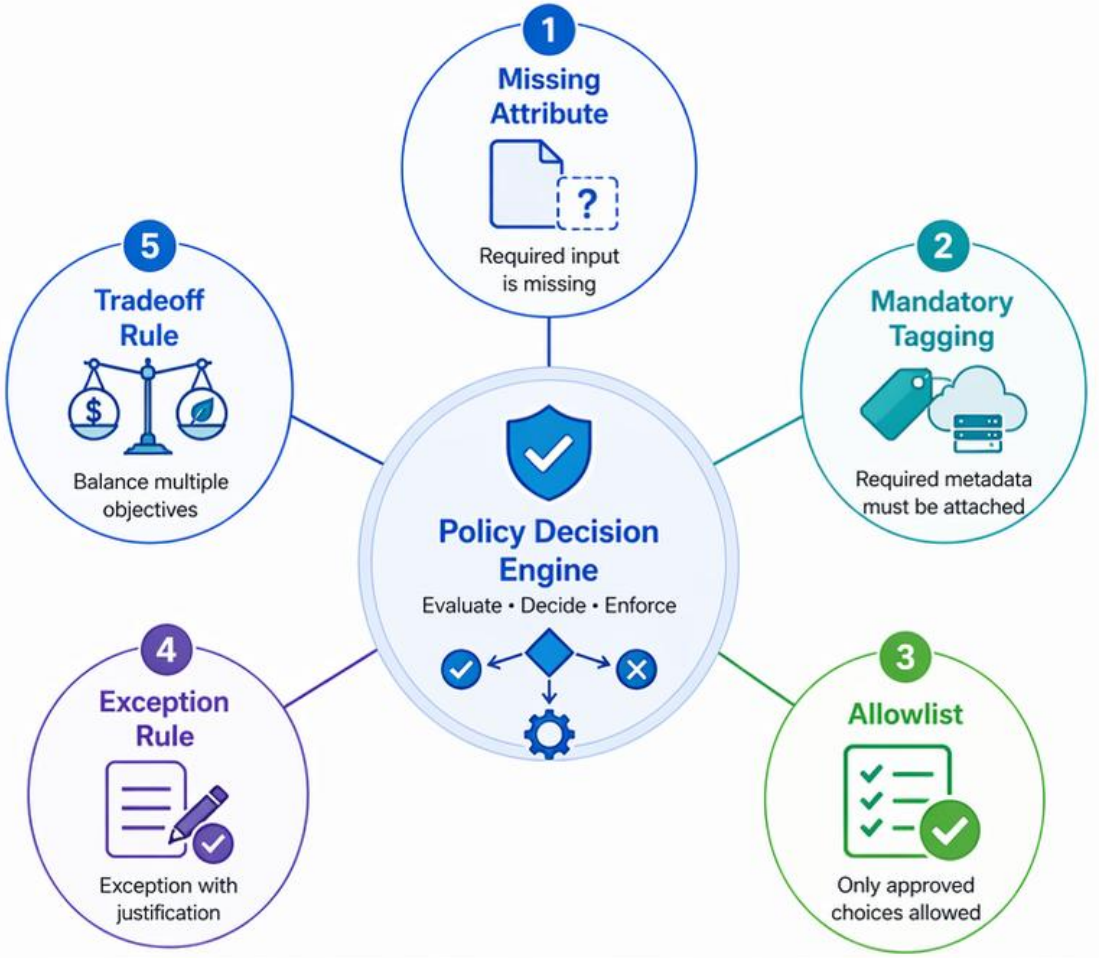
Policy Patterns for Cloud Governance

Rather than treating each rule as an isolated example, we organize them as **reusable policy patterns** for cloud governance.

These patterns capture recurring decision structures that appear when translating guidance into machine-enforceable rules:

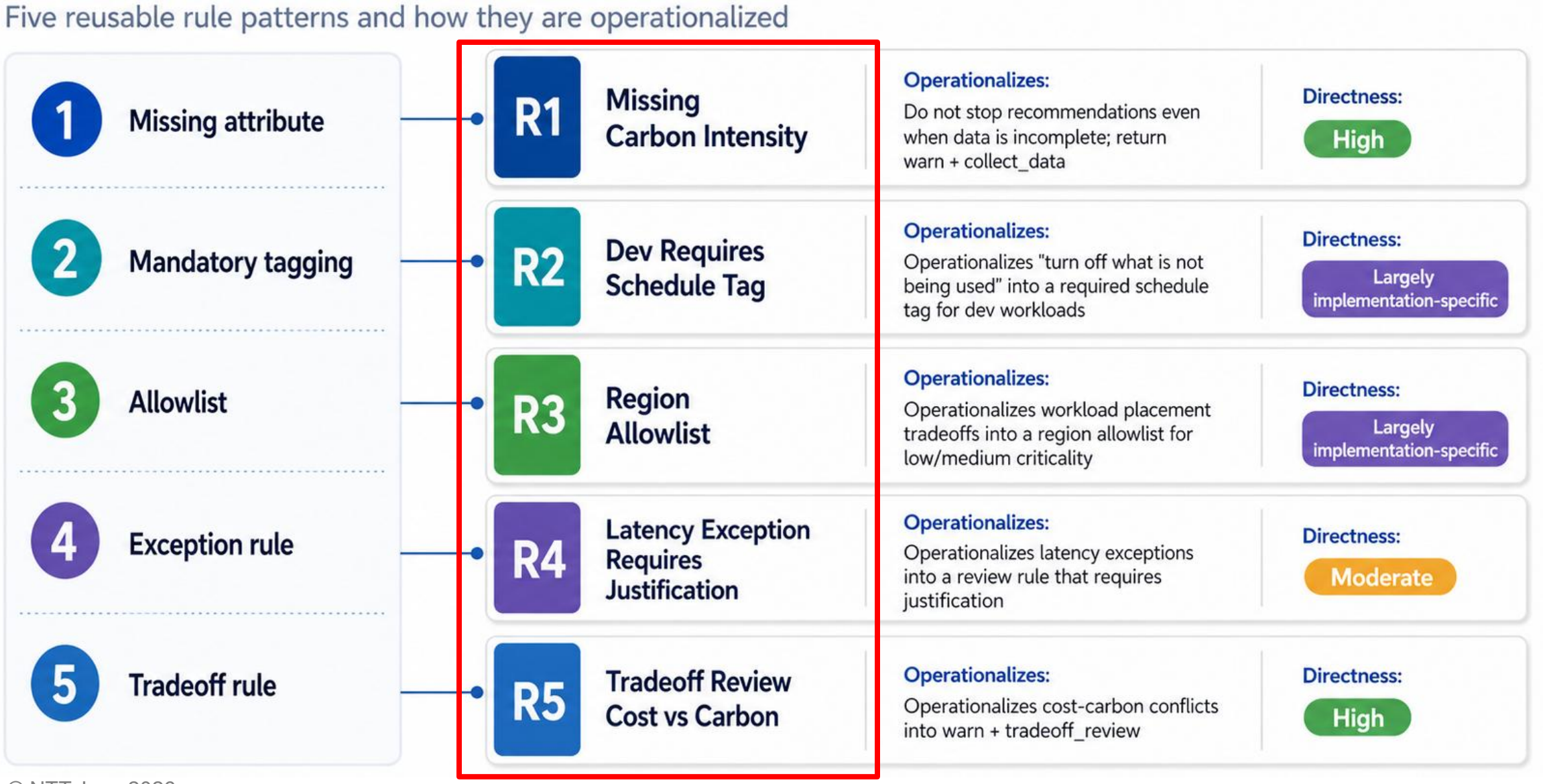
1. **Missing attribute**
2. **Mandatory tagging**
3. **Allowlist**
4. **Exception rule**
5. **Tradeoff rule**

This view helps practitioners design new policies systematically, not case by case.



