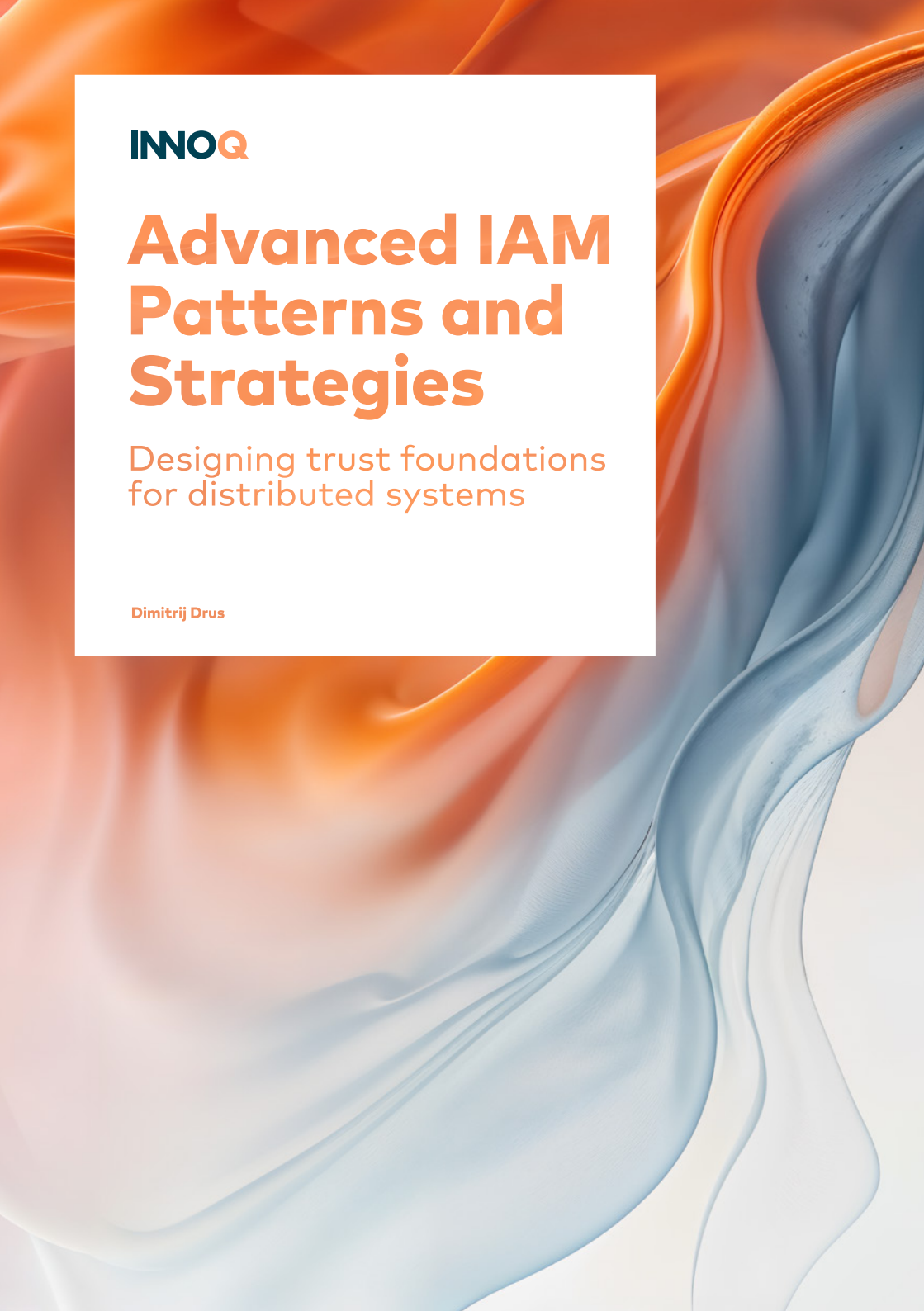
INNOQ
Advanced IAM Patterns and Strategies
Designing trust foundations for distributed systems
Dimitrij Drus

Advanced IAM Patterns & Strategies

Designing Trust Foundations for Distributed Systems

Dimitrij Drus

innoQ Deutschland GmbH
Krischerstraße 100 · 40789 Monheim am Rhein · Germany
Phone +49 2173 33660 · www.INNOQ.com

Layout: Tammo van Lessen with XeLaTeX
Design: Murat Akgöz
Typesetting: André Deuerling

Advanced IAM Patterns & Strategies – Designing Trust Foundations for Distributed Systems
Published by innoQ Deutschland GmbH
1st Edition · January 2026

Voices From The Field

When I speak to teams about security, identity, and how to secure microservices in the real world, the question is rarely about what tool, and more about how do you approach solutions strategically. Resources tend to provide high level advice, but rarely go into how to actually arm professionals with details and considerations to make the right choice for their own business. This guide has been tremendously helpful in filling in that gap and is a go-to resource of mine when speaking with security and development teams.

Karl Lewis, Founder, Cloud Native CISO

This critique of the OWASP Microservice Security Cheat Sheet highlights where the guide falls short and sparks a call for deeper, more modern guidance so teams don't end up defaulting to weak or risky authorization choices. It's only getting harder with the introduction of AI and MCP, so this guide is timely in calling for us to evaluate the security of critical infrastructure.

Mike Schwartz, Founder & CEO, Gluu

Contents

Preface

Distributed systems have changed significantly over the past decade. Microservices, APIs, and increasingly interconnected components have become the norm, and identity information now moves through these systems in ways that earlier security guidance rarely anticipated. Yet in many organizations, decisions are still based on documents that reflect assumptions which do not always align with today's architectural realities.

A good example is the OWASP Microservices Security Cheat Sheet[1]. It reflects ideas from several standards in a developer-friendly way, contains valuable guidance, and has shaped the thinking of many teams. But it also leaves important questions unanswered. Some of its patterns are described too briefly to offer real guidance, a few contain contradictions, and some rely on examples that fit only specific contexts[2]. In practice, these gaps often lead to fragile or "accept-by-default" designs and implementations.

And that brings us to a broader, more subtle issue I encounter regularly in my work. It is not just about the cheat sheet itself, but also about how such documents are used. As part of my job, I often try to help teams at different organizations make better decisions based on real-world risk and context. But the moment I question a practice that is claimed to be based on OWASP or similar guidance, I usually get immediate pushback:

Developers say: "Our security team requires this".

Security teams say: "We follow OWASP. It is the gold standard".

