National College of Ireland

Healthcare-Grade Cross-Cloud Policy Delivery: Signed OPA Bundles as OCI Artifacts on AWS, Azure & GCP

MSc Research Project
MSc in Cloud Computing

Lakshmi Prasanna Maddipoti
Student ID: 23409029

School of Computing
National College of Ireland

Supervisor: Sean Heneey

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Lakshmi Prasanna Maddipoti
Student ID:	23409029
Programme:	MSc in Cloud Computing **Year:** 2025
Module:	Research in Project
Supervisor:	Sean Heneey
Submission Due Date:	28/01/2026
Project Title:	Healthcare-Grade Cross-Cloud Policy Delivery: Signed OPA Bundles as OCI Artifacts on AWS, Azure & GCP
Word Count:	7297 **Page Count:** 22

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: M.L.Prasanna

Date: 28/01/2026

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Healthcare-Grade Cross-Cloud Policy Delivery: Signed OPA Bundles as OCI Artifacts on AWS, Azure & GCP

Lakshmi Prasanna Maddipoti
23409029

Abstract

Healthcare organisations increasingly operate workloads across multiple public clouds to improve uptime, cost efficiency, and regional coverage. In such environments, access-control and compliance policies must remain consistent wherever protected health data is processed, while also being tamper evident and auditable to support regulatory expectations associated with HIPAA/HITECH and GDPR. In practice, however, policies are often distributed through manual file copying, informal syncing scripts, or mutable tags, which creates configuration mismatch over time and introduces time-of-check/time-of-use risk: the same reference can silently point to different policy content over time. This thesis examines whether treating policy-as-code as a primary supply-chain artifact can provide a provider-neutral foundation for healthcare-grade policy delivery across AWS, Azure, and GCP. It proposes and implements a reference workflow in which Open Policy Agent (OPA) policy bundles are packaged in a repeatable way, tied to immutable content digests, published through standard OCI-compatible registries, and protected by cryptographic signatures attached as associated artifacts. A CLI-first toolchain runs the pipeline end-to-end build, sign, publish/discover, verify, and execute while emitting machine-readable evidence suitable for audit import. A prototype implementation demonstrates verify-before-use enforcement gating (fail closed), structured evidence generation, and tamper detection via deliberate small change of the bundle payload and signature material. The results support the claim that digest-tied, signed policy artifacts can reduce mismatch over time, strengthen integrity guarantees, and improve audit readiness without cloud-specific policy distribution mechanisms. The thesis concludes with an extensible evaluation framework and practical guidance for expanding the approach into full cross-cloud registry deployments and operational oversight workflows.

Keywords: Policy-as-Code, Open Policy Agent (OPA), OCI artifacts, container registries, content digests, Sigstore/Cosign, supply chain security, tamper-evidence, origin record, auditability, multi-cloud compliance, healthcare security.

1 Introduction

1.1 Motivation

There is a common practice among healthcare providers, payers, and research networks to run workloads over AWS, Azure, and GCP in an attempt to balance ability, uptime, and expenses. Because systems and data are transferred across (and among) regions and vendors, policies controlling access to secure health information must be consistent, that detects tampering, and auditable. HIPAA/HITECH and the GDPR regulations increase the level of these expectations beyond the preference levels to the required compliance level. Practically, though, the distribution of policies is usually based on copying source files, pinning mutable image tags, or custom synchronisation scripts patterns which cause mismatch over time to become likely and origin difficult to establish.

At the same time, the cloud-native ecosystem has also been raising the standards of OCI to package, address, and transport artifacts and popular signing and origin tools. This is encouraging, and may indicate a workable future: policy-as-code should be treated as a signed and immutable entity which can be moved by common registries and verified prior to use, regardless of the cloud which serves the bytes.

1.2 Problem Statement

There is no widely documented, cloud-neutral workflow for distributing and verifying policy-as-code across AWS, Azure, and GCP that both

- preserves a machine-verifiable chain of custody
- fits naturally into the command-line-centric practices of cloud operators

Without such a workflow, organisations face inconsistent configs, increased audit burden, and raised risk of undetected policy substitution or rollback.

1.3 Research Question

The aim is to examine a provider-neutral approach for healthcare-grade, cross-cloud policy distribution that is digest-tied, signed, and verifiable before runtime. And so the research question becomes:

To what extent does packaging and distributing OPA policy bundles as digest-pinned, Cosign-signed OCI artifacts improve tamper-evidence, reduce policy mismatch over time, and increase auditability of policy enforcement across multi-cloud healthcare workloads (AWS, Azure, GCP)?

1.4 Research Objectives

In this thesis, "to what extent" is assessed using measurable indicators:

- Tamper-evidence, measured by whether signature verification fails for any modified payload/signature and for tag/digest mismatches.
- Auditability, measured by whether the pipeline emits machine-readable evidence linking enforcement outcomes to a specific signed digest.
- Portability, measured by whether the same artifact shape and verification logic work across OCI registries without cloud-specific distribution mechanisms.

Treat policy as a first-class OCI artifact. Package OPA bundles in a repeatable way, sign the registry manifest digest (what consumers actually pull) with Cosign/Sigstore, attach origin via OCI referrers, and verify before runtime.

1.5 Scope and Novelty

This study focuses on policy distribution and verification rather than policy authoring. It examines the portability of a registry-centred model across the three largest public clouds and the suitability

of digest-tied, signed artifacts for healthcare audit needs. The work assumes familiarity with policy-as-code and container registries and treats legal mapping as engineering-oriented (i.e., identifying technical controls that support compliance) rather than as formal legal advice. No patient data is required for example policies and inputs are synthetic. The thesis seeks to:

- reduce inconsistent configs by replacing mutable tags with immutable digests as the unit of trust
- provide machine-readable evidence (digests, signatures, origin links) compatible with audit workflows in healthcare
- preserve operator usability via a concise, CLI-first surface that is scriptable, accessible, and automation-friendly
- offer a cloud-neutral pattern that can be replicated across ACR, ECR, and GAR without bespoke tooling per provider

2 Literature Review

2.1 Multi-Cloud Compliance Challenges

Multi-cloud strategies are becoming popular in organizations, as it helps to optimize the cost, availability, and uptime. But, this variety makes it difficult to govern and manage compliance (Webster et al., 2023). It is challenging to keep consistent access control policies between AWS, Azure, and GCP, since each of the three platforms has different controls and APIs. The distributed nature of multi-cloud environments increases the difficulty of achieving consistent regulatory requirements (e.g. HIPAA, GDPR) on the access and audit trail of the data (Webster et al., 2023). Manual processes, one-off scripts, and isolated cloud-specific tools are long used traditional methods that result in configuration mismatch over time, errors, and late detection of policy violations (Webster et al., 2023). This inconsistency may lead to breach of compliance, data breaches, and audit failure. These loopholes have led to the study of automated, unified policy enforcement systems across many clouds. The initial practices in the industry saw the introduction of cloud-native compliance tools (e.g. AWS Config Rules, Azure Policy), which are platform-specific and have no neutral control plane across multiple clouds. Therefore, it is evident that a cloud-neutral solution is necessary that maintains a machine-enforceable chain of custody of the policies with a mixed infrastructure.