From rules to reusable patterns.
A consistent structure to build, understand, and evolve policies at scale.

Policy Rules and Representative Example



Sutanability Capability

Pattern 1: Missing Attribute

What is it

Some decisions cannot be made safely when required context is missing.

How it works

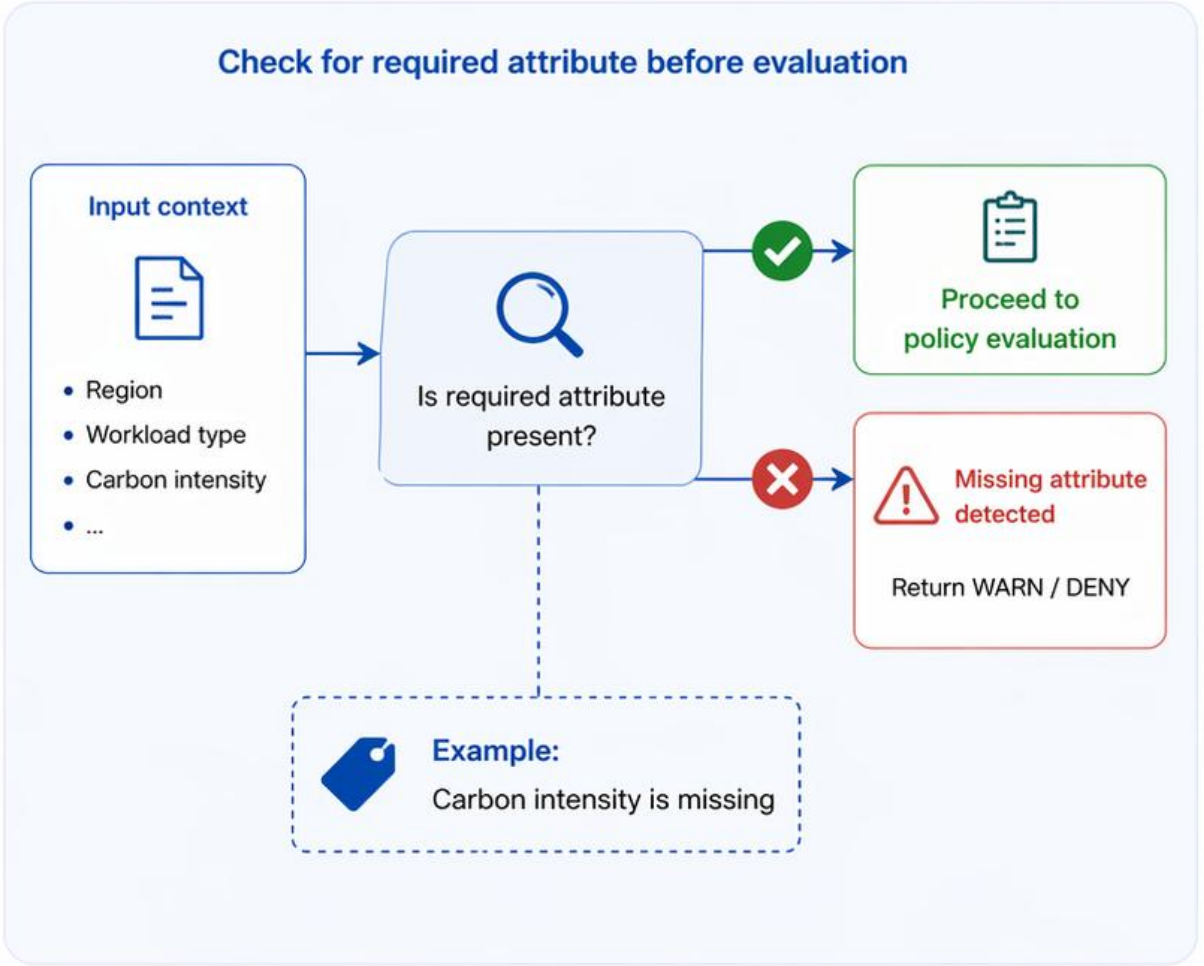
This pattern checks whether critical input attributes are present before evaluating downstream policies.

In our example

The system detects when carbon intensity information is missing.

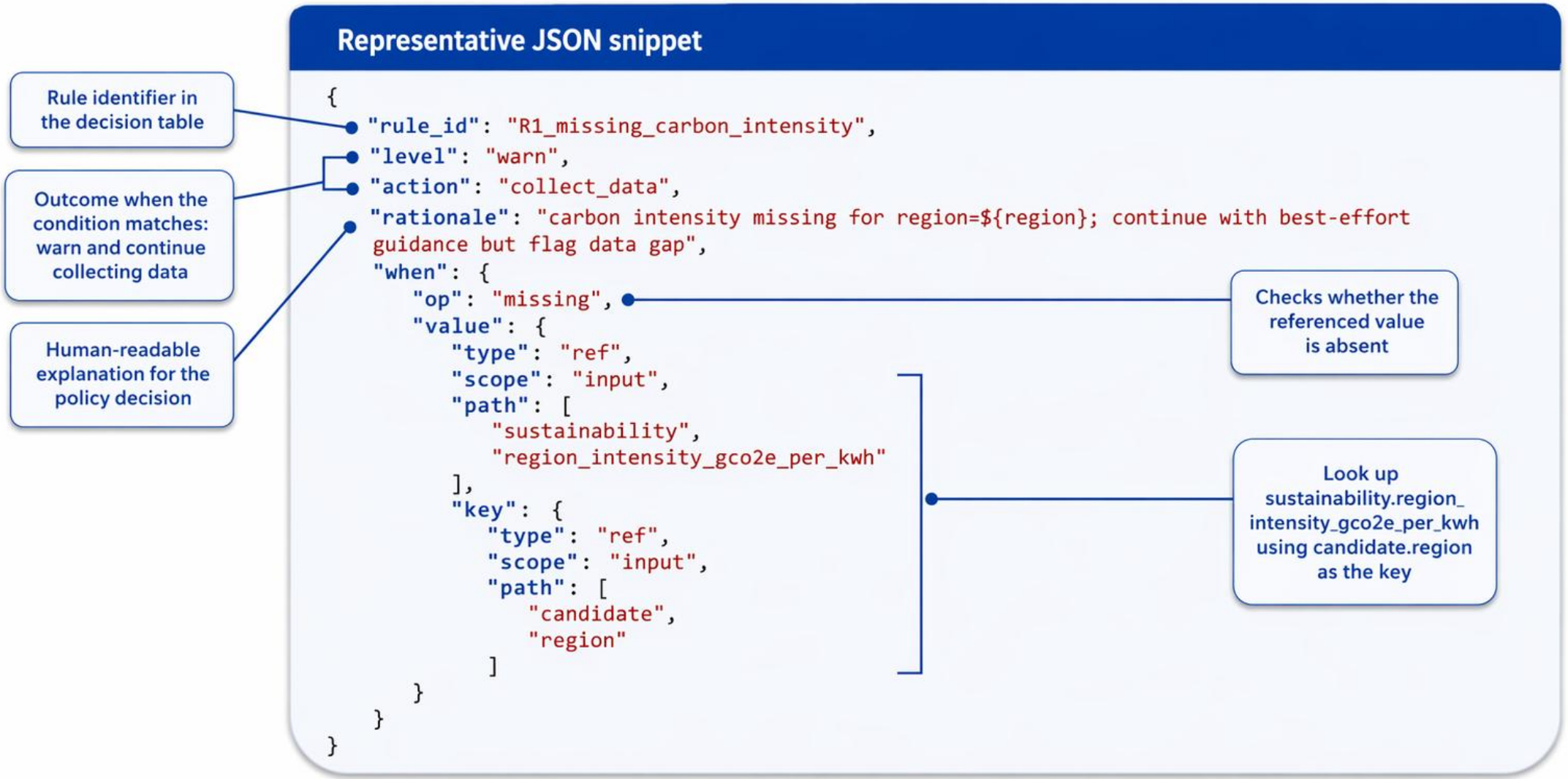
Why it matters

- incomplete input leads to unreliable decisions
- early detection prevents false confidence
- missing context can be surfaced as warn or deny depending on policy intent



Representative rule: Missing carbon intensity

Json Snippet: Missing Attribute



Pattern 2: Mandatory Tagging

What is it

Many governance policies depend on metadata that identifies ownership, environment, or operational intent.