Because of this, practices that may be risky or outdated can become immune to scrutiny simply because they are rooted in a respected source. This dynamic makes it difficult to challenge entrenched guidance, even when there are good

[1] https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Microservices_Security_Cheat_Sheet.md

[2] A detailed discussion of these points, including specific examples from the cheat sheet, can be found in the first post of the blog series on which this book is based.

reasons to do so. As noted earlier, this phenomenon isn't unique to OWASP. Similar issues have emerged around other well-known guidance, such as the widely adopted but later reconsidered NIST password policies originally shaped by Bill Burr, whose own reflections[3] highlight how even well-intentioned recommendations can unintentionally lead to widespread problems when the context changes.

These observations eventually led to a conference talk[4] I gave at OWASP AppSec EU 2025[5]. Although the abstract only hinted at it, the talk examined several gaps in the Microservices Security Cheat Sheet and outlined possible ways to rethink some of its underlying patterns. Shortly before the talk, Jim Manico - one of the chairs of the OWASP Cheat Sheets Project - suggested that it would be great if I opened a pull request with an improved version of the cheat sheet during the session. I hadn't planned for that and certainly hadn't prepared a rewrite. But his remark stayed with me. Combined with colleagues repeatedly encouraging me to write down the explanations I often shared on projects, it felt like the right moment to put these ideas into a more structured form.

When I returned home, I began drafting a revised version of the cheat sheet. Very quickly it became obvious that the material had grown far beyond what could reasonably fit into such a format. The concepts, patterns, and decision points needed more space - and in some cases, the patterns themselves had no established terminology at all. That raised a different question: how do I collect feedback, validate the descriptions, and refine the terminology before contributing anything back?

The answer turned out to be a blog series[6]. Several colleagues who reviewed the early drafts independently suggested the same approach. That is how the series came into being, and how the content that eventually grew into this book first took shape - with the aim of providing a clearer foundation for authentication, identity

[3] https://www.cbsnews.com/video/man-who-suggested-random-ever-changing-passwords-regrets-that-guidance/

[4] https://owasp2025globalappseceu.sched.com/event/1wh8k/the-edge-strikes-back-challenging-owasps-take-on-edge-level-authorization

[5] https://owasp.glueup.com/event/owasp-global-appsec-eu-2025-123983/

[6] https://www.innoq.com/en/blog/2025/07/whats-wrong-with-the-current-owasp-microservice-security-cheat-sheet/

propagation, and authorization in distributed systems, where conceptual misunderstandings most often translate into real-world vulnerabilities. The patterns described in the following chapters are intended to help you evaluate trade-offs, understand where certain approaches work well, and see where they fall short.

Relation to the OWASP Cheat Sheets Project

You may now be wondering what happened to the intention to update the OWASP Microservices Security Cheat Sheet, or - if you have looked at it recently - what exactly I was referring to in the previous chapter. Depending on when you're reading this primer, one or the other situation may well apply.

In fact, as I write this primer, roughly half of the content from the blog series has already been transformed into new OWASP Cheat Sheets, but they have not yet been officially published - yes, several cheat sheets have been created or are currently in progress. This is simply because these documents build on one another, and the new Microservices Security Cheat Sheet, at least as currently planned, is meant to serve as an overarching document for the others. Once all of them are ready, they will be published together, creating an entirely new category of cheat sheets-architectural security-something the project hasn't had before.

As this primer predates the completion of that work, it can be seen as the second iteration of the knowledge consolidation that began with the blog series. Some parts have indeed been updated and expanded. Accordingly, the new cheat sheets may be understood as a third iteration and will likely contain additional refinements - or may already do so, depending on when you read this book.

Navigating This Primer

This primer is structured around six numbered chapters, each building on the concepts introduced before it. The actual content begins with **Core Concepts**, which provides the foundation for all that follows. To help you orient yourself, I'll outline what each chapter covers and how the pieces fit together.

- As already mentioned, **Chapter 1 - Core Concepts** establishes the terminology and mental models used throughout the primer: subjects and identities, first- and third-party contexts, policy lifecycles, trust boundaries, and the architectural reference model that underpins the later patterns. If you are unfamiliar with these ideas, this is the chapter to read carefully. The rest of the book assumes you have these concepts in mind.

- **Chapter 2 - Authentication Patterns** examines different architectural approaches to authentication. It focuses on trade-offs and constraints rather than prescribing a single "correct" pattern. Frankly, this also applies to every other chapter in this primer.

- Modern distributed systems rely on identity flowing between components. **Chapter 3 - Identity Propagation Patterns** explores how this is done in practice, what can go wrong, and what different identity propagation strategies imply. It provides the conceptual bridge between authentication and authorization.

- **Chapter 4 - Authorization Patterns** describes where and how authorization decisions can be made: embedded enforcement, externalized PDPs, and further variants. It shows how architectural choices influence security properties, operational flexibility, and failure modes.

- Authorization patterns are rarely "good" or "bad" in isolation. **Chapter 5 - Decision Dimensions for Authorization Patterns** introduces a structured way to analyze them using multiple dimensions - such as policy complexity, performance, or availability requirements - so you can evaluate which patterns fit your system's needs.

- The final **Chapter 6 - Practical Considerations & Recommendations** pulls the threads together and looks at real-world implications: operational pitfalls, deployment scenarios, governance and lifecycle challenges, and how to evolve trust architectures over time. It is the most applied chapter and best read after the preceding ones.

How to Read This Primer

If you are new to the topic, reading the chapters in order will give you a coherent, cumulative understanding of authentication, identity propagation, and authorization in distributed systems. If you already have experience with some of these areas, you may choose to focus on specific chapters - for example, if you are primarily interested in access control, you may start with **Authorization Patterns**, visiting **Core Concepts** for orientation as needed.

Wherever you begin, the intention of this primer is to offer a foundation of concepts and patterns that you can adapt to your own context. There is no silver bullet, and every system has its own requirements, constraints, and trade-offs. Use the material here for reasoning, experimentation, and refinement.

Acknowledgments

Neither the blog series this primer builds on, nor the primer itself, nor even my involvement in the OWASP Cheat Sheets Project would likely exist without an early exchange I had with Jim Manico. His encouragement - and the well-timed "push" that came with it - convinced me to turn an initial critique into something more substantial. Without that nudge, I might never have started writing any of this.

I also want to thank my colleagues Martina Meng, Dominik Guhr, Timo Loist, Gil Breth, and Felix Schumacher. Your reviews, questions, and different perspectives helped me sharpen my thinking in ways I could not have done alone. Many ideas in this work exist because you challenged me, and many others became clearer through our discussions. Thank you all for the time, honesty, and patience you invested along the way.