2.2 Policy-as-Code and Open Policy Agent (OPA)

Policy-as-Code (PaC) has also become a solution to these issues, with policies being treated as code: versioned, testable, and automated (Gazitt, 2022). Policies are written in a high-level language and processed by engines instead of informal documents or console settings and allows policies to be continuously complied with. Initial PaC systems such as HashiCorp Sentinel and product-specific rule engines were useful to automate checks and were frequently product-locked or cross-platform (Webster et al., 2023). With the release of Open Policy Agent (OPA), a major breakthrough in this field was made. OPA is a works across vendors, general-purpose, open-source policy engine that is intended to be deployed anywhere (Webster et al., 2023). It has a rule-based

language (Rego) to specify policies, which encourages transparency, versioning, and audit trail of decisions. More importantly, OPA is compatible with a wide range of systems (Kubernetes clusters, microservice APIs, CI/CD pipelines, Terraform, etc.), and a single policy definition can be applied in different environments. Research has shown OPA to be effective in other areas - such as utilizing Kubernetes network security policies (Smith et al., 2020, as cited in Webster et al., 2023) and testing infrastructure-as-code (Li and Chen, 2021, as cited in Webster et al., 2023). OPA allows the enforcement of policies in a consistent manner, by separating policy logic and application code (or cloud-specific configs). Notably regarding multi-cloud application, cloud-neutral nature of OPA implies that the same Rego policies can be used to assess resources in AWS, Azure or GCP with relevant integration. Recent studies suggest that OPA can be used to implement automated compliance across AWS, Azure, and GCP and helps reduce compliance mismatch over time and enhance security posture in mixed cloud environments (Webster et al., 2023). OPA fundamentally offers the means of standardizing the implementation of policies; the only remaining puzzle is how to allocate and trust these definitions of policies to clouds in a fingerprint-free fashion.

2.3 Distribution of Policy Bundles as OCI objects

OPA permits the aggregation of policies (and related data) into a bundle - simply a file archive (tarball) which can be loaded by OPA. Practically, such policy bundles have been distributed across environments through file and container image copying in many organizations. This is a fragile design: file copying can introduce version mismatch over time and the use of changeable image labels (e.g. :latest or semantic labels) brings about time of check/time of use (TOCTOU) errors - a label may point to different content at different times, and thus a policy may be changed or rolled back without the label noticing. Now, it is considered best practice in the cloud-native industry to store objects using immutable references (content digests) in order to provide consistency. As an example, the image SHA-256 digest is used in container deployments to ensure the same image is downloaded each time, and the uncertainty of moving tags is removed (IBM, 2020). Making policy objects under this principle implies policy bundles are labeled with cryptographic digest instead of names that can be changed over time or altered by malicious intentions.

One more recent development that can help such a style is the treatment of policy bundles as OCI (Open Container Initiative) objects. OCI image and distribution specifications (v1.0 and the more recent v1.1) present a definition of a standardization of packaging content and moving it using container registries - however, this standard is not exclusive to container images of old. It is capable of capturing arbitrary content as a special type of object. Organizations can use the existing container registry ecosystem to store and share policies by packaging an OPA bundle as an OCI object (essentially an image layer or tarball with a manifest). Engineers have observed several advantages to this method, such as cannot be changed (digest-based storage with digests), version tagging, and a standard push/pull protocol (Gazitt, 2022). In short, policy-as-container-image transforms policy distribution into the identical issue as software object distribution, which already has well-developed infrastructure.

OCI object support has been growing quite rapidly within the past couple of years, so it is a genuinely cross-cloud solution. Amazon provided its Elastic Container Registry with the ability to store arbitrary objects (not just Docker images) and showed the storage of OPA policy bundles in ECR as early as 2020 (Deshpande, 2020). In 2022, Docker Hub introduced the ability to push custom types of objects (e.g. Helm charts, WASM modules, OPA bundles, etc.) as OCI images (Gajdos, 2022). An early implementer of the OCI 1.1 draft spec was the Azure Container Registry (ACR) of Microsoft, which in 2024 declared its support of OCI objects, and specifically OPA bundles as a kind of object (Zhou, 2024). In like manner, Google object Registry allows any content in OCI image format in its repositories. This broad use implies that an OPA policy bundle can be bundled once and stored in any general registry (regardless of whether it is AWS, Azure, GCP, or even on-premises) and used in the same manner. It offers work across vendors portability, and thus the requirement of a cloud-neutral workflow.

To ease the use of these bundles, tools such as ORAS (OCI Registry As Storage) have been created as generic clients of OCI objects. ORAS can push and pull files as objects to registries of custom media types, and has been demonstrated by both the AWS and Azure teams to use in demos to work with OPA bundles (Deshpande, 2020; Zhou, 2024). In parallel, OPA itself introduced native support for pulling bundles from OCI registries (since OPA v0.40, it can treat an OCI image as a bundle source) – effectively aligning with this distribution model (Gazitt, 2022). By using OCI registries as the single source of truth for policy bundles, organizations gain a uniform distribution mechanism across all clouds, and by referencing bundles via their digest (SHA-256), they ensure that what each environment pulls is exactly what was published and approved, with no mismatch over time. In summary, the literature and industry developments strongly advocate for treating policy-as-code as a primary object in the cloud-native supply chain, analogous to an application container or Helm chart (Gajdos, 2022; Gazitt, 2022).

2.4 Tamper-Evident Signatures and Supply Chain Security

It is enough to store policies in a central registry to address the distribution issue, but we need to ensure integrity and authenticity also - in particular, regulated healthcare workloads where tamper-evidence and audit trails are a requirement. Software supply chain security has gained critical importance over recent years, due to high-profile breaches (e.g. SolarWinds) which affected upstream objects. According to Newman et al. (2022), the number of attacks on weak areas of the software pipelines increases, and that all objects should be provided with strong origin and signing. This gave birth to the Sigstore open-source object signing and verification suite in the cloud-native ecosystem. Tooling provided by Sigstore (like Cosign) enables developers to sign container images (or any other OCI object) in a clear manner, without the hassles of operating key management (Newman et al., 2022). Cosign helps short-lived key-based keyless signing and a public public log, which reduces the cost of adoption of cryptographic signing. This technology has been quickly taken up - one example is that Kubernetes now signs its release objects with Sigstore (Gazitt, 2022) - and is the logical choice when one wants to sign OPA policy bundles.

Tamper-evidence is a requirement that is addressed by signing policy bundles. Signing the policy bundle manifest digest (i.e. the actual hash of the bundle content) will guarantee that any alteration in the bundle (even a single bit) would cause the signature to fail. Cosign signs have signatures as independent OCI objects identified by the digest of the object. The association was initially performed conventionally (e.g. Cosign v1 could perform a signature blob with a special tag such as sha256-<digest>.sig). This has however been standardized by the OCI community through the OCI Referrers API in the v1.1 spec (Berry, 2024). Referrers feature gives one object official reference to another as its subject. It implies that an OCI registry can store a graph of associated objects: e.g. an image may have references to its signature, SBOM, evidence, and so on all to the immutable digest of the image (Berry, 2024). The advantage is a better structured and safe origin tracking: signatures are carried around with their object, and may be found or duplicated along with it, without incurring naming hacks. According to Quiana Berry (2024), in this way, the tag namespace will remain clean and unlinked metadata will not exist: clients can also query a particular digest to retrieve all the attached security objects (signatures, scan reports, policy files, etc.). OCI referrers enable us to place an evidence or origin record signed by OCI referrers onto the policy object itself in the registry. This will give a verifiable chain: an auditor can draw up the policy bundle digest and can get its signature and origin (who built it, when, using what source) in a standard form (Berry, 2024).