How it works

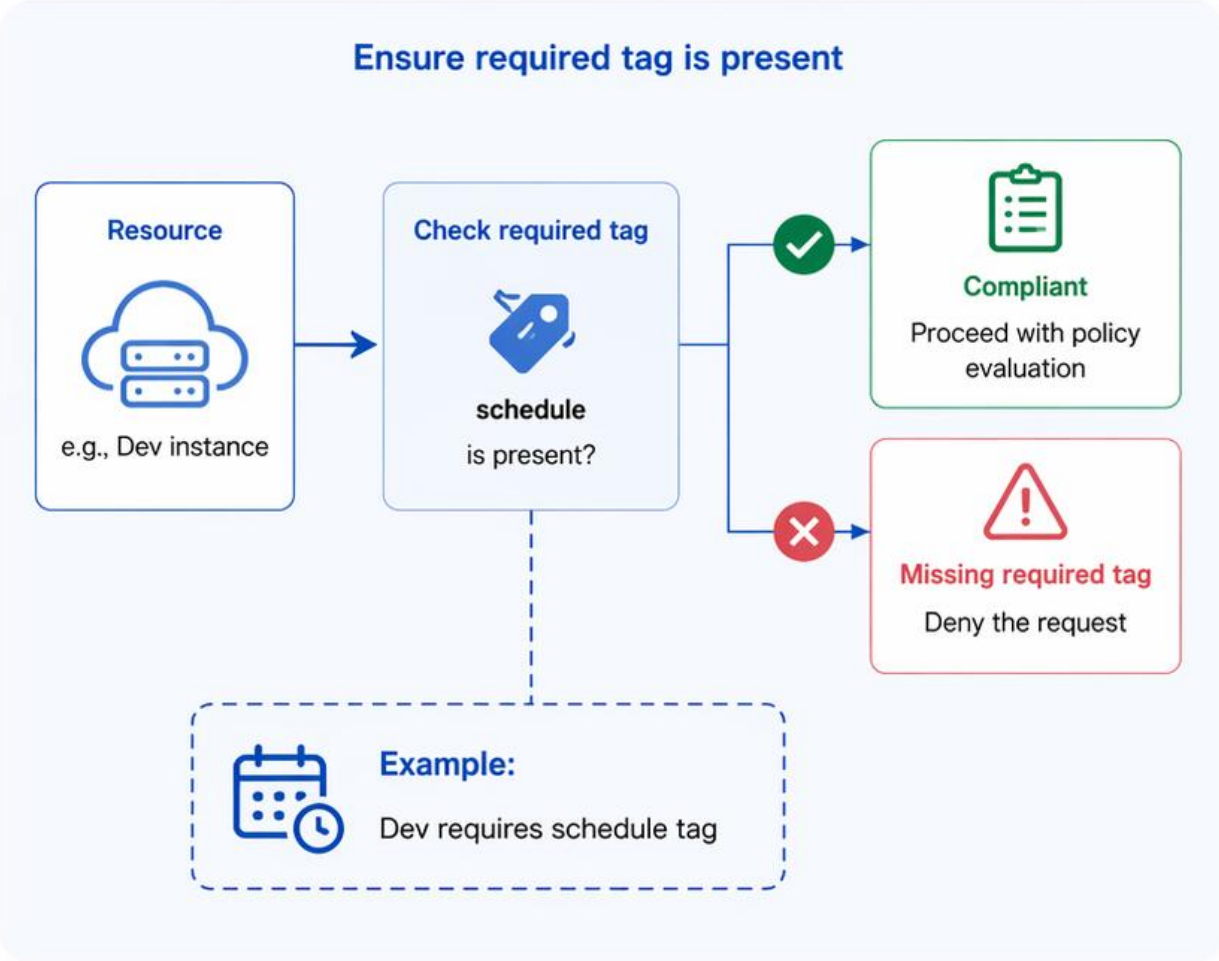
This pattern enforces the presence of required tags so that cost, accountability, and automation decisions can be applied consistently.

In our example

Development resources must include a schedule tag.

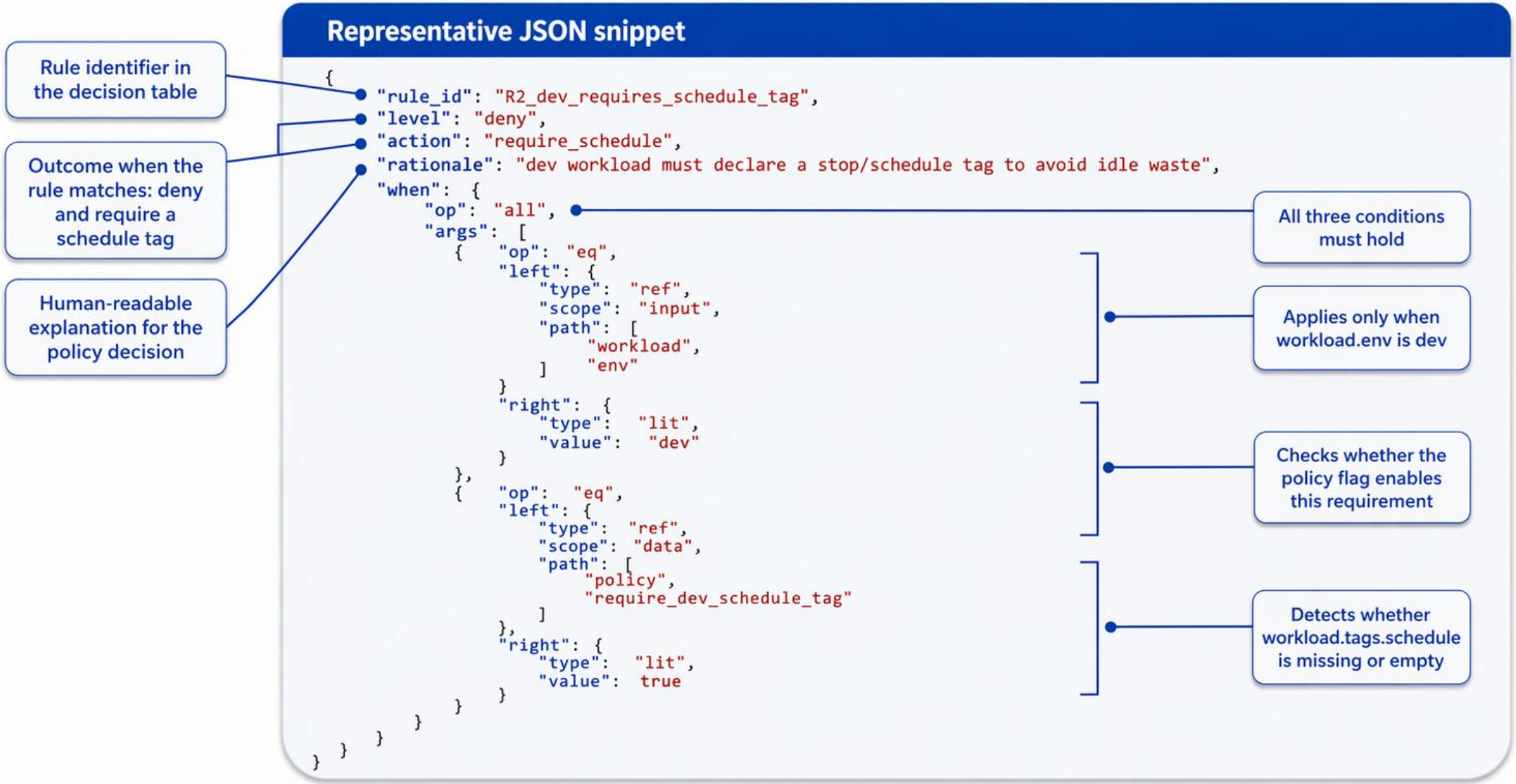
Why it matters

- tags enable allocation and accountability
- downstream automation depends on consistent metadata
- the rule is simple, but operational impact is high