And above all, I want to thank my wife and my daughters. This work came with its share of long evenings spent writing, and more than once I was less present than I should have been. Their patience and understanding made it possible for me to bring this project to an end, and I am deeply grateful for that.

Some Last Words

Naming is hard, and the world of authentication and authorization is no exception. Many terms are overloaded or misleading. Take HTTP's `401 Unauthorized`, for example: despite its name, it represents an authentication error - returned when credentials such as a username or password are missing or invalid. The `403 Forbidden` status is reserved for true authorization failures. Confusing, right?

To avoid this kind of ambiguity and to make discussions clearer, I've tried to establish a consistent and understandable vocabulary throughout my work - a kind of ubiquitous language, if you like - to make the topics easier to follow and reason about.

Please also note that several of the patterns described here do not currently exist in other literature, at least to the best of my knowledge. Therefore, if something doesn't quite make sense or feels off, please bear with me - and feel free to question it. The field is evolving, and so is our shared understanding.

Last, but not least, your feedback, insights, and experiences are always welcome, so we can continue to learn from one another.

1 Core Concepts

This chapter defines the core terminology and architectural concepts used throughout the primer. The patterns in later chapters rely on these foundations, so familiarity with them is essential.

1.1 Authorization Reference Architecture

To lay the foundation for the patterns described in this primer, this chapter introduces the general building blocks of an authorization system, based on NIST SP 800-162[1]. While that standard focuses on Attribute-Based Access Control (ABAC), the conceptual roles it defines are relevant to nearly any access control system.

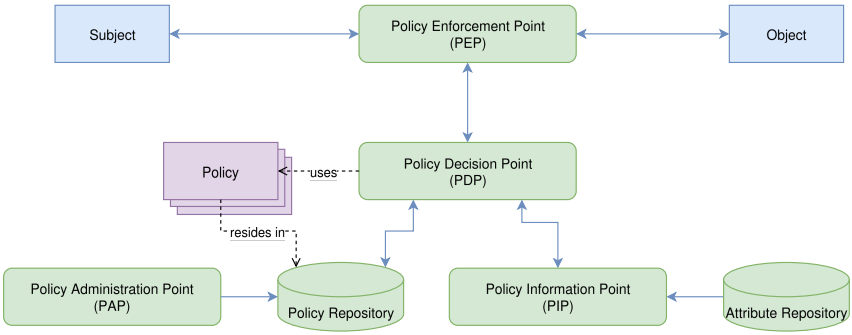

Figure 1.1: Authorization Reference Architecture

These roles are:

- **Subject:** An active entity (e.g., a user, application, or device) that attempts to perform an action on an Object.
- **Object:** A passive entity (e.g., a file, an insurance record, or a blog article), that is the target of an action attempted by the Subject.

[1] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-162.pdf

- **Policy:** A set of rules that define who is allowed to do what under which conditions – for example, "Managers can approve expenses under $1000", or "Users can access documents they own". Policies are evaluated at runtime using attributes of the user, the resource, and the context (such as time, or location).
- **Policy Enforcement Point (PEP):** The component that intercepts a request and enforces the outcome of an authorization decision, allowing or denying the request.
- **Policy Decision Point (PDP):** Evaluates policies and computes authorization decisions based on the incoming request and available data.
- **Policy Information Point (PIP):** Supplies attribute data or contextual information that the PDP requires to evaluate a policy.
- **Policy Administration Point (PAP):** Allows management of access control policies by providing tools for authoring, testing, and maintaining them.

1.2 A Story to Ground the Concepts

Imagine Alice wants to read an article on her favorite blog platform. In this story:

- Alice is the subject.
- The article she wants to read is the object.
- The action ("read") is what she wants to perform.
- Her browser sends a request on her behalf to the platform's backend services.

Let's break down what happens step by step:

Each time Alice interacts with the platform – by clicking a link, submitting a form, or opening a page – a request is made to one or more backend services. These services must decide: *Can Alice do this?* and more subtly: *What exactly is Alice allowed to do in this context?*

That decision process starts with the **Policy Enforcement Point** (**PEP**). Think of the PEP as a gatekeeper – it sees the request and knows it must enforce some kind of access control. But it doesn't contain the logic to decide **what's allowed**. Instead, it delegates that to the **Policy Decision Point** (**PDP**).

The PDP evaluates the request against a set of policies. These policies might include conditions like:

- Alice must be logged in.
- Alice must have an active subscription.
- Alice can only read the full article if her subscription level is “Premium”.

To perform this evaluation, the PDP often needs more information than what’s in Alice’s request. This is where the **Policy Information Point** (**PIP**) comes in. In this case, one PIP is involved: a user management service that provides Alice’s subscription information. This PIP supplies the PDP with the information needed to evaluate the aforesaid policies.

But here’s where a crucial detail often gets missed: the PDP doesn’t always answer a **closed question** like “yes” or “no”. In many cases, the PDP may answer **open questions** with answers like:

- Alice can read the article, but only the excerpt.
- Alice can read the full article if her subscription is “Premium” or if the article is marked as public.
- Alice can read up to three full articles per day on a free plan.
- Alice can read that particular set of articles

In these cases, the PDP returns a decision along with additional access context information that describes *how* access is permitted, and which additional actions to perform. This might include details like which parts of a resource are visible or what usage limits apply. For example, if Alice is on a free plan and has already read three full articles today, the PDP might return a “permit” decision along with structured attributes specifying that only the excerpt of the requested article should be shown.

The PEP then takes that decision and enforces it, meaning the request is either allowed to proceed to the protected resource or is blocked. Enforcement is binary: permit or deny. But if the decision includes additional data – like an instruction to show only the excerpt – it’s up to downstream components to interpret and act on that. In Alice’s case, this means shaping the response to include only the article excerpt.

There is, however, one essential prerequisite: the system must know who the subject is – that is, it must verify the subject's identity, confirming that Alice is indeed Alice. This verification process is the domain of authentication. Without it, the PEP has no basis on which to enforce access decisions. Authentication is therefore foundational, which is why we begin by examining authentication patterns and approaches. But before doing that, there is a need to explore a few essential concepts that provide context for understanding the broader landscape of authentication and authorization.