It may be mentioned that the authors of OPA have also provided OPA with some signing functionality in its bundle format (OPA can generate and verify a .signatures.json file inside the bundle using keys or JWTs - see OPA Documentation, 2023). However, in practice, external signing tools such as Cosign can be a more collaborative solution - they use well-vetted cryptography and can be used with public logs (Sigstore) and can be used with any type of object. As Gazitt (2022) notes, policies are equally important as code or container images within a system, hence they must be signed and checked numerically. In a proposed secure workflow a policy is packaged as an OCI image, and when signed by developers (e.g. cosign sign <repo>/policy-bundle@sha256:...) and published. Next, the signature will have to be confirmed against a known public key or Sigstore certificate before enforcing or deploying any policy, so that the bundle remains unaltered (Gazitt, 2022). The system will only load the policy into OPA when the signature is valid. This pre-use check is comparable to an admission controller in Kubernetes verifying an image signature prior to it being able to execute - in this case, it ensures that no malicious or stale policy is being added to the policy engine. Each of these signature verification operations may also provide machine-readable results (pass/fail, signer identity, timestamp), and these may be used to make audit trails. Overall, the current trends in container security - digest- based storage, digital signatures, and OCI reference types - offer the components of tamper- evident, auditable policy distribution across clouds. The implementation of them is firmly supported in the literature to comply with the requirements: e.g., the secure supply chain framework by Microsoft also suggests keeping signatures and evidence together with the objects in the registry in order to achieve end-to-end integrity (Zhou, 2024). As we trust in an irreversible digest and cryptographic signature, we remove the possibility of unaware policy replacement or reversal, making it harder to comply strictly with the bar of safety required in healthcare regulations.

2.5 CLI-First Tooling

A solution should be practical to fit the workflow of cloud operators and DevOps engineers in order to be workable. Command-line interfaces (CLIs) continue to be a predominant form of interaction in this field because of their scriptability and efficiency. Voronkov et al. (2019) showed that system administrators tend to use CLI tools when they need to perform more complex tasks (e.g. firewall and infrastructure management), as they feel that this approach gives them more control and allows them to write commands into scripts. Likewise, a survey of cloud developers conducted by Coleman et al. (2022) has found that in numerous operational tasks, CLIs are more preferred due to the provision of consistency and speed, particularly in the process of handling a variety of resources or automation via shell scripts. This means that a cross-cloud policy management solution must provide a powerful CLI to make engineers who reside in the terminal adopt it.

Customisability and automation is another benefit of CLI-centric design. Schroder and Cito (2022) indicated that CLI commands are often customized and written by developers to fit the workflow to their requirements and this is friendly to DevOps pipelines. To input a CLI tool, it can be readily triggered in CI/CD pipelines or scheduled work (such as the regular release and validation of policy bundles as a deployment operation). This automation is necessary to achieve ongoing compliance - policies updates and their delivery can be incorporated with existing build/release processes. Additionally, the results of CLI can be in machine-readable form (e.g. JSON outputs or particular exit codes), so other systems can use the output of CLI. This helps with auditing: an example would be the verify command of the tool producing a comprehensive verification report that is stored as evidence in an audit log, which would be more difficult to reliably record with a point-and-click interface.

Accessibility and usability of CLIs also should be taken into consideration. Although powerful, CLI tools should be made in a simple way to prevent user error. Sampath et al. (2021) explain about the accessibility of CLI, and that documentation and reasonable defaults are the key to allowing set command-line tools to be used by a broad professional audience (including individuals that might use assistive technologies). Simplifying the commands (possibly one unified tool to create, sign, and install policies) and giving feedback in plain words (e.g. an explicit success message that says Signature valid, bundle safe to use) will make the operators feel more confident and will decrease the training load, in our case. Being able to provide evidence at the command line: each stage (build, sign, verify) should generate some output, which can be logged, is a conceptual match to both operator preferences and audit requirements.

Lastly, the use of Minimum workable Product (MVP) to develop this tooling is justified by the recent literature. According to De Alwis and Sedera (2022) and Lortie et al. (2024), when developing an MVP in software development, core features providing the main value must be concerned with, and the refinement can be made iterative after user feedback is received. The tool suggested in this study is CLI-first and works across vendors, and it is new as well, so adhering to the principles of MVP application would imply a basic but dependable implementation of the

necessary pipeline (reproducible build, push to registries, signature attach/verify, and OPA bundle run). Unnecessary frills (like a GUI or wide policy editing functions) are not in scope at first - this is to prove the point and get actual usage information on the ground. The project provides a functional solution within a short timeframe to start with a very simple CLI containing the open-source components required (OPA, ORAS, Cosign, and others), and then the project can be extended as needed. The advantage of this approach is that it aligns with the culture of agile and DevOps that is common in most cloud interactivity, where small tools that are effective at a single task can be combined and iterated quickly (Raghavan et al., 2020).

2.6 Summary of Gaps

Overall, existing literature and the modern best-practices provide bits of the puzzle: multi-cloud policy enforcement has been researched (demonstrating the viability of OPA), OCI registry ecosystem now supports portable policy bundles and supply chain security tooling can sign and verify tamper-evidence. These aspects have not yet been combined into a single, end-to-end solution to deliver healthcare-level cross-cloud policies. No documented case studies or reference architectures exist that illustrate how to package, sign, distribute and auto-verify policy-as-code across AWS, Azure and GCP in a holistic way. This study fills that gap by contributing to the state of the art - that is, turning into practice the idea that policies represent immutable, signed objects, just like application code. The proposed tool, a CLI-based tool and reference pipeline will bring the concepts in the literature (digest-based integrity, Sigstore signing, OPA policy engine) to bear in a unified model. This way, it will offer a roadmap to delivering tamper-evident, auditable policy enforcement across clouds to meet the technical and compliance needs previously identified in the literature.

3 Methodology

This dissertation follows an artifact-focused, build-and-evaluate methodology. First define compliance-driven requirements for policy distribution in multi-cloud healthcare settings (integrity, origin record, auditability, portability, operator ease of use), then engineer a minimal yet complete reference pipeline that treats policy bundles as primary OCI artefacts and binds them to immutable digests and signatures. The study proceeds in two paired environments to control for portability and operator realism:

- Windows workstation where the CLI-first Python tool runs OPA/ORAS/Cosign end-to-end.
- Cloud-hosted Linux shell (Azure Cloud Shell) where the identical steps run with zero local state and temporary tooling.

Small, synthetic Rego policies are used as inputs to avoid handling real PHI. All bundles are built in a repeatable way to ensure that a byte-for-byte identical policy directory always yields the same tar digest. The independent variable is the delivery substrate (local filesystem vs OCI registries on GHCR/ACR), while the controlled process is the six-step pipeline (build → sign → push → discover → verify → run). Dependent measures include:

- tamper-evidence (does verification fail on any bit-flip or tag rollback),
- auditability (are machine-readable facts digest, signature identity/material, timestamps, and subject/reference links available at each hop), and
- portability (can the same signed bundle and its referrers be pushed, discovered, and verified across registries without bespoke packaging).