Representative rule: Dev requires schedule tag

Json Snippet: Mandatory Tagging




```
{
  "rule_id": "R2_dev_requires_schedule_tag",
  "level": "deny",
  "action": "require_schedule",
  "rationale": "dev workload must declare a stop/schedule tag to avoid idle waste",
  "when": {
    "op": "all",
    "args": [
      {
        "op": "eq",
        "left": {
          "type": "ref",
          "scope": "input",
          "path": [
            "workload",
            "env"
          ]
        },
        "right": {
          "type": "lit",
          "value": "dev"
        }
      },
      {
        "op": "eq",
        "left": {
          "type": "ref",
          "scope": "data",
          "path": [
            "policy",
            "require_dev_schedule_tag"
          ]
        },
        "right": {
          "type": "lit",
          "value": true
        }
      }
    ]
  }
}
```

Pattern 3: Allowlist

What is it

Some governance decisions are best expressed as explicit sets of acceptable options.

How it works

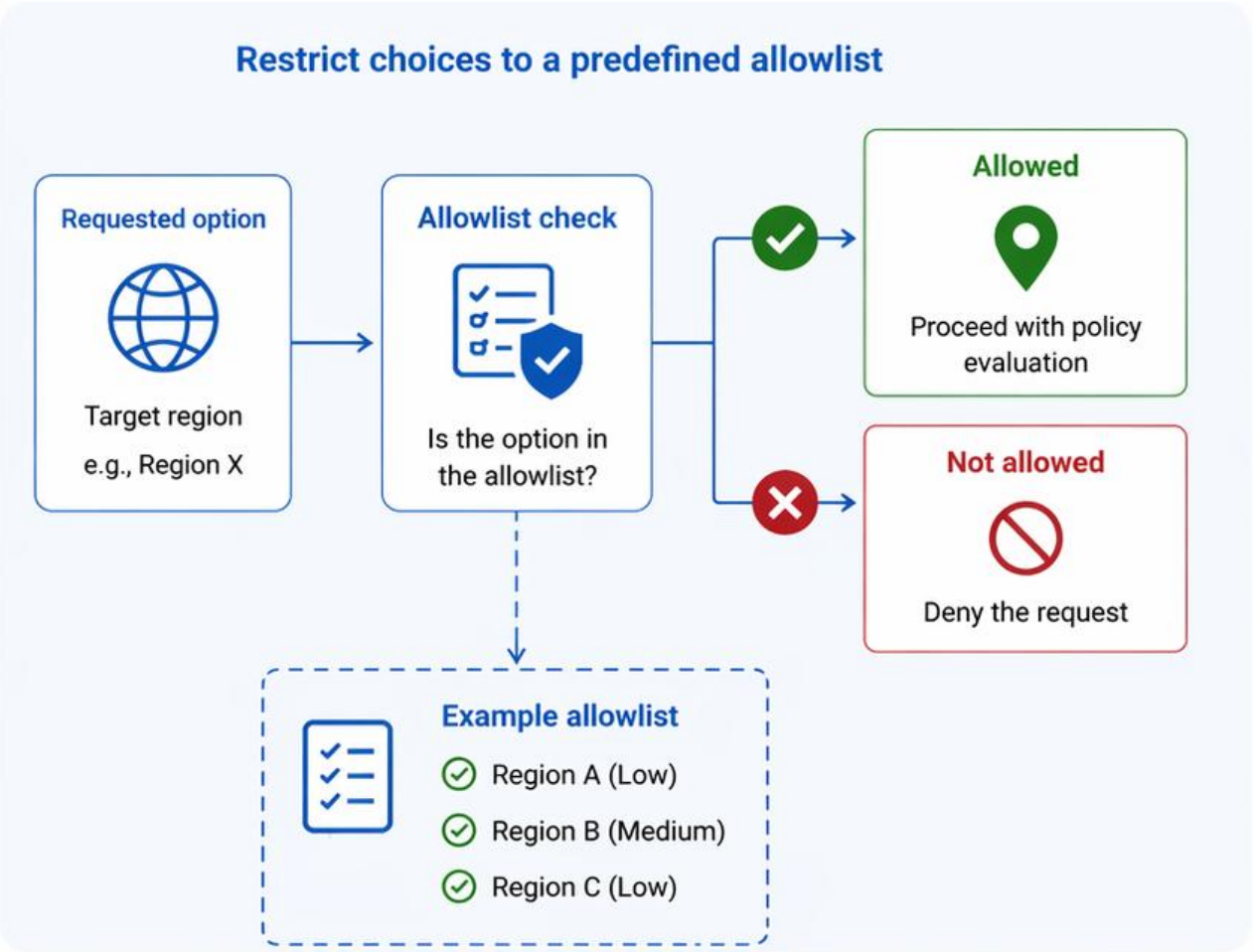
This pattern restricts choices to a predefined allowlist.

In our example

Only low- or medium-carbon regions are allowed for the target workload.