1.3 Policy Representations and Lifecycle

As described above, authorization policies define who can do what under which conditions. Depending on how they're represented and integrated into a system, their impact on development, operations, and security can vary widely. In practice, policies are implemented in two main ways:

- **Hardcoded policies:** These are embedded directly in application code – for example, conditional checks like `if user.role == 'admin'`. In this model, the PDP is implicit within the application logic, and the PEP might be an interceptor, handler, or a simple conditional branch.
- **Declarative policies:** These are defined outside the application code in structured formats, evaluated by a dedicated PDP. Examples include policies written in Rego[2], Cedar[3], XACML[4], or other authorization languages. This model clearly separates the enforcement logic (PEP) from decision logic (PDP) and externalizes policy definition.

Declarative policies allow the policy lifecycle to be managed independently of the application lifecycle. This separation has several operational and organizational benefits:

[2] https://www.openpolicyagent.org/docs/policy-language
[3] https://www.cedarpolicy.com/en
[4] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

- Access policies reflect business rules, regulatory obligations, or security requirements – and while these are also drivers for application features, their rate of change, ownership, and scope typically differ:
  - Regulatory updates may require immediate changes to access conditions without modifying the underlying feature set.
  - Security incident response might require temporary or permanent changes to access controls outside a normal release cycle.
  - Declarative policies empower non-developers to request or implement access changes (e.g., enabling partner access during a pilot program) without waiting for a full development cycle or redeployment, especially when the changes don't require altering core application logic.
  - Policies may vary in scope or depth, with some governing access across multiple services or applications – for instance, feature-flag-related policies – while others are more narrowly focused. These aspects are depicted by the diagram below.

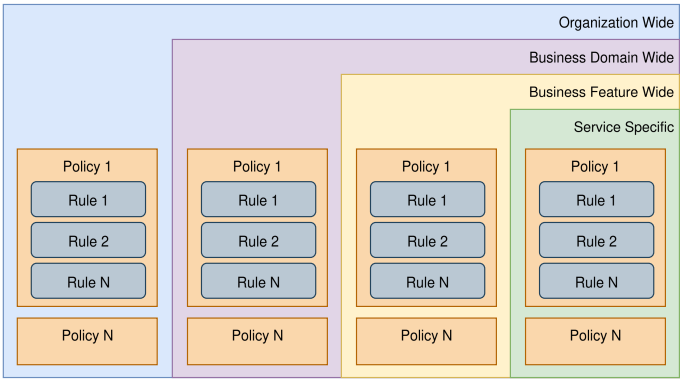

Figure 1.2: Scope & Depth of Policies

When access policies are tightly coupled to code, any change – no matter how urgent or isolated – requires application code change, test cycle, and deployment. Separating policy from application code allows organizations to

react faster and more safely to changes, without compromising the integrity of the software development process.

- Policies are high-stakes – access control bugs are different from feature bugs. They tend to be catastrophic, not just annoying:
  - Granting access when you shouldn't can lead to **data breaches**
  - Revoking access incorrectly can **break business** processes

  Externalized policies are easier to review, audit, and test – just like any other configuration artifact. They can also be subject to staged rollouts and automated validation. Enabling and governing these capabilities is the responsibility of the **Policy Administration Point** (**PAP**), which orchestrates the authoring, validation, and controlled distribution of policies. Policies themselves are also subject to access controls. In that sense, the PAP also incorporates aspects of the PEP and the PDP – but is focused entirely on the policy lifecycle rather than on business-related decisions.

- Declarative policies also enable *before-the-fact audit* – the ability to answer "Who currently has access to this object/resource?" without needing to wait for a request to happen or instrument code. This reverse-query capability is essential for governance, compliance, and risk assessments, and is practically impossible when policies are hardcoded and scattered across multiple applications.

1.4 First-Party vs. Third-Party

A key distinction in access control scenarios lies in **on whose behalf** the subject is acting. This determines whether we're dealing with a **first-party** or a **third-party** context.

In a **first-party** scenario, the subject acts on their own behalf – for instance, when Alice reads or edits articles in her account. In a **third-party** scenario, the subject is acting on behalf of someone else – like when Alice delegates access to an

external service that analyzes her articles, e.g., to check the grammar. The third-party service becomes the subject, while Alice remains the principal authorizing access.

This distinction has important implications for how delegation is modeled, how trust is established, and what guarantees are needed from the involved systems.

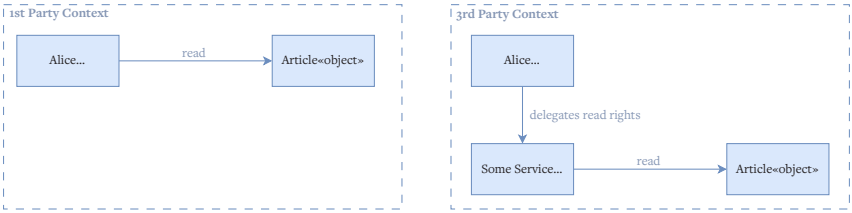

Figure 1.3: First Party vs Third Party

Between these scenarios, there is also a special case within the first-party context, where a user explicitly authorizes a trusted internal service to act on their behalf. In this situation, the user acts both as the subject (requesting the action) and as the PDP by giving explicit consent. For example, in a banking app, the user approves a transaction, while the banking system acts as the PEP. These interactions rely on existing authentication and authorization mechanisms and are enhanced by dedicated protocols to ensure integrity and non-repudiation. Additionally, other PDPs within the system may apply further controls, such as fraud detection, compliance verification, or transaction limits, before final enforcement.

While exploring these contexts, it's important to understand that different protocols address different needs. Some protocols are tailored to the first-party context only, such as Security Assertion Markup Language (SAML)[5] or Central Authentication Service (CAS)[6], which are primarily designed for direct user authentication and carry attributes for authorization purposes within trusted domains. Others, like Open Authorization (OAuth 2.0)[7], focus exclusively on the

[5] https://www.oasis-open.org/standard/saml/
[6] https://apereo.github.io/cas/7.2.x/index.html
[7] https://datatracker.ietf.org/doc/html/rfc6749

third-party context, enabling delegated access to resources on behalf of another party. And then there are protocols like OpenID Connect (OIDC)[8] that support both contexts, combining identity information with delegated access.

What all these protocols have in common is that they define mechanisms to authenticate the involved parties. However, the details of how this authentication is performed - e.g., through passwords, or by making use of other factors - are not covered in this primer. Likewise, the protocols themselves are not the focus here.

1.5 On Subjects, Principals and Identities

Another important topic to understand before we explore authentication and authorization patterns is the distinction between **subjects**, **principals**, and **identities**. These terms appear in various standards – such as NIST SP 800-63[9], NIST SP 800-162[10], and ITU-T X.800[11] – but they are often used inconsistently or even interchangeably. From my experience, this inconsistency tends to cause more confusion than clarity. The definitions used in this primer therefore reflect how these concepts typically manifest in systems we design and operate today.