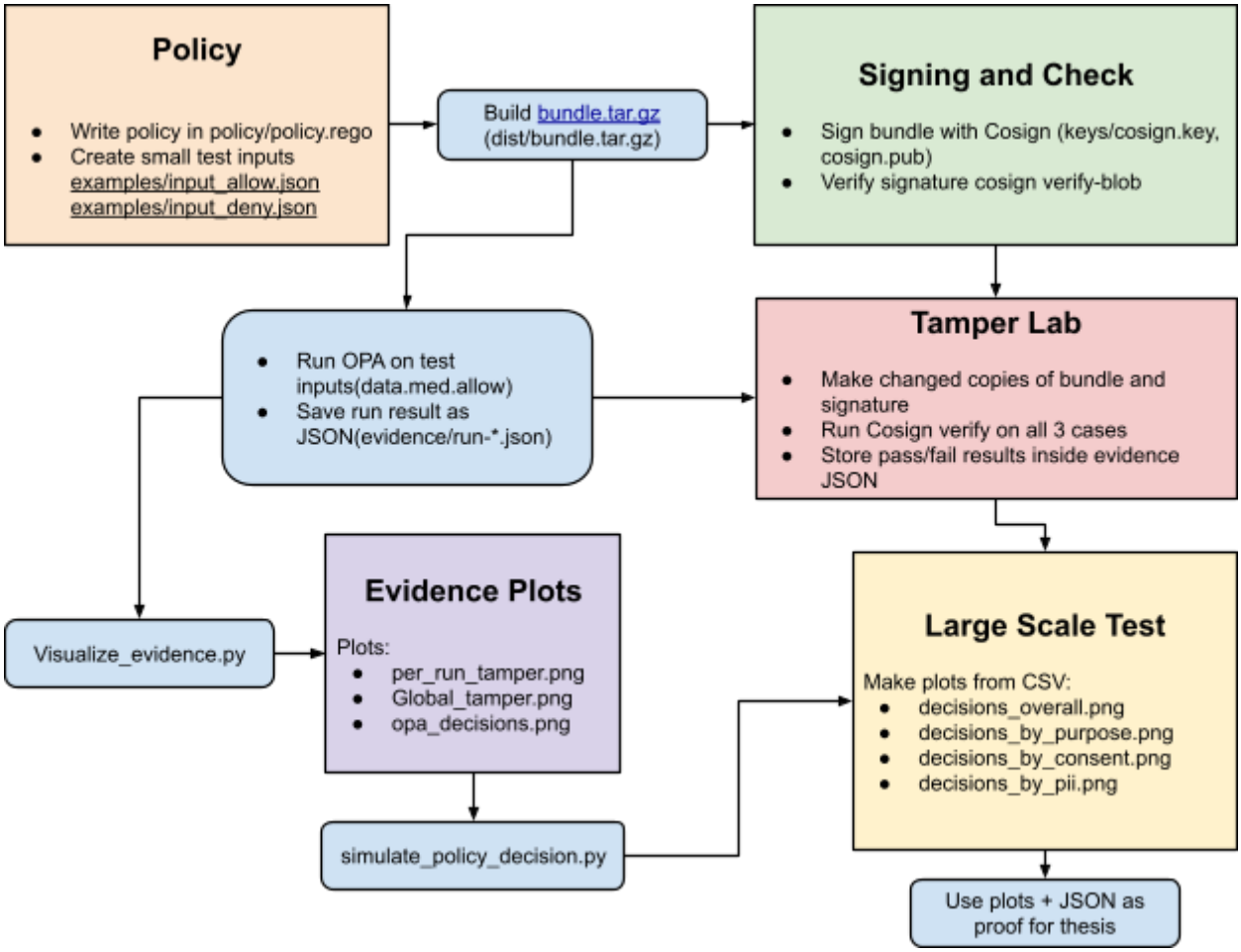

Figure 1: The architecture diagram for the MedPolicy CLI and OCI

Each step is logged at each stage to produce JSON or stable text outputs captured to a run log, and we pin every consumer to the manifest digest (@sha256:…) to remove uncertainty from mutable tags. Threats to validity are addressed by fixing tool versions inside the session, probing Cosign capability (v2/v3) at runtime to avoid flag skew, and re-running the pipeline after deliberate changes: tag reassignment, digest corruption, and signature removal. Success criteria are defined in advance:

- Verification must succeed only for the exact subject digest and fail for any modified payload or mismatched key.
- Referrers discovery must return signatures bound to the same digest regardless of registry.

- OPA must only evaluate bundles that have passed signature verification.
- All evidence required for an audit trail must be exportable as machine-readable artefacts.

By constraining scope to distribution and verification (not authoring), and by using CLI-first interactions with scriptable outputs, the methodology emphasizes repeatability and objective, automatically verifiable results aligned with healthcare audit workflows.

Requirement (healthcare)	Operational risk	Measurable outcome in study	Mechanism tested
Integrity & tamper-evidence	Silent policy substitution/rollback	Verify fails on bit-flip or tag mismatch over time	Digest pinning + Cosign verify on manifest
origin record & end-to-end tracking	Unclear origin, no chain of custody	Signer identity & time recoverable	Signature bundle/referrer discovery
Auditability	Manual screenshots/logs	JSON evidence artefacts	CLI JSON output + digest/subject linkage
Portability	Registry/provider lock-in	Same artefact works on GHCR/ACR	OCI v1.1 media types + ORAS referrers
Operator usability	mismatch over time from complex tooling	Single command surface & exit codes	Python CLI wrapper + POSIX/Windows parity

4 Design Specs

The system is a narrow, pipeline-shaped toolchain that treats “policy as artefact” under OCI semantics and attaches verifiable metadata through signature referrers. The primary components are:

- Python CLI controller that normalizes the environment, wraps subprocess calls, and emits machine-readable results.
- OPA to compile a directory into a bundle (or accept an existing bundle) and to evaluate decisions.
- ORAS to push/pull non-image artefacts with correct media types and to query referrers.
- Cosign to produce and verify signatures over the registry manifest digest (the object consumers actually pull).

The architecture assumes a minimal trust base: clients trust their own key material (file-backed keys for MVP; KMS or keyless OIDC are easy to swap in later), registries store subjects and their referrers, and verifiers enforce digest-tied trust before running any policy.

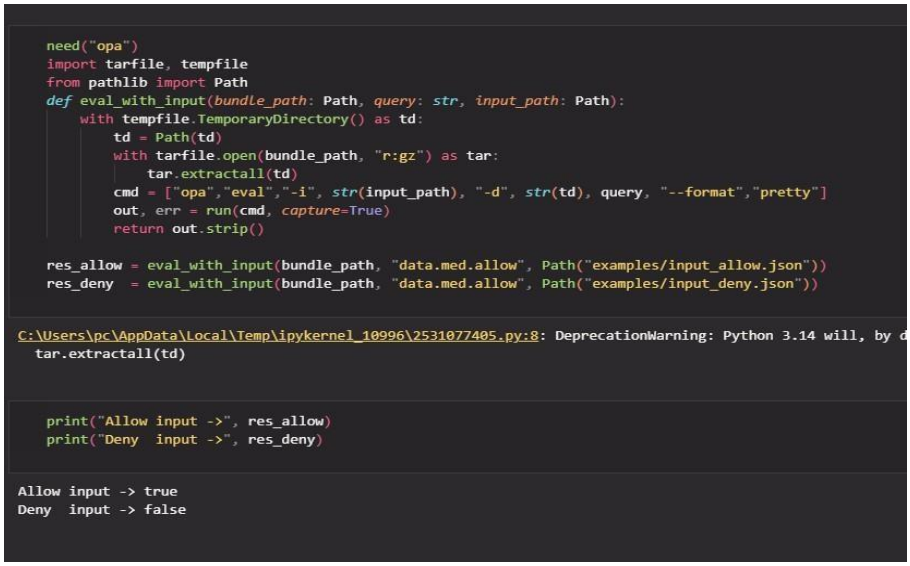

Figure 2: Local Cloud Shell OPA, Cosign, and ORAS installed with versions verified.

The build stage produces a deterministic tar.gz by zeroing ownership, modes and sorting entries; the push stage publishes a config blob plus the bundle layer using OCI-standard media types so that any OCI-compliant registry can store the object. The discover stage queries referrers by subject digest (not by tags), returning signatures and related evidence. The verify stage checks Cosign signatures either via the v2 signature file or the v3 "bundle" format depending on capability. The run stage loads the verified bundle into OPA and evaluates data queries with inputs supplied by tests or calling systems. Errors are surfaced with non-zero exit codes and captured stderr, and the controller prefers safe to rerun operations (re-running the same digest is a no-op).

The user experience is intentionally CLI-centric: single-purpose commands, quiet success, explicit failure, and a --json/machine-readable mode for CI and audit import. Security design decisions include digest pinning everywhere, verification gates prior to execution ("fail closed"), and key safe key handling (private key never committed. Public key distributed to verifiers). Cloud-specific logic is minimized to authentication: GHCR uses PAT, ACR uses az acr login --expose-token for bearer credentials; the artifact shape and verification logic remain provider-neutral.