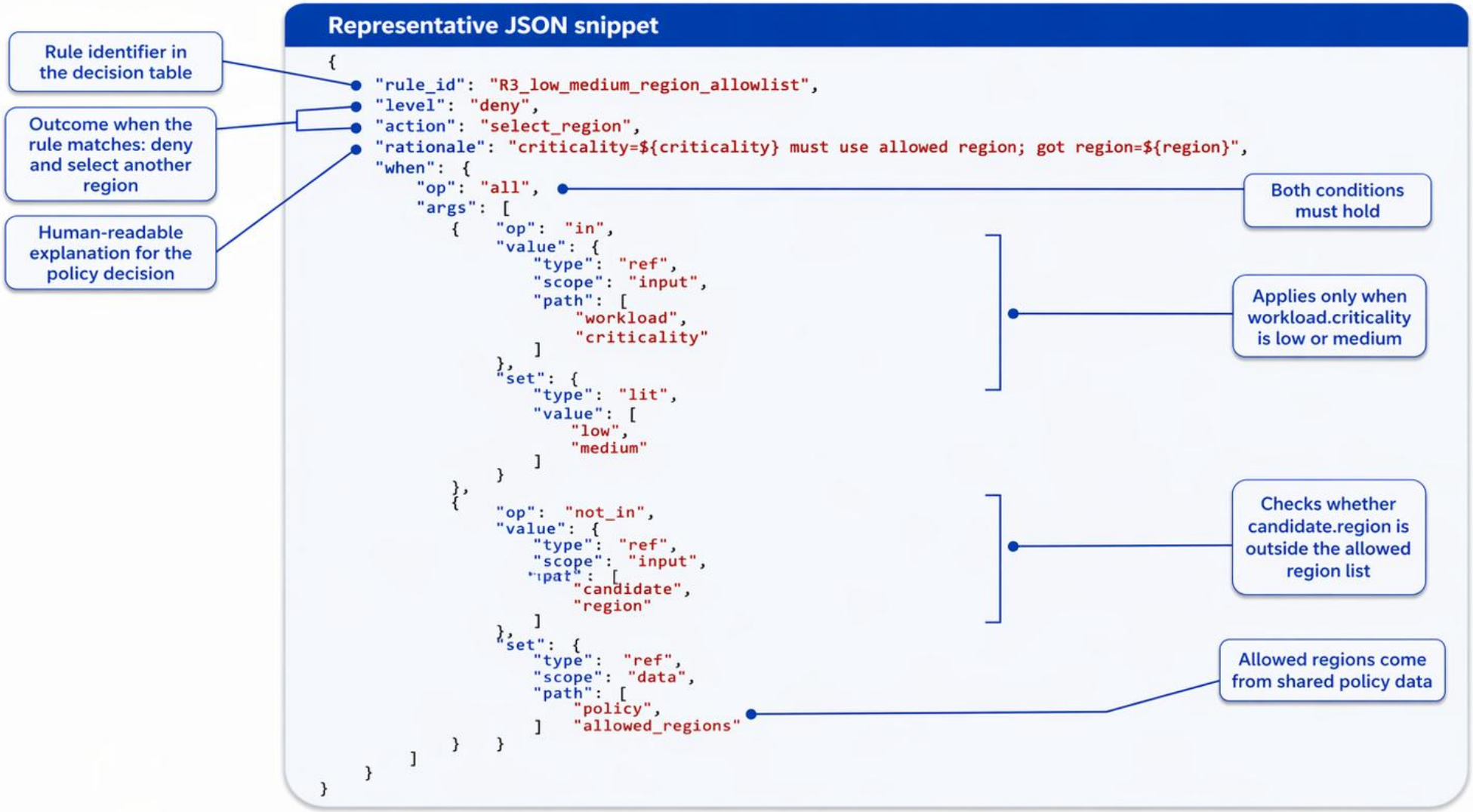
Why it matters

- acceptable choices are made explicit
- review becomes reproducible
- policy updates are easier when the list is separated from evaluation logic



Representative rule: Low/medium region allowlist

Json Snippet: Allowlist



Pattern 4: Exception Rule

What is it

Not every constraint should be absolute. This pattern allows controlled exceptions when additional justification is provided.

How it works

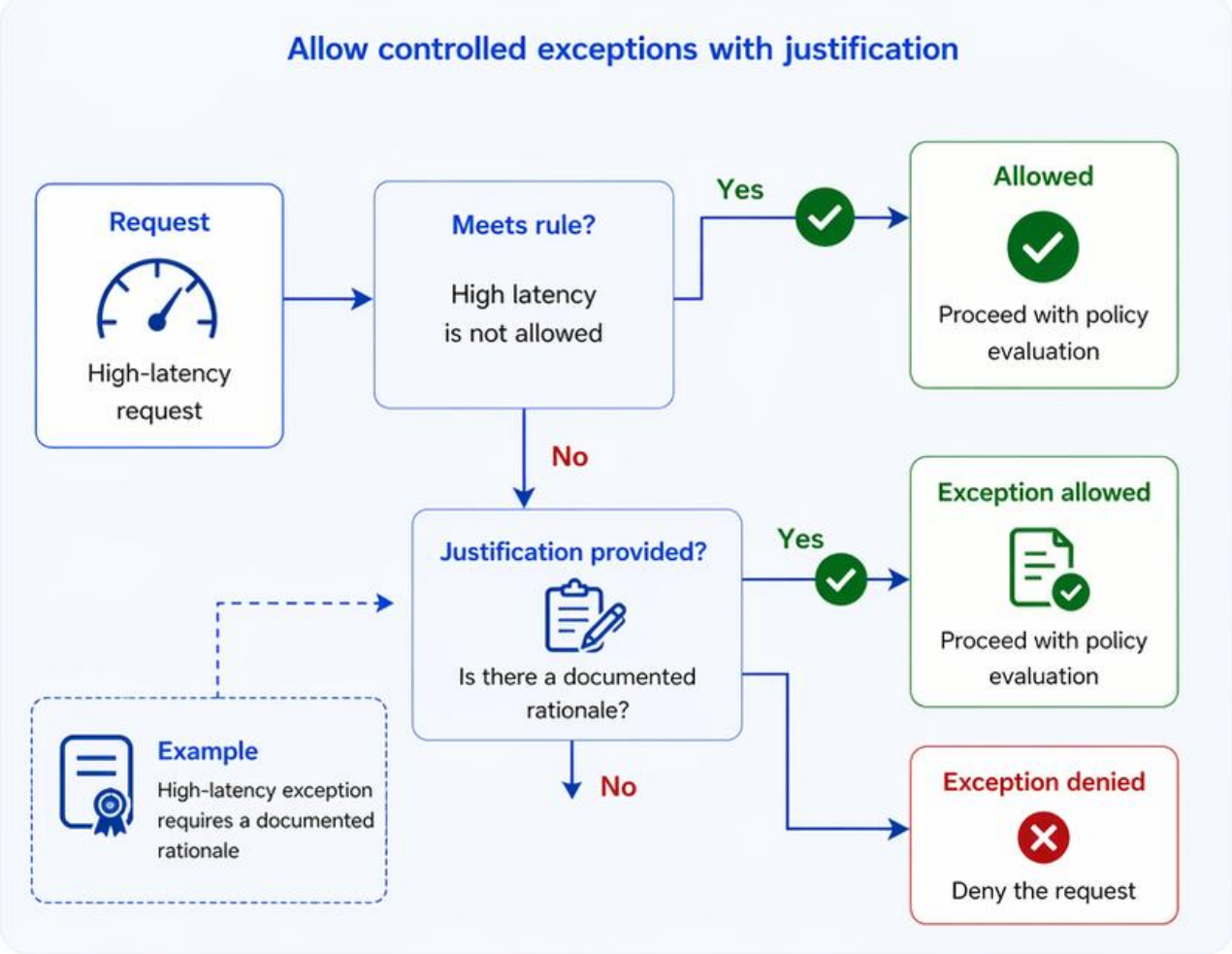
High-latency exceptions are accepted only when the request includes a documented rationale.

In our example

High-latency exceptions are axcepted only when the request includes a documented rationale.

Why it matters

- governance should not block all edge cases
- exceptions remain visible and reviewable
- flexibility is introduced without losing control



Representative rule: High latency exception requires justification

Json Snippet: Exception Rule

Rule identifier in the decision table

Outcome when the rule matches: warn and require justification

Human-readable explanation for the policy decision

Representative JSON snippet

```
{
    "rule_id": "R4_high_latency_exception_requires_justification",
    "level": "warn",
    "action": "require_justification",
    "rationale": "latency_sensitive high/mission uses non-allowed region=${region}; justification required to document tradeoff",
    "when": {
        "op": "all",
        "args": [
          {   "op": "in",
              "value": {
                  "type": "ref", "scope": "input", "path": [ "workload", "criticality" ]
              },
              "set": { "type": "lit", "value": [ "high", "mission" ] }
          },
          {   "op": "eq",
              "left": { "type": "ref", "scope": "input", "path": [ "workload", "riegór-
                        "requirements", "latency_sensitive" ] },
              "right": { "type": "lit", "value": true }
          },
          {   "op": "not_in",
              "value": { "type": "ref", "scope": "input", "path": [ "candidate", "region" ]
                  ,
              "set": { "type": "ref", "scope": "data", "path": [ "policy", "allowed_regions" ] }
                      }
          },
          {
              "op": "missing_or_empty",
              "value": { "type": "ref", "scope": "input", "path": [ "metadata", "justification" ] }
          }
        ]
    }
}
```

All four conditions must hold

Applies only when workload.criticality is high or mission

Checks whether the workload is latency sensitive

Checks whether candidate.region is outside the allowed region list

Allowed regions come from shared policy data

Triggers only when metadata.justification is missing or empty

Pattern 5: Tradeoff Rule

What is it

Cloud governance often involves competing objectives rather than single-metric optimization.