According to the reference architecture and the story above, a **subject** is an active entity that carries an identity and is the target of authentication.

However, in most real-world systems, authentication is not limited to a single type of active entity. Instead, there are often multiple forms of identity involved, each representing a different kind of actor or context:

- **End-users:** Human users interacting with a system via e.g., a browser, or a mobile app.
- **Devices:** The user's device (e.g., smartphone, laptop, or IoT hardware), which may have its own identity.

[8] https://openid.net/specs/openid-connect-core-1_0.html
[9] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63-3.pdf
[10] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-162.pdf
[11] https://www.itu.int/rec/T-REC-X.800

- **External clients:** Applications or scripts accessing an API on behalf of a user or system.
- **Internal workloads:** Services or components within a distributed system communicating with each other.

All of these are **principals** – identifiable entities that can be authenticated and authorized. A **subject**, in turn, may consist of one or more such principals. For example, a request from a mobile app may involve both the authenticated user and the device they're using. In a service-to-service call, the subject might be the internal service identity, optionally carrying along delegated user context.

Importantly, the definition of a subject is often shaped by the perspective of the PDP that evaluates the request. Each PDP – or even each policy – may view the subject differently, based on what attributes or entities are relevant for its decision-making. One policy may only care about the identity of the user. Another may treat the combination of user and device as the subject. A third may include the client application or network context as additional principals. In this sense, a subject is not a fixed notion, but a context-dependent composition of principals as seen by the evaluating component.

Understanding subjects in this compositional and context-sensitive way is key to interpreting the patterns described in this primer. While many patterns focus on a single principal type (e.g., user or service), they often support **multi-principal subjects** through identity propagation and proper orchestration of authentication mechanisms.

The last remaining concept to cover is **identity**. An identity is a collection of attributes that uniquely identify an entity, similar to a primary key in a database. In some cases, this might be a single attribute such as an ID, while in others it can be a combination of several attributes. Unlike subjects and principals, which refer to active entities or actors, the concept of identity also applies to passive entities – the objects – as well.

2 Authentication Patterns

Effective authorization depends fundamentally on accurately identifying who is making a request. Before a Policy Decision Point (PDP) can calculate a meaningful access control decision, and a Policy Enforcement Point (PEP) can enforce it, the identity – or identities – of the subject associated with that request must first be established. And that's exactly the focus of this chapter: exploring architectural patterns commonly used in distributed systems to establish identity.

These patterns also lay the foundation for a later discussion on identity propagation – a crucial topic, since trustworthy identity propagation is key to maintaining strong trust boundaries across a system.

2.1 Introduction to Authentication Patterns

Authentication can be handled at different layers of a system's architecture. Broadly speaking, there are three main approaches:

- **Service-Level:** responsibility for verifying identity is delegated to each service, or to a proxy tightly coupled to it.
- **Edge-Level:** authentication is centralized in a shared component at the system boundary.
- **Kernel-Level:** authentication is performed in the operating system kernel, using cryptographic identities enforced at the transport layer.

Each approach comes with trade-offs in terms of scalability, consistency, and operational complexity.

Authentication applies to different types of actors. These can be **external** actors, such as end users or client applications outside the system, or **internal** actors, such as services, other workloads, and even the nodes those workloads run on, all operating within the system boundary. The same architectural patterns can often be applied to both kinds of actors, though the technical mechanisms and trust assumptions differ.

Before diving into these patterns, it is also important to clarify what is being verified. Most systems handle authentication in two phases:

- **Primary Authentication:** This is the process of directly verifying credentials tied to authentication factors, such as passwords, biometric inputs, or signed challenges like WebAuthn assertions. For internal actors, this might involve validating machine-issued certificates, SPIFFE IDs, or workload authentication data issued by the platform. This step establishes identity by proving control over a credential and linking it to a known identity, such as a user account or a system identity.
- **Authentication Proof Verification:** After successful primary authentication, the system typically issues an authentication proof – a reusable artifact that confirms the authenticated identity in subsequent interactions. At the application layer, this might take the form of a session cookie, token, or assertion. At lower layers, it can take the form of cryptographic session state, such as a TLS session key, IPsec Security Association, or similar. Verifying the proof ensures that the identity remains trusted without repeating primary authentication.

Where this distinction is not relevant, the term **authentication data** is used to refer collectively to both primary credentials and authentication proofs. The following subchapters use this term when referring to either or both phases.

The patterns described below differ in **what** is verified (credentials in primary authentication vs. authentication proofs), **where** verification happens, and **which** implications this has for system design and trust boundaries.

2.2 Service-Level Embedded Authentication

In this pattern, each service is responsible for handling primary authentication internally. This includes managing identities and credentials, performing credential verification, and implementing authentication workflows. Common credential types used in this setup include username/password, API keys, and similar simple methods. All authentication logic and subject-related data storage are embedded directly within the service, often through custom code or built-in libraries.

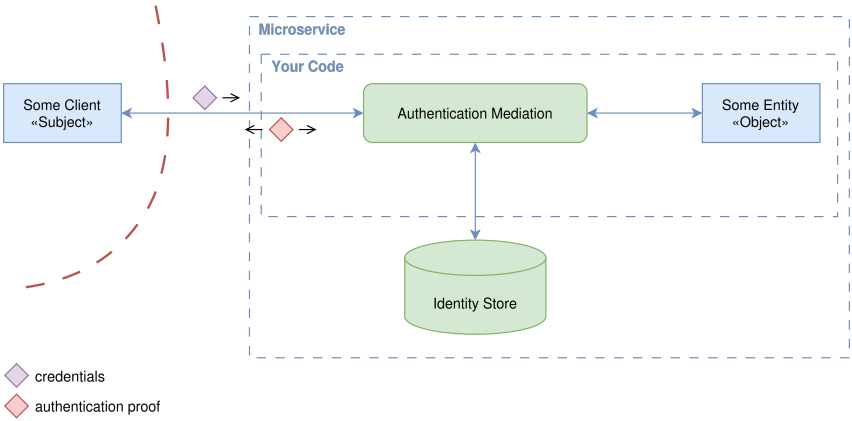

Figure 2.1: Service-Level Embedded Authentication

Pros

- **Simplicity:** Each service is fully self-contained and does not rely on external systems or additional infrastructure for authentication.
- **Customization freedom:** Authentication behavior can be adapted to service-specific requirements without external constraints.
- **Support for external and internal actors:** Since the implementation of a service can fully control all authentication-related functionality, orchestration of different authentication contexts – like authentication of internal services and external users - is possible, but comes with significant complexity (see also the authentication orchestration con below).