Component	Interface	Input	Output	Notes
Python controller	CLI / subprocess	Paths, keys, registry refs	JSON/lines + exit codes	Detects Cosign v2/v3; safe to rerun
OPA	opa build, opa eval	Rego dir, input JSON	bundle.tar.gz, decisions	Deterministic bundle; eval gates
ORAS	oras push/pull/disc over	Files, media types	Digest, referrers graph	OCI v1.1 referrers; digest-first

Cosign	sign, verify	Manifest digest	Signature, verify status	File-key MVP; KMS/keyless later
Registry	OCI Distribution	Subject digest, referrers	Immutable content address	GHCR/ACR/ECR/GAR parity by design

5 Implementation

The prototype is installed in two paired tracks to show that it is portable and operator friendly. On Windows, a Jupyter/.ipynb notebook can be used locally to run the whole flow without terminal access. The notebook preconfigures PATH to C:\Tools, makes sure that the files cosign.exe and opa.exe can be found and COSIGN_PASSWORD is exported to prevent an interactive response. The keys/cosign.key and keys/cosign.pub are generated by a key-management cell when missing as a result of invoking cosign generate-key-pair (managing legacy builds with no output-key/ no output-pub by creating in CWD and relocating files under./keys/). To be compatible between Cosign releases, the code uses cosign sign-blob --help and selects the v3 bundle flow or the v2 --output-signature flow.

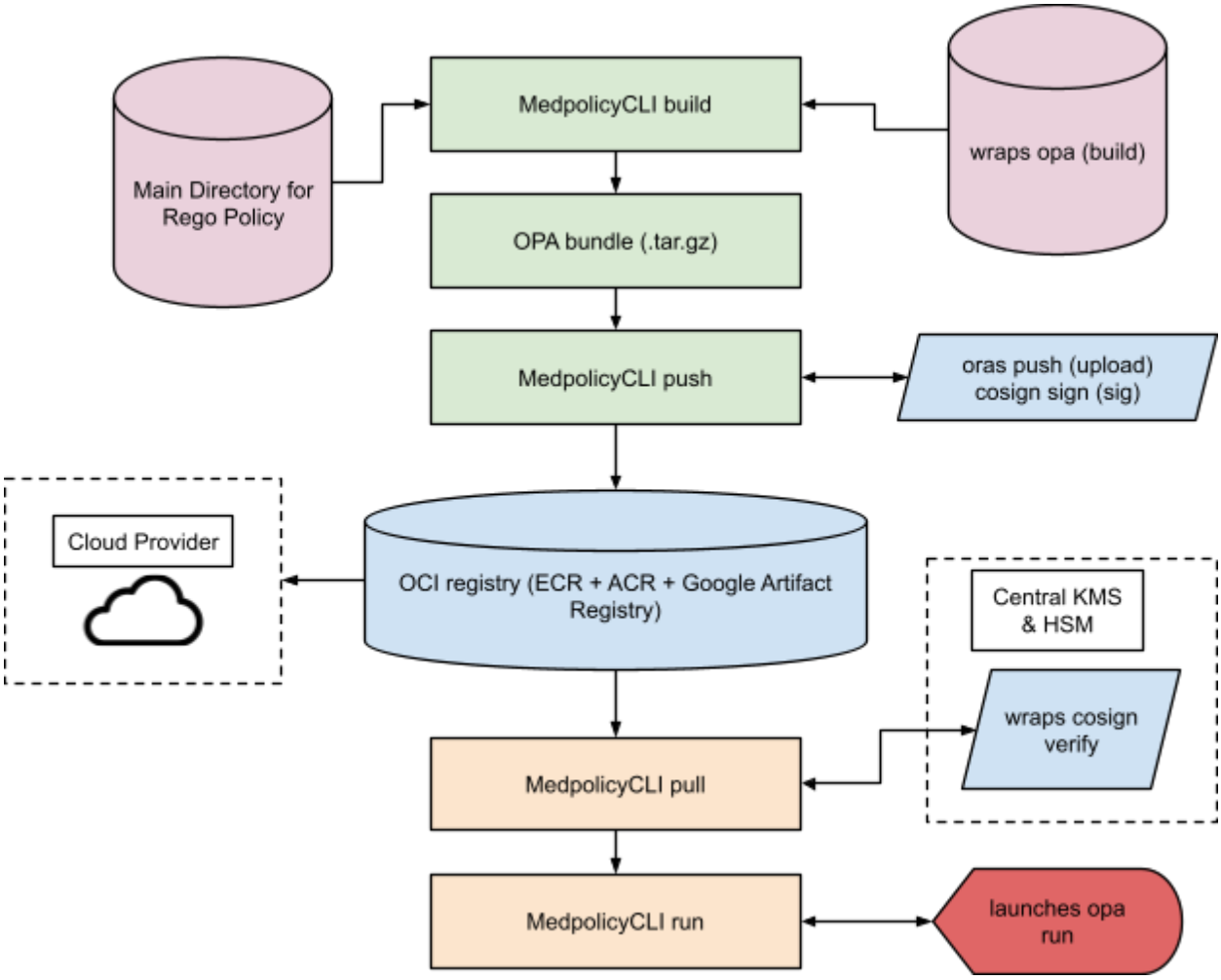

Figure 3: Detailed working structure for the bundle signing and evaluation process by Cosign & OPA.

The deterministic packer transfers bundle.tar.gz in the policy/ directory with a zeroed uid/gid/mtime and sorted contents; and its digest is printed and stored as the unit of trust in the future. Assessment is done by untarring a temp directory and running opa eval on data.med.allow with two small inputs which account for the positive and negative decision paths. Each of the subprocesses is enclosed by a helper run() which captures output/stderr, exits structured errors and writes machine readable artefacts (e.g., derived public key, signature file or bundle). It is local and can test the pipeline in a comfortable, scriptable environment, as well as give a quick inner loop on development.

As part of the cloud, there is a temporary but repeatable runner known as Azure Cloud Shell (Bash). One set up script renders the session standalone: it adds the directory of binaries to PATH, installs static binaries of ORAS, Cosign, and OPA into the directory of binaries, and then runs the same six stages on a registry. To have a zero-cost path, the script is aimed at GitHub Container Registry (GHCR) with a PAT to be able to prove the entire flow without creating Azure resources. To create or reuse an Azure Container Registry (ACR) a helper script with Azure CLI is used to create or reuse an Azure Container Registry, acquires a short-lived bearer token with az acr login with exposetoken, and authenticates oras without Docker.

In both examples, the script creates the policy/ directory to bundle.tar.gz (so the native bundle semantics of OPA can be used), pushes it with oras push using OCI-standard media types (config + tar layer), calculates the immutable manifest digest of the tag (to demonstrate tag mismatch over time is eliminated), signs the digest with cosign sign (file-key MVP with a passphrase; keyless/KMS are easy to swap in later), verifies with cosign verify, discovers the referrers with oras discover to demonstrate that the signature attached to the subject has been found Since Cloud Shell is a transient application, persistence is obtained cheaply by storing the scripts in either a public or a private GitHub repository (or Gist) and executing them with a one-line bash. In case the repository is private, a GitHub API raw fetch is performed in authenticated mode such that no cloning is necessary, the credentials are requested by the script, which purges them after access. The scripts are independent (re-execution does not corrupt state), and produce logs that can be accepted by an audit.