How it works

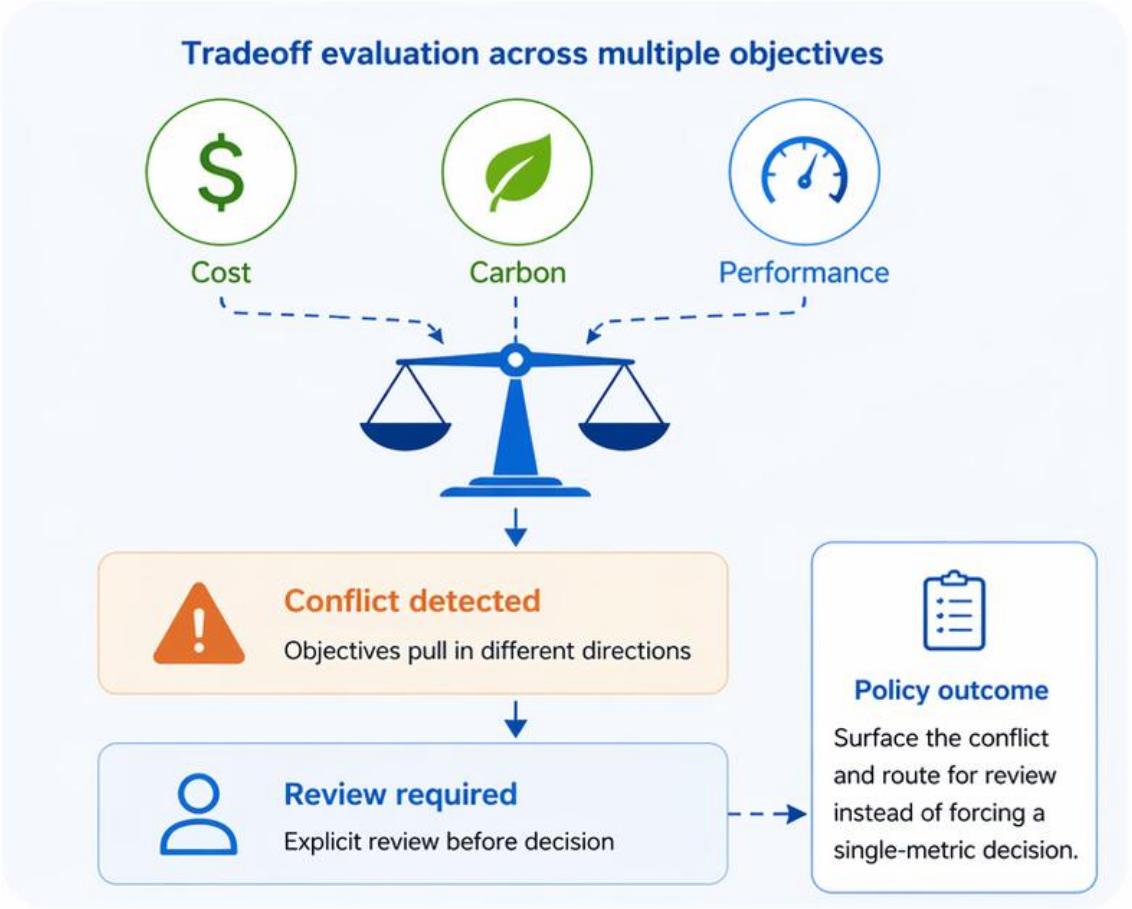
This pattern evaluates tradeoffs across multiple concerns instead of enforcing one objective in isolation.

In our example

The policy requests an explicit review when cost and carbon objectives conflict.

Why it matters

- real decisions involve multiple objectives
- conflicts should be surfaced, not hidden
- review points can be designed into the policy outcome



Representative rule: Tradeoff review between cost and carbon

Json Snippet: Tradeoff Rule

Rule identifier in the decision table

Outcome when the rule matches: warn and request tradeoff review

Human-readable explanation for the policy decision

Representative JSON snippet

```
{
    "rule_id": "R5_tradeoff_review_cost_vs_carbon",
    "level": "warn",
    "action": "tradeoff_review",
    "rationale": "high carbon region=${region} (ci=${ci}) with significant cost delta=${delta}; require explicit cost-vs-carbon tradeoff review",
    "when": {
        "op": "all",
        "args": [
          {
              "op": "present",
              "value": {
                  "type": "ref",
                  "scope": "input",
                  "path": [ "sustainability", "region_intensity_gco2e_per_kwh"],
                  "key": {
                      "type": "ref",
                      } "path": [ "candidate", "region" ]
                  }
              },
              {  "op": "ge",
                  "left": {
                      "type": "ref",  "scope": "input",  "path": ["sustainability", "region_
                      intensity_gco2e_per_kwh" ] { "type": { "type"a_treshof",
                      "path": [ "candidate", "canddan", "region"]
                  },
                  "right": { "type": "ref",  "scope": "data",
                  "path": [ "policy",  "tradeoff",  "carbon_intensity_high_threshold_gco2e_per_kwh" ]
              },
              {  "op": "le",
                  "left": {
                      "type": "ref",  "scope": "input",  "path": [ "cost",
                      "estimated_effective_cost_delta" ]
                  },
                  "right": {
                      "type":  "neg",
                      "value": {
                          "type":  "ref",  "scope":  "data",  "scope":  "data",
                          } "path": [ "policy",  "tradeoff",  "cost_delta_significant_tthreshold"  ]
              } }
          ]
      }
}
```

All three conditions must hold

Checks that carbon intensity data exists for candidate.region

Compares region carbon intensity against the shared high-carbon threshold

Checks whether the cost delta is significantly lower than the threshold

Uses shared tradeoff thresholds from policy data

Agenda

1. Introduction
2. Challenge: The Gap from Guidance to Decision-Making
3. Approach: Policy-as-Code with OPA
4. System Design and Architecture
5. Policy Patterns and Representative Rules
6. Implementation and Operations
7. Demonstration
8. Future Roadmap and Takeaways

Example: OPA in CI with Jenkins

A Jenkins pipeline converts artifacts into normalized JSON, runs OPA, and returns review-ready decisions.

OPA runs inside CI • Jenkins executes the same checks automatically and repeatedly

Simplified Jenkins pipeline

```
pipeline {
  agent any
  stages {
    stage('Normalize input') {
      steps {
        sh 'python tools/normalize.py > build/input.json'
      }
    }
    stage('OPA check') {
      steps {
        sh 'opa eval --data policy \
                --input build/input.json \
               "data.guardrails.results" > build/opa.json'
      }
    }
    stage('Publish result') {
      steps {
        sh 'python tools/render_review.py build/opa.json'
      }
    }
  }
}
```

What Jenkins returns

```
{
  "decision": "allow / warn / block",
  "rule_id": "R2_dev_requires_schedule_tag",
  "rationale": "required schedule tag is missing for dev",
  "follow_up": "add the schedule tag or request exception"
}
```

- runs the same policy checks on every PR or build
- can archive input.json and opa.json for auditability
- can fail the build only on block, while warn remains reviewable

What Result comes back from OPA in CI?

Jenkins runs OPA and returns a simple overall status plus review-ready decisions.

Returned JSON structure

```
{
   "status": "allow | warn | block",
   "decisions": [
      {
         "rule_id": "R2_dev_requires_schedule_tag",
         "level": "deny | warn",
         "action": "require_schedule",
         "rationale": "required schedule tag is missing for dev"
      }
   ]
}
```

1. `status` = overall outcome used by CI
2. `decisions[]` = rule-by-rule details for review
3. `action` / `rationale` = what to do next

How to read it

1. **allow:** no blocking or warning decisions
2. **warn:** review is needed, but the pipeline may continue
3. **block:** the build should fail or stop for review

Example A: warn

```
{
   "status": "warn",
   "decisions": [
      {
         "rule_id": "R1_missing_carbon_intensity",
         "level": "warn",
         "action": "collect_data"
      }
   ]
}
```

Missing carbon-intensity data -> continue with review follow-up.