Cons

- **Inconsistency:** Authentication behavior, credential storage, and authentication flows differ across services, leading to fragmentation and a poor user experience, incl. not being able to support SSO.
- **Security risk:** Authentication code is duplicated across services, increasing the risk of vulnerabilities and complicating audits.
- **Maintenance burden:** Changing authentication methods (e.g., introducing MFA) requires updates across all affected services.

- **Limited scalability:** Each service is responsible for identity management, complicating secure identity management across a large system. This makes the pattern unsuitable for scalable service-to-service authentication.
- **Limited observability and governance:** Suspicious activity often goes undetected without centralized monitoring. Credential reuse, account compromise, or brute-force attacks on one service remain invisible to others, hindering coordinated detection and response.
- **Authentication orchestration:** Handling of multi-principal subjects – that is supporting multiple authentication configurations, including protocol chaining and subject-specific variations, required to support different contexts, like first- and third-party, or external client and service-to-service authentication – adds significant complexity.
- **Coupling of external authentication data with internal trust assumptions:** Using the same authentication data for both external clients and internal services increases the risk of leakage and unauthorized access. If an internal service is inadvertently exposed because of a misconfiguration or an attacker gaining internal access, the leaked authentication data may enable unauthorized access to sensitive resources.

2.3 Service-Level Code-Mediated Authentication

This pattern addresses key limitations of the **Service-Level Embedded Authentication**, such as fragmented identity management, duplicated credential stores, and lack of support for SSO. In this pattern, the service no longer verifies credentials directly. Instead, an external Identity Provider (IdP) authenticates the subject and issues authentication proofs. The service verifies these internally and extracts identity attributes for request processing.

Pros

- **SSO support:** Identity and credential lifecycle is consolidated in the IdP, enabling Single Sign-On and reducing duplication.
- **Lower security risks:** Centralized authentication reduces the attack surface related to credential handling.

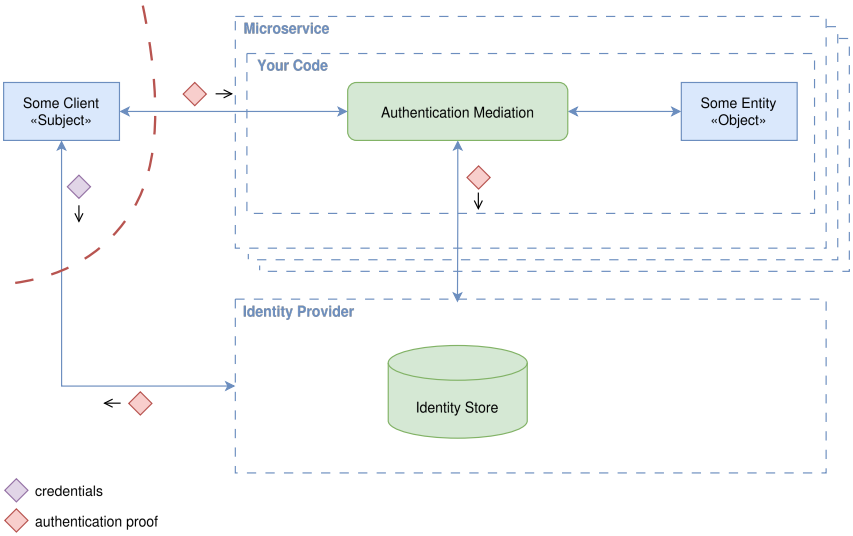

Figure 2.2: Service-Level Code-Mediated Authentication

- **Improved user experience:** Consistent authentication flows and session handling across services.
- **Interoperability:** Widely adopted protocols like OIDC[1] and SAML[2] provide flexibility and broad integration possibilities with various IdPs.
- **Customization freedom:** Services can still tailor authentication behavior to specific needs, for example, in environments where standards like OIDC are not applicable.
- **Support for external and internal actors:** Since the implementation of a service can fully control all authentication-related functionality, orchestration of different authentication contexts – like authentication of internal services and external users - is possible, but comes with significant complexity (see also the authentication orchestration con below).

[1] https://openid.net/specs/openid-connect-core-1_0.html
[2] https://www.oasis-open.org/standard/saml/

Cons

- **Protocol handling overhead:** Each service must implement and maintain logic for authentication proof verification and protocol-specific behavior.
- **Misconfiguration risks:** Incorrect verification logic, such as missing expiration checks or improper cryptography use, can introduce severe security vulnerabilities.
- **Authentication orchestration:** Handling of multi-principal subjects – that is supporting multiple authentication configurations, including protocol chaining and subject-specific variations, required to support different contexts, like first- and third-party, or external client and service-to-service authentication, adds significant complexity.
- **Coupling of external authentication data with internal trust assumptions:** Using the same authentication data for both external clients and internal services increases the risk of leakage and unauthorized access. If an internal service is inadvertently exposed because of a misconfiguration or an attacker gaining internal access, the leaked authentication data may enable unauthorized access to sensitive resources.

2.4 Service-Level Proxy-Mediated Authentication

This pattern builds on the previous pattern but further reduces complexity within services by offloading authentication-related logic to a dedicated proxy deployed as a sidecar alongside the service. The proxy operates in front of the application, forwards requests locally to it, performs verification of authentication proofs with the Identity Provider (IdP), and injects identity context, typically via headers, into requests before forwarding them to the service.

Pros

- **SSO support:** Identity and credential lifecycle is consolidated in the IdP, enabling Single Sign-On and reducing duplication.
- **Lower security risks:** Centralized authentication reduces the attack surface related to credential handling.