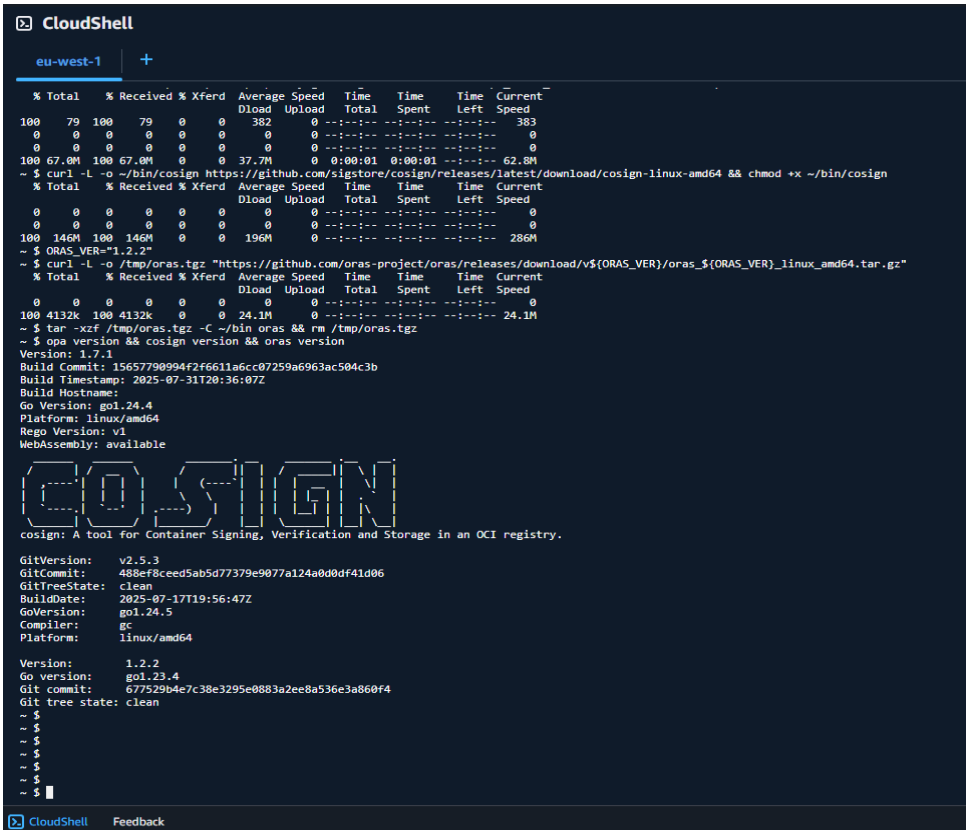

Figure 4: AWS CloudShell OPA, Cosign, and ORAS installed with versions verified.

In order to make such implementation concrete and auditable, key evidence is registered at the end of each run by the pipeline. An example of a simple JSON record may contain the source directory hash (pre-build), the bundle digest (post-build) of the result, the reference to the registry to be pushed and the resolved @sha256 (post-push), the Cosign verification results and the signer identity and time (post-verify), the discovered referrers (post- discover) and the OPA decision results, along with input digests (post-run). This trail shows who signed what, when, and what was carried out precisely, and this is the key to healthcare-quality assurance.

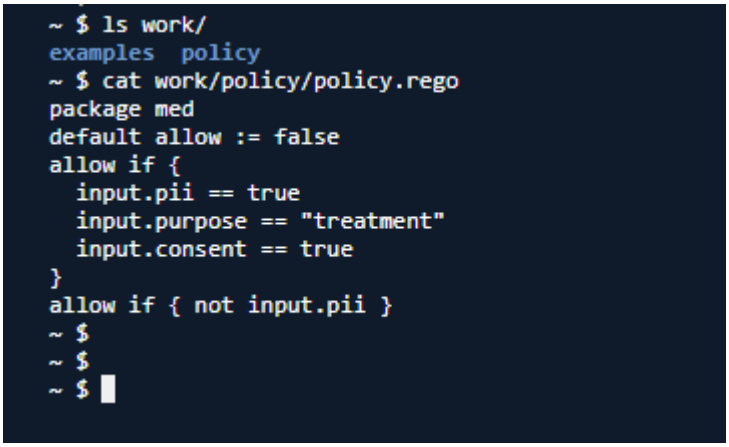

Figure 5: Canonical policy (package med) used for experiments (default-deny).

An adoption of a few of the defensive behaviors learned in the course of development is also codified: rejection of evaluation in the event which came under verification. The refusal to execute against a tag is the no prompts surface Cosign prompts, via environment variables. Explicit, human-readable causes of failure (e.g. unknown flag, auto-handled by invoking a compatible Cosign).

Step	Local (Windows)	Cloud (GHCR/ACR)	Evidence captured
Build	Python tar packer or opa build -b	opa build -b	bundle.tar.gz digest
Push	(optional) ORAS via WSL or skip	oras push (OCI media types)	Repo + manifest digest
Sign	cosign sign-blob (v2/v3)	cosign sign by digest	Signature object/referrer
Discover	n/a or oras discover	oras discover -o json	Signature → subject link
Verify	cosign verify-blob	cosign verify	Pass/fail + signer material
Run	opa eval gated by verify	opa eval gated by verify	Decision outputs

6 Evaluation

6.1 Evaluation goals and success criteria

The evaluation tests whether the implemented pipeline provides:

- integrity/tamper-evidence.
- auditability via machine-readable evidence.
- safe enforcement gating (OPA runs only after verification).

These map directly to the thesis requirements: digest-tied trust, signature validation prior to runtime, and evidence suitable for audit. Success criteria were defined in advance:

- Verification succeeds only for the exact, originally signed artifact (or subject) and fails for any modified payload or mismatched signature.
- OPA evaluation is gated behind verification (“fail closed”).
- Evidence artifacts record at minimum: bundle digest, signature/verification outcome, evaluation query + inputs + decision outputs.

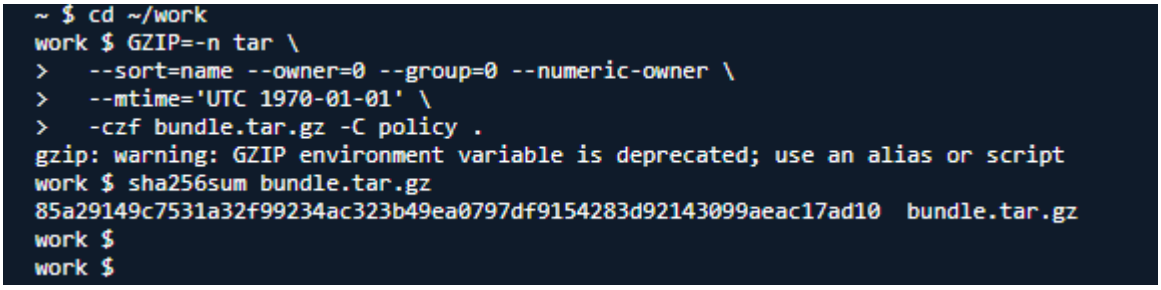

Figure 6: Reproducible bundle build (tar flags) and SHA-256 digest recorded.

The evaluation uses a minimal healthcare-themed policy and synthetic inputs (no PHI). A deterministic bundle is built as dist/bundle.tar.gz, then signed and verified with Cosign, and finally evaluated with OPA against allow/deny inputs. The run record includes timestamp, bundle path, SHA256, size, signing material pointers, verification status, and OPA outcomes.

6.2 Experiment 1: End-to-end pipeline correctness

Procedure:

- Build bundle.tar.gz from policy/ (deterministic tarball).
- Sign the bundle using Cosign.
- Verify signature using Cosign.
- Evaluate data.med.allow with two test inputs (one expected allow, one expected deny).
- Persistent JSON.

The result is that this experiment is a complete run, producing a stable bundle digest and recorded verification status "ok" followed by OPA evaluation results allow=true and deny=false for the two test inputs.

The interpretation for the results is that this experiment shows the MVP pipeline works end-to-end, and that evidence can be emitted as a durable, machine-readable artifact suitable for later audit review (timestamp + digest + verify status + decision outputs).