Example B: block

```
{
   "status": "block",
   "decisions": [
      {
         "rule_id": "R2_dev_requires_schedule_tag",
         "level": "deny",
         "action": "require_schedule"
      }
   ]
}
```

Dev resource has no schedule tag -> block until fixed.

Example C: allow

```
{
   "status": "allow",
   "decisions": []
}
```

No rule triggered -> clean result.

The runtime returns status plus decisions[]; in this demo, representative outcomes are allow / warn / block.

Readable OPA result in Jenkins

OPA returns JSON internally; Jenkins can render a review-friendly dashboard.

1 OPA output


```
{
  "status": "partial",
  "summary": {
    "allow": 1,
    "warn": 2,
    "deny": 1
  },
  "decisions": [
    {
      "rule_id": "R1_missing_carbon_intensity",
      "level": "warn",
      "action": "collect_data",
      "rationale": "Carbon intensity data is
         missing for the selected region."
    },
    {
      "rule_id": "R2_dev_requires_schedule_tag",
      "level": "warn",
      "action": "require_schedule",
      "rationale": "Unscheduled dev workload
         may cause cost and carbon spikes."
    },
    {
      "rule_id": "R5_tradeoff_review_cost_vs_carbon",
      "level": "warn",
      "action": "tradeoff_review",
      "rationale": "Cost is high relative to
         carbon benefit; request review."
    }
  ]
}
```


2 Jenkins post-processing

- Shared library / parser
- Transform JSON (parse & map)
- Build summary (UI model)

Convert machine output into human-friendly data

3 Jenkins GUI

Overall Status

ALLOW	WARN	BLOCK
1	2	1

Triggered Rules

Rule ID	Level	Action	Rationale (summary)
R1_missing_carbon_intensity	WARN	collect_data	Carbon intensity data is missing for the selected region.
R2_dev_requires_schedule_tag	DENY	require_schedule	Unscheduled dev workload may cause cost and carbon spikes.
R5_tradeoff_review_cost_vs_carbon	WARN	tradeoff_review	Cost is high relative to carbon benefit; request review.

Recommended Next Actions

- Collect missing data
- Add schedule tag
- Request review

Why (Rationale)

Unscheduled dev workload may cause cost and carbon spikes. Adding a schedule tag helps enforce guardrails.

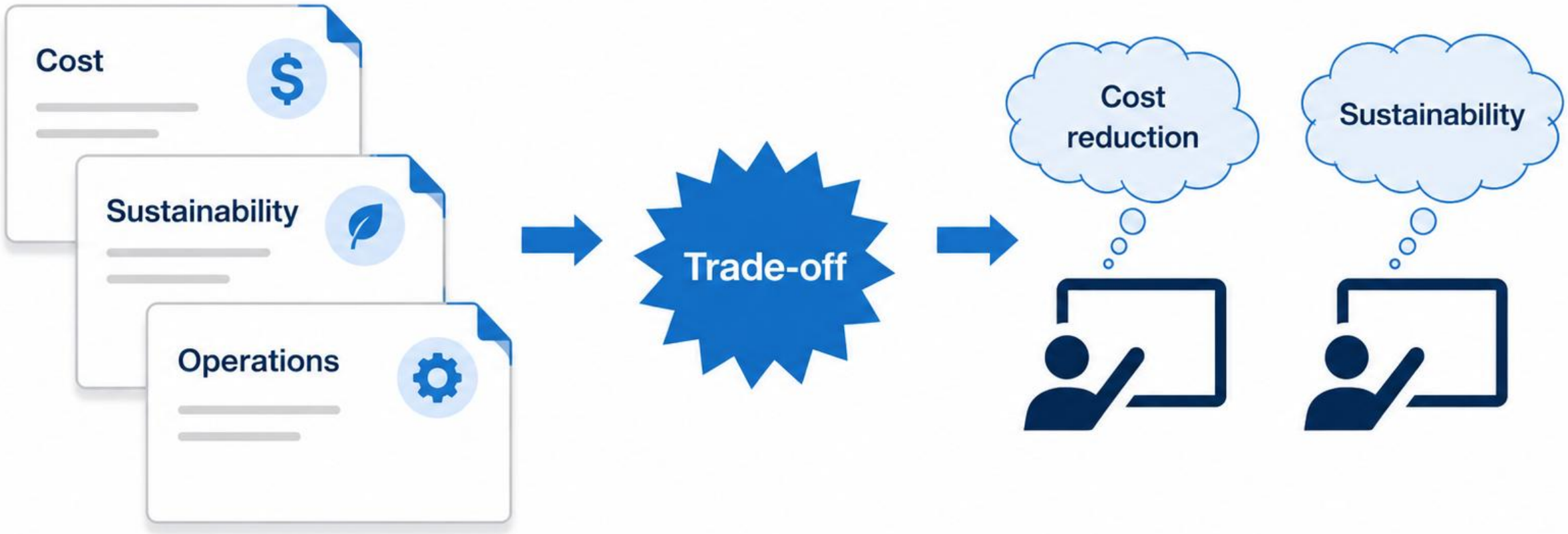
Implementation idea: OPA evaluates policy -> Jenkins captures JSON -> a plugin, shared library, or HTML report renders the results as a dashboard.

Agenda

1. Introduction
2. Challenge: The Gap from Guidance to Decision-Making
3. Approach: Policy-as-Code with OPA
4. System Design and Architecture
5. Policy Patterns and Representative Rules
6. Implementation and Operations
7. Demonstration
8. Future Roadmap and Takeaways

From Guidance to Decisions Is Not Straightforward

Translating sustainability and FinOps guidance into concrete cloud decisions is not straightforward. Guidance often mixes abstract principles with operational details, so consistent manual review becomes difficult.



What the Decision Table Actually Decides

The rule catalog encodes both blocking conditions and review-triggering conditions.

Representative rules

- **R1** Missing carbon intensity → warn
- **R2** Dev workload without schedule tag → block
- **R3** Low/medium criticality in non-allowed region → block
- **R4** High/mission + latency-sensitive + non-allowed region without justification → warn
- **R5** High-carbon region with significant cost saving → warn

JSON rule example

```
{
  "rule_id": "R2_dev_requires_schedule_tag",
  "level": "deny",
  "action": "require_schedule",
  "when": { ... }
}
```

Rules live in JSON.
Rego acts as the evaluator.

Meaning of outcomes

warn = review required

block = must fix before merge

Live demo next:
bash scripts/demo.sh

OPA in CI with a Two-Layer Policy Design

Keep the evaluator small and move policy rules into a JSON Decision Table.

1 Input to policy evaluation

JSON

- normalized policy input JSON
- workload / candidate / cost / metadata
- generated by CI normalization
- stable input schema

2 Two-layer policy design

Layer 1: JSON Decision Table

- R1 missing carbon intensity → warn
- R2 dev without schedule → deny
- R3 region allowlist → deny
- R4 exception requires justification → warn
- R5 cost vs carbon tradeoff → warn

policy data / rule catalog

Layer 2: Rego evaluator

- generic rule evaluation logic
- reads the JSON rule catalog
- returns status + decisions[]

small reusable engine

3 Output from OPA in CI

```
{
  status: "warn",
  triggered rules: [
    "R1_missing_carbon_intensity",
    "R5_tradeoff_review_cost_vs_carbon"
  ],
  action examples: [
    "collect_data",
    "tradeoff_review"
  ]
}
```

OPA eval | Conftest / CI gate | Jenkins report

Two-layer design separates policy content from evaluation logic.

JSON rules can evolve without rewriting the Rego evaluator.