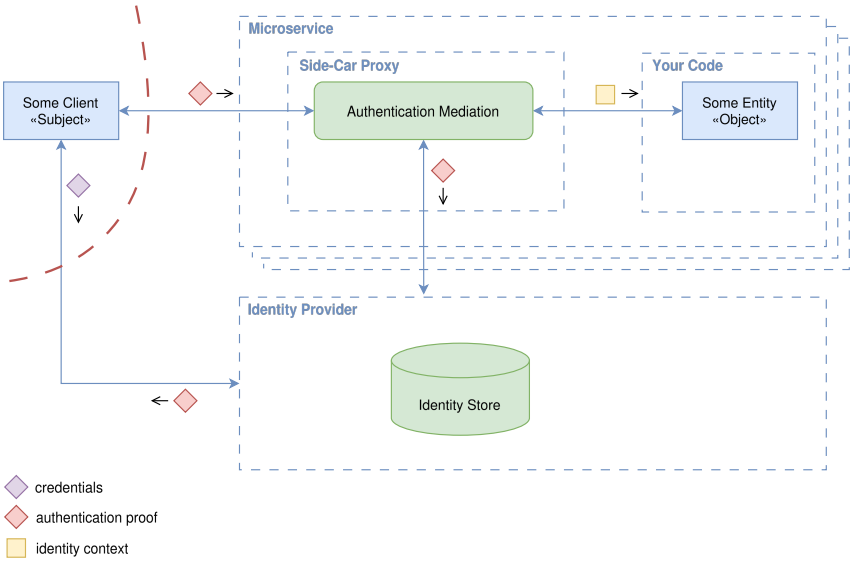

Figure 2.3: Service-Level Proxy-Mediated Authentication

- **Improved user experience:** Consistent authentication flows and session handling across services.
- **Interoperability:** Widely adopted protocols like OIDC[3] and SAML[4] provide flexibility and broad integration possibilities with various IdPs.
- **Separation of concerns:** Removes authentication-related logic from application code by offloading it to the proxy, simplifying service development and reducing maintenance effort.
- **Consistent behavior:** Identity verification and protocol handling in the proxy ensure uniform behavior across services.
- **Improved security posture:** Reduces the risk of implementation flaws by consolidating authentication-related logic into a dedicated, hardened component.
- **Authentication orchestration:** Some proxies support multiple authentication configurations, including protocol chaining and subject-specific variations.

[3] https://openid.net/specs/openid-connect-core-1_0.html
[4] https://www.oasis-open.org/standard/saml/

This enables support for different contexts, such as first- and third-party access, or a mix of external clients and internal services.

- **Strong foundation for service-to-service trust:** Enables Zero Trust[5] networking with workload identity, typically realized via systems like SPIFFE/SPIRE[6], which define workload identities embedded in X.509 certificates[7] used for mTLS[8] authentication between services.

Cons

- **Operational complexity:** Requires deployment and maintenance of additional components per microservice, leading to higher resource usage and costs.
- **Header spoofing risk:** Misconfiguration or insufficient validation in the proxy can allow malicious clients or internal actors to spoof or manipulate identity headers. Ensuring correct proxy setup and strict header validation is essential to maintain the integrity of identity information.
- **Configuration consistency:** All proxies across the service landscape must be configured uniformly to ensure consistent authentication behavior and user experience. Inconsistencies in configuration can lead to confusing user flows or even security vulnerabilities.
- **Coupling of external authentication data with internal trust assumptions:** Using the same authentication data for both external clients and internal services increases the risk of leakage and unauthorized access. If an internal service is inadvertently exposed because of a misconfiguration or an attacker gaining internal access, the leaked authentication data may enable unauthorized access to sensitive resources.

[5] https://csrc.nist.gov/pubs/sp/800/207/final
[6] https://spiffe.io/
[7] https://www.rfc-editor.org/rfc/rfc5280
[8] https://www.rfc-editor.org/rfc/rfc8446

2.5 Edge-Level Authentication

In this pattern, authentication is handled at the system boundary by a shared component such as an API gateway or ingress proxy. This component authenticates incoming requests from external clients before they reach internal services. It integrates with one or multiple Identity Providers (IdPs) using protocols such as OIDC[9], OAuth 2.0[10], SAML[11], mTLS[12], or other mechanisms, and propagates verified identity information, typically via headers, to downstream services for further processing.

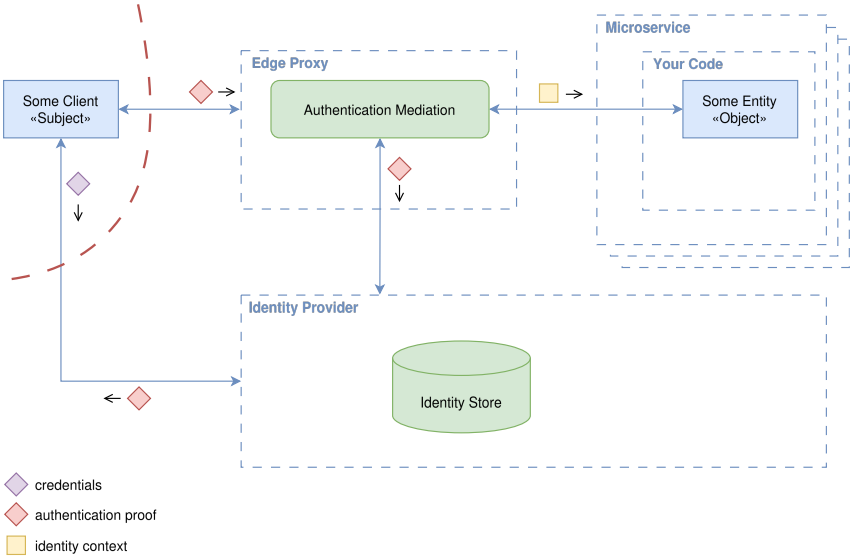

Figure 2.4: Edge-Level Authentication

[9] https://openid.net/specs/openid-connect-core-1_0.html
[10] https://www.rfc-editor.org/rfc/rfc6749
[11] https://www.oasis-open.org/standard/saml/
[12] https://www.rfc-editor.org/rfc/rfc8446

This approach consolidates authentication logic into a single enforcement point, simplifies service implementation by removing per-service authentication handling, and is particularly common in Zero Trust[13] architectures.

Pros

- **Improved consistency:** Authentication is performed uniformly and consistently across services at a single entry point, reducing fragmentation, configuration drift, and improving auditability.
- **Simplified service logic:** Internal services are relieved from implementing authentication logic, focusing only on authorization and business functionality.
- **Faster service onboarding:** New services can rely on existing infrastructure for authentication, requiring minimal additional setup.
- **Protocol-agnostic identity propagation:** Verified identity information can be propagated to internal services using trusted, implementation-independent formats (e.g., via a newly issued JWT[14], injected headers carrying identity information in protected form using standards like HTTP Message Signatures[15]), or signed proprietary structures. This avoids passing raw external authentication data, as is necessary with all previous patterns.

Cons

- **Limited granularity:** Fine-grained or per-endpoint authentication policies (e.g., step-up authentication) are generally harder to implement and may require additional coordination with downstream services. This depends heavily on the capabilities of the edge proxy
- **Identity propagation challenges:** Ensuring secure and reliable propagation of identity context (e.g., via headers) requires strict validation and trust models between the edge and internal services. Proper governance can help overcome this limitation.
- **Single point of failure:** While the ingress proxy or gateway is already a central component in most architectures, performing authentication at the edge

[13] https://csrc.nist.gov/pubs/sp/800/207/final
[14] https://www.rfc-editor.org/rfc/rfc7519
[15] https://www.rfc-editor.org/rfc/rfc9421.html

makes it a critical part of the security infrastructure. Misconfiguration or compromise can impact not just access, but the integrity of authentication decisions system-wide.