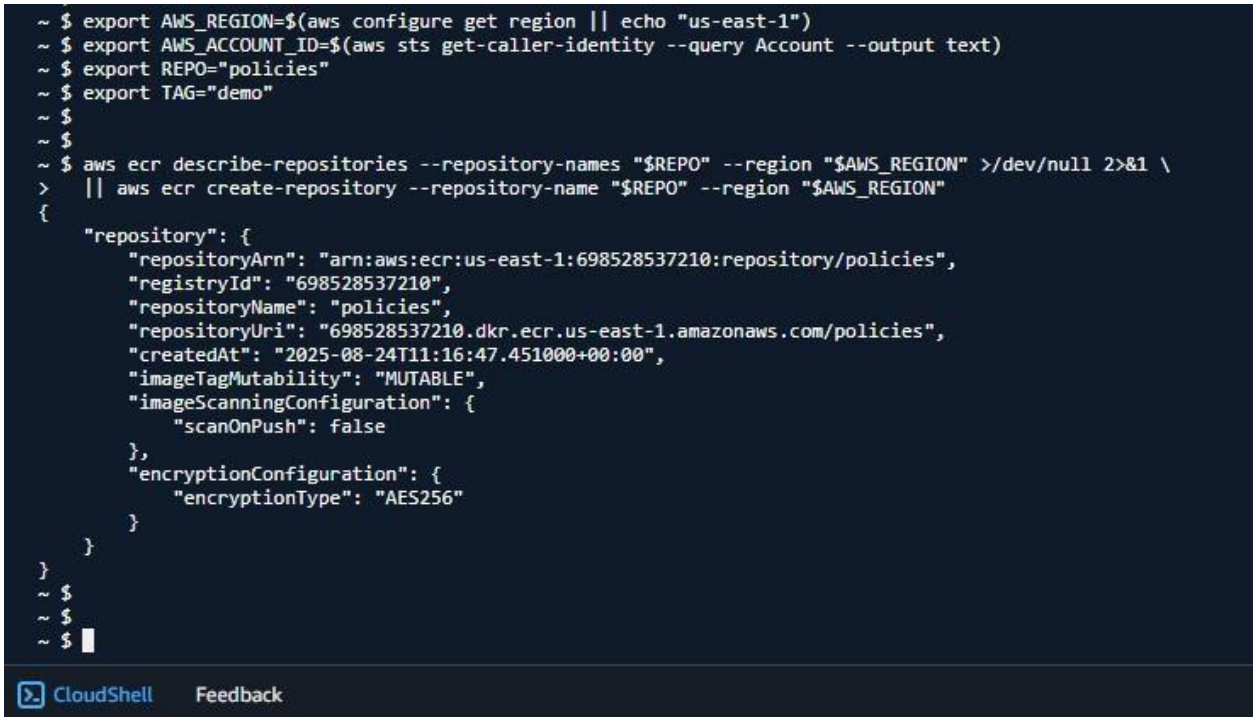

Figure 7: ECR repository created for policy artifacts with ARN and URL returned.

6.3 Experiment 2: Tamper-evidence

Procedure:

- Two attacker-style small changes were introduced:
- Payload tamper: flip a byte in the bundle tarball.
- Signature tamper: flip a byte in the signature file.
- Then run verification over the three cases:
    - original bundle + original signature
    - tampered bundle + original signature
    - original bundle + tampered signature

Verification succeeded only for the original/original pair and failed for both tampered scenarios:

- original bundle + original signature → success
- tampered bundle + original signature → failure
- original bundle + tampered signature → failure

This provides direct evidence of tamper-evidence: any single-bit modification breaks verification, preventing silent substitution. It supports the thesis claim that cryptographic signatures can detect unauthorized policy modification prior to use.

6.4 Experiment 3: Policy decision simulation

To complement the single allow/deny inputs, the simulator generated a larger synthetic dataset and produced plots summarizing decision distributions by purpose, PII flag, and consent flag. This experiment is not meant to validate healthcare policy correctness, but to show the "run" stage behaves consistently and can generate operator-friendly outputs for evaluation/benchmarking.

Across one 500-sample run of synthetic inputs, the decisions showed a stable allow/deny split and clear separation by input attributes (especially by the PII flag, reflecting the policy rule structure). These outputs provide a template for future benchmarking (latency, throughput, mismatch over time detection under policy changes) and usability testing.

6.5 Auditability assessment

The evidence JSON produced by the pipeline contains:

- Timestamp (UTC) and bundle digest (unit of trust).
- Verification outcome recorded as a status field.
- OPA query and decision results tied to named inputs.
- Tamper-lab case outcomes recorded per run (success/failure), suitable for audit evidence of control how well it works.

This meets the MVP auditability goal: evidence can be archived, re-read, and summarized automatically without screenshots or manual transcription.

6.6 Findings and Discussion

The results of the evaluation confirm the key argument of this thesis: policy-as-code is a digest-pinned signed OCI artifact that can deliver more integrity guarantees and a more audit-friendly operation than an ad-hoc file distribution or mutable tags. In the pipeline runs, the unit of trust is the fixed subject digest and the verification is done against that digest prior to the execution of any policy. This directly attacks the time-of-check/time-of-use risk mentioned in the motivation: the consumer is no longer relying on a moving reference, but a content-addressed object with a cryptographic assertion of integrity. The most transparent integrity finding is the tamper-lab experiments. Cosign verification failed, and the original signed artifact passed when either the bundle payload was changed by a single-byte change, or the signature material was changed. When aggregated with all the runs, the only case that was successful was the original bundle + original signature (3 successes), and all the tampered cases were unsuccessful (6 failures). This is very solid empirical evidence of tamper-evidence: any unknown alteration can be noticed at verification time and may be exploited to impose a fail-closed posture (i.e. prevent policy analysis and as a consequence prevent enforcement in case of untrusted policy content).

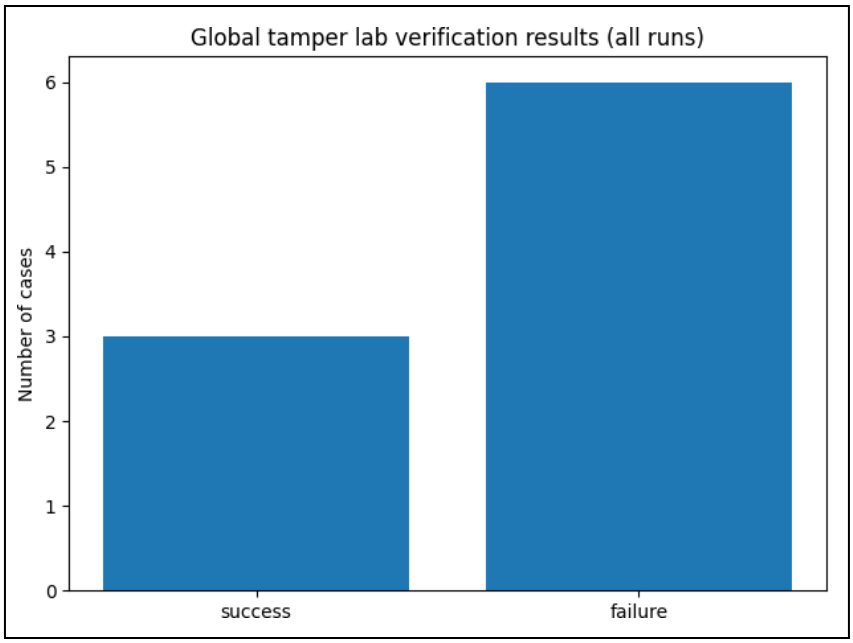

Figure 8: Tamper lab results i.e. verification succeeds only for the original signed bundle and fails for all tampered cases.

Experiment 1 also shows that this verify-before-use control is operationally compatible with the execution of OPA. There are consistent results in the repeated-run OPA result chart, the two canon test inputs being the expected path, allow, repeatedly generated allow=true, and the expected path, deny, repeatedly generated allow=false. This, combined with the verification gate, allows a significant operational property of regulated environments: the system is not only deterministic under the same input but also safe by default, since only policy decisions are generated when the signed artifact has been verified.