Why this design works

- easier policy maintenance
- less Rego expertise required
- reusable evaluator logic
- reviewable policy changes in JSON

Before OPA: From Pull Request to Normalized Policy Input

How PR metadata and IaC changes become the validated JSON input evaluated by OPA

Automated by Jenkins

1. User / Pull Request

feat: update schedule, tags and region — Open

Conversation 0 | Changes 2 | Checks 0

main.tf +9 -3

```
@@ resource "aws_instance" "app" {
-   region        = "us-east-1"
+   region        = "us-west-2"
    ami           = "ami-0abc1234def567890"
-   instance_type = "t3.medium"
+   instance_type = "t3.large"
+   tags = {
+     Environment = "prod"
+     Owner       = "platform-team"
+   }
```

IaC change (Terraform)

```
region              = "us-west-2"
instance_type       = "t3.large"
tags.Environment    = "prod"
tags.Owner          = "platform-team"
schedule            = "rate(1 hour)"
```

Submitted by @dev-team

2. CI normalization

- Read PR metadata + IaC fields
- Map to one stable schema
- Validate before policy input

policy input JSON

3. What OPA reads

```
{
  "workload.env": "prod",
  "workload.criticality": "high",
  "requirements.latency_sensitive":true,
  "candidate.region": "ap-northeast-1",
  "cost.estimated_effective_cost_delta":
   -30
}
```

4. Normalized input ready

policy-input.json

schema-validated
ready for OPA evaluation

How to Read the Output: status + decisions[]

The policy output is both machine-actionable and human-reviewable.

Output contract

Example output

```
{
  "status": "warn",
  "decisions": [
    {
      "rule_id": "R5_tradeoff_review_cost_vs_carbon",
      "level": "warn",
      "action": "tradeoff_review",
      "rationale": "high carbon region
                with significant cost savings"
    }
  ]
}
```

Aggregated status and CI gate view

allow
= Proceed
CI gate: **PASS**

warn
= Review needed
CI gate: **WARN**

block
= Must fix
CI gate: **FAIL**

status summarizes the whole input; decisions[] lists the triggered rules.

Three layers of meaning

1. Rule-level decision

- level belongs to each triggered rule in decisions[]
- level = warn → review item
- level = deny → blocking issue

2. Aggregated status

- status is the result for the whole input
- allow / warn / block

3. CI gate view

- CI tools can map the aggregated result for pipeline use
- allow → PASS
- warn → WARN
- block → FAIL

Three Views of the Same Policy Evaluation

After OPA evaluates the normalized input, only the presentation changes.

Shared policy logic

Decision Table + Rego evaluator

OPA eval view

Inspect the structured decision

```
{
  "status": "warn",
  "decisions": [
    { "rule_id": "R1_missing_carbon_intensity",
      "level": "warn",
      "action": "collect_data" }
  ]
}
```

- rule_id
- level
- action
- rationale

Purpose: transparency of the decision logic

Conftest / CI gate view

View the same result as CI pass / warn / fail

```
WARN  inputs/01-missing-intensity.json
FAIL  inputs/02-dev-no-schedule.json
FAIL  inputs/03-low-disallowed-region.json

20 tests, 15 passed, 3 warnings, 2 failures
```

- deny — FAIL
- warn — review item
- Natural presentation for CI users

Purpose: show CI gate behavior

Jenkins report view

Convert the same result into a human-readable GUI

Overall status	WARN
Triggered rules	R1 R3 R5
Rationale	
Next actions	

Purpose: review-friendly visualization

Conftest is not a separate policy engine

OPA eval, Conftest, and the Jenkins report use the same Decision Table and the same Rego evaluator.

Takeaway

OPA eval shows what the policy decided, Conftest shows how it behaves in CI, and Jenkins shows how people read the result.

What This Demo Proves — and What It Does Not Yet Prove

The demo proves the CI policy core, not the full enterprise production workflow.

Truth boundary

What the demo proves today

- Guidance can be encoded as a Decision Table
- OPA can return status + decisions[]
- CI can normalize request / IaC data into policy input
- Results can be shown in a Jenkins-readable dashboard
- A minimal Terraform-derived path is demonstrable offline

What still needs real operations work

- Broad Terraform / Kubernetes / CloudFormation support
- Full approval workflow and exception lifecycle
- Real-time carbon / power integration
- ServiceNow / Jira / Backstage integration
- Production-grade extractor and policy ownership

The demo proves the CI policy core, not the full production workflow.

Extractor of Source (IaC, Terefrom)

NTT

Source-specific artifacts → stable policy input

The extractior is a tested mapping layer, not AI inferoufence in the CI gate.

1. Source artifacts

Terraform plan
terrafrorm plan JSON, tags, variables, outputs, provider region, ...

Kubernetes manifest
labels, annotations, namespace, nodeSelector, topology, ...

CloudFormation template
Parameters, Tags, Mappings, Outputs, stack region, ...

Request form / ticket
env, business unit, justification, target region, cost estimate, ...

Service catalog metadata
product, tier, environment, owner, target region, ...

IaC format is not the policy interface.

2. Extractor design

1. Define required policy fields
2. Create a source-to-canonical mapping table
3. Fix priority order when multiple sources contain the same field
4. Implement deterministic extractor code
5. Validate generated JSON
6. Test expected policy decisions

Example mapping (partial)

Canonical field	Source example	Priority order
workload.env	PR form, tag, variable	PR form → tag → variable
candidate.region	request field, output, provider region	request → output → provider
cost.estimated_ effective_cost_delta	cost estimate, PR metadata	estimate → metadata
metadata.justification	PR form, ticket, tag	PR form → ticket → tag

Priority order = which source wins?
Ensures deterministic and auditable extraction.

3. Canonical policy input

All extractors emit the same stable schema used by OPA and Decision Table.

```
{
  "workload": {
    "env": "prod",
    "criticality": "high",
    "requirements": {
       "latency_sensitive": true
    }
  },
  "candidate": {
     "region": "ap-northeast-1"
  },
  "cost": {
    "estimated_effective_cost_delta": -30
  },
  "metadata": {
    "justification": "lower latency
                     for Japan users"
  }
}
```

Different extractors, same OPA-facing input.

Quality guardrails

- Schema validation (types, required, enums)
- Fixture tests for expected decisions
- Fail-fast on missing critical fields
- Field provenance for auditability

About AI

- AI may help draft mappings or extractor code.
- AI should not freely generate CI gate input.
- The gate path must be deterministic and auditable.

Demo scope

- This demo implements the Terraform path as one concrete example.
- Production use adds tested extractors for the organization's real sources.

Many sources, one goal: extract decision-relevant fields → stable policy input → consistent, explainable evaluation.

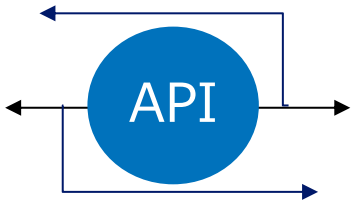
Agenda

1. Introduction
2. Challenge: The Gap from Guidance to Decision-Making
3. Approach: Policy-as-Code with OPA
4. System Design and Architecture
5. Policy Patterns and Representative Rules
6. Implementation and Operations
7. Demonstration
8. Future Roadmap and Takeaways

Future Roadmap

Several Data Integration using API

- Access actual data such as carbon intensity via an API
- Integrate with various workflows via an API

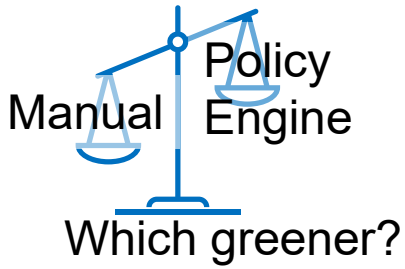


Sugession

- Suggest actions to reduce emissions and costs

Make cloud operation greener

- Promote sustainability by operational improvement using OPA



Takeaways

Rule via Code

- Manage guidelines not as document,
  but as machine-readable policies to execute in CI
- Use rule patterns as a start point to define your own
  sustainability and cost policies

Handle imperfection

- Don’t let conflicts stop decision-making
- Use Warn to identify gaps and drive the improvement cycle

Embedded Sustainability

- Promote effective sustainability by integrating it into
  standard FinOps governance such as policies and guardrails