- **Not suitable for service-to-service authentication:** Edge-level authentication only applies to incoming external requests. Internal service-to-service calls require additional authentication mechanisms. Although technically possible, routing internal communication through the edge may introduce severe performance bottlenecks.

2.6 Kernel-Level Authentication

This pattern involves performing authentication at the operating system kernel level using cryptographic identities attached to either a service or the machine/node it runs on. The actual implementation is based on protocols, such as IPSec[16], or WireGuard[17]. The identity of a peer is cryptographically verified on each exchanged packet and is limited to layer 3[18]. This form of enforcement is transparent to applications, making it a strong foundation for secure communication between workloads.

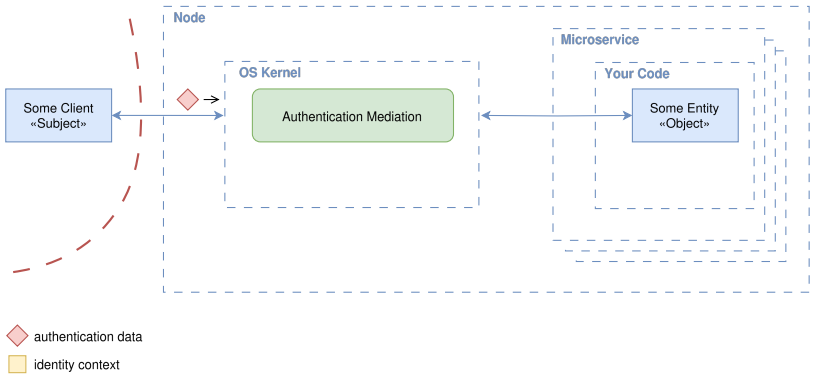

Figure 2.5: Kernel-Level Authentication

[16] https://www.rfc-editor.org/rfc/rfc6071
[17] https://www.wireguard.com/
[18] https://en.wikipedia.org/wiki/Network_layer

Pros

- **Transparent to applications:** Services do not need to implement authentication logic; identity is enforced by the OS Kernel.
- **Protocol-agnostic:** Applies to all traffic types, not just HTTP.
- **Low latency:** Enables fast connection setup with strong isolation guarantees.
- **Provides strong workload identity:** Provides identity verification tied directly to the transport channel, reducing risk of spoofing or replay, which makes it a strong foundation for service-to-service trust and enables Zero Trust[19] networking models.

Cons

- **Not suitable for layer 7 – application-level – authentication:** Identities are tied to workloads or nodes only and not to individual users or external clients. Because of this, this pattern cannot convey user-specific identity attributes.
- **Limited observability:** Monitoring is confined to connection-level data (e.g., source/target workloads), lacking insight into user-driven actions within the application.
- **Infrastructure complexity:** Requires robust automation for identity management, and OS- or kernel-level authentication policy enforcement mechanisms (e.g., via eBPF[20]).

2.7 Operational and Security Considerations

While the above authentication patterns differ primarily in terms of *where* and *how* authentication is performed, they also have significant implications for operations and authorization. Choosing the right pattern often comes down to balancing development flexibility, operational effort, and risk tolerance.

[19] https://csrc.nist.gov/pubs/sp/800/207/final
[20] https://ebpf.io/

2.7.1 Operational Considerations

Pattern	Configuration & Implementation Burden	Operational Overhead	Observability Scope
Service-Level Embedded	High	High	Application-specific
Service-Level Code-Mediated	Medium	Medium	IdP + Application-specific
Service-Level Proxy-Mediated	Medium	High (infra cost)	Proxy + Application
Edge-Level	Low	Low	Centralized (Proxy)
Kernel-Level	Low-Medium	High (infra complexity)	Network-level only

Patterns with decentralized authentication (like **Service-Level Embedded Authentication**) typically incur more operational overhead due to inconsistencies, duplicated configuration, and monitoring complexity. Centralized patterns reduce duplication but introduce infrastructure dependencies and require resilient design.

2.7.2 Security Considerations

Security risks increase significantly when authentication logic and credentials are handled directly within application code. Centralized enforcement approaches – whether at the IdP, edge, or within the OS kernel – help limit exposure, enforce stronger boundaries, and reduce the risk of misconfiguration (especially at the edge). However, care must be taken to prevent trust leakage, which directly impacts the ability to enforce the principle of least privilege. Achieving this depends not only on where authentication occurs, but also on how identity information

is propagated and verified downstream. Without trustworthy, tamper-resistant propagation, even strong initial authentication can be undermined – weakening trust boundaries and ultimately impairing the system's ability to make reliable authorization decisions. To address this, the next chapter examines common identity propagation strategies and their impact on system security, observability, and trust enforcement.

Additional important aspects

Operational and security concerns such as token theft, replay protection, session lifecycle, and reauthentication are critical when implementing authentication mechanisms. These topics are extensively covered in e.g., OWASP Authentication Cheat Sheet[a], and Session Management Cheat Sheet[b].

[a] https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

[b] https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

3 Identity Propagation Patterns

As mentioned in the previous chapter, trustworthy identity propagation is essential for maintaining strong trust boundaries across a system.

Architectures following Zero Trust[1] principles exemplify this need, as they emphasize strict access control and continuous verification. This chapter introduces commonly used identity propagation patterns – that is, the ways in which identity context flows between services. These patterns influence where and how access control decisions are made, the reliability and trustworthiness of those decisions, and ultimately how effectively least privilege can be enforced. They also differ in how tightly internal services are coupled to the external authentication mechanisms and identity representations used at the boundary.

Some identity propagation patterns aim to decouple internal service logic from specific external authentication protocols and data formats. This approach, often called *protocol-agnostic* or *token-agnostic* identity propagation, means internal services consume a normalized, unified identity representation that abstracts away the details of the original authentication protocol and authentication data (including both primary credentials and authentication proofs). This abstraction enables internal services to remain stable, simplified, and focused on authorization logic, even as external authentication methods evolve or change.

At one end of the spectrum, some patterns directly forward externally issued authentication data (such as OAuth 2.0 tokens, session cookies, or certificates) downstream, requiring internal services to understand and process the original authentication protocols. This approach can increase complexity