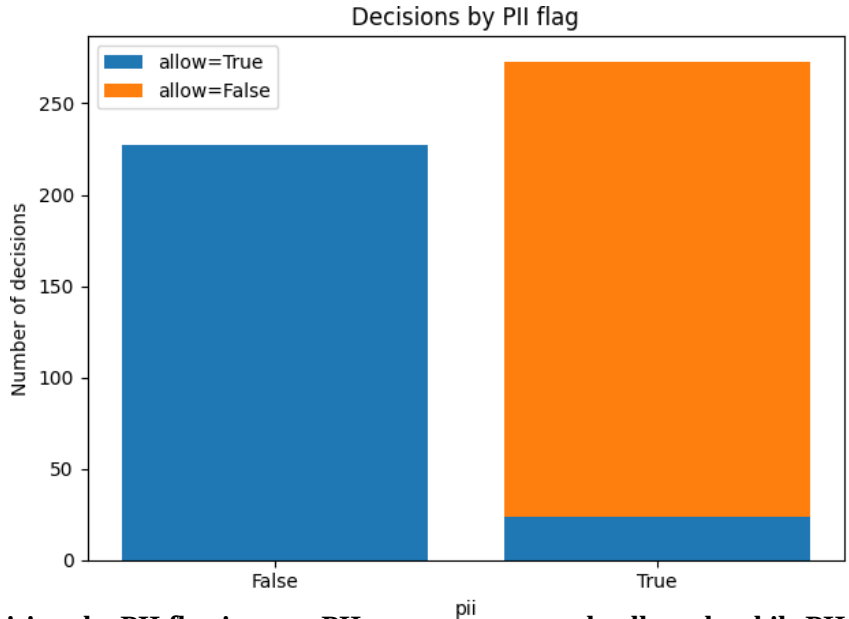

Figure 9: Decisions by PII flag i.e. non-PII requests are mostly allowed, while PII requests are mostly denied.

Lastly, the simulation experiment demonstrates that, with synthetic data, the operator- and audit-friendly summaries can be generated at scale (with a run stage), without altering the underlying security model. The stratified decision plots are very sensitive to the important attributes: The decisions made are heavily biased towards allow when PII is false and towards deny when PII is true, demonstrating that the policy logic is being implemented in a predictable and explainable manner. There are also changes in outcomes with consent and purpose-based grouping indicates visible differences (e.g., it would seem that treatment is more permissive than other purposes in this dataset). Although this does not prove the policy policy right, it shows that proper policy implementation can result in verifiable evidence (distributions and breakdowns) which may be affixed to a particular signed digest, making audit readiness more than simple verification passed.

7 Conclusion & Future Work

This thesis designed and confirmed a practical, CLI-first method for healthcare-grade policy delivery by treating OPA bundles as primary, immutable artifacts and enforcing verify-before-use as a hard gate. The implemented pipeline in a repeatable way packages policy-as-code into a bundle, derives a stable content digest, produces a cryptographic signature, verifies that signature prior to evaluation, and then executes OPA queries only when verification succeeds. The evaluation demonstrates the core safety guarantee required in regulated environments: any unauthorized change to the policy payload (or its signature material) is detectable and blocks execution, while successful runs produce structured evidence that links enforcement outcomes back to a specific policy artifact.

Beyond the prototype mechanics, the main result is a concrete distribution and assurance model that replaces informal file copying and mutable references with digest-tied trust and repeatable release semantics. By making the digest the unit of trust, the approach reduces configuration mismatch over time, eliminates uncertainty from “same name, different content” scenarios, and improves operational consistency in multi-environment deployments. The pipeline also treats auditability as an engineering output rather than a manual last-minute task: each run can emit machine-readable records describing what policy was used (by digest), what verification was performed, and what decision results were produced for given inputs supporting end-to-end tracking, repeatability, and automated evidence collection aligned with healthcare-grade expectations.

Future work should expand this thesis into a deployable cross-cloud policy supply chain and a reusable reference implementation for organizations. The next step is to run the identical artifact workflow against OCI registries across AWS, Azure, and GCP, demonstrating that one policy bundle can be published once, pinned and retrieved by digest everywhere, and verified consistently prior to enforcement using registry-centric discovery of associated security metadata where available. Building on that, the CLI and evidence format can be productized into a lightweight “policy release” toolchain integrated into real delivery points (CI/CD pipelines, cluster admission checks, API gateways, or sidecar-based enforcement), including policy promotion stages (dev → test→ prod) driven by approved digests. Finally, the thesis can be used as a platform for broader adoption and impact: standardizing an evidence schema for audit import, adding richer origin

record signals (identity-based signing, attestations, build context), and conducting benchmarking and usability studies to quantify extra cost, operator error rates, and the oversight benefits of shifting from mutable policy distribution to verifiable, artifact-based policy management.

8 References

Agarwal, M. et al. (2020) Project CLAI: Instrumenting the Command Line as a New Environment for AI Agents. arXiv:2002.00762.

Berry, Q. (2024) Announcing the Open Container Initiative Referrers API on Quay.io: A step towards enhanced security and compliance. Red Hat Blog, 2 December 2024.

Coleman, C., Griswold, W.G. and Mitchell, N. (2022) Do Cloud Developers Prefer CLIs or Web Consoles? CLIs Mostly, Though It Varies by Task. arXiv:2209.07365.

Cosign repo signing docs (cosign_sign.md). Low-level flag behavior for methods section. https://github.com/sigstore/cosign/blob/main/doc/cosign_sign.md

Deshpande, S. (2020) OCI Artifact Support in Amazon ECR. AWS Containers Blog.

De Alwis, I. and Sedera, D. (2022) Minimum workable Product in Information Systems Development Context: Systematic Mapping Study. Proceedings of the Australasian Conference on Information Systems (ACIS 2022), Paper 83.

Distribution Spec (v1.x). Canonical API for pushing/pulling and referrers support. https://github.com/opencontainers/distribution-spec

Gajdos, M. (2022) Announcing Docker Hub OCI Artifacts Support. Docker Blog. Gazitt, O.

(2022) Securing the Software Supply Chain for Policy-as-Code. Aserto Blog. IBM (2020)

Digests versus image tags. IBM Filenet Documentation.

Lortie, J., Chao, R., Tracy, J. and Fietkiewicz, K.J. (2024) Unpacking the Minimum workable Product (MVP): A Framework for Use, Goals and Essential Elements. Journal of Small Business & Enterprise Development (early access).

Newman, Z., Meyers, J.S. and Torres-Arias, S. (2022) Sigstore: Software Signing for Everybody. Proceedings of the ACM Conference on Computer and Communications Security (CCS 2022).

Open Policy Agent Documentation (2023) Bundles and Signature Verification. OpenPolicyAgent.org.

Raghavan, D. et al. (2020) POSH: A Data-Aware Shell. Proceedings of the USENIX Annual Technical Conference (ATC), pp. 617–631.

Sampath, H., Merrick, M.A. and Macvean, A. (2021) Accessibility of Command Line Interfaces. Proceedings of the CHI Conference on Human Factors in Computing Systems, Article 489.

Schröder, M. and Cito, J. (2022) An Empirical Investigation of Command-Line Customization. Empirical Software Engineering, 27, 30.

Voronkov, A., Martucci, L.A. and Lindskog, S. (2019) System Administrators Prefer Command Line Interfaces, Don't They? An Exploratory Study of Firewall Interfaces. Proceedings of the 15th Symposium on Usable Privacy and Security (SOUPS 2019).

Webster, N., Burton, A., Hawkins, E. and Pum, M. (2023) Automated Compliance Checks with Open Policy Agent (OPA) in Multi-Cloud. Research article.

Zhou, F. (2024) Announcing support of OCI v1.1 specification in Azure Container Registry. Microsoft Tech Community Blog, 28 June 2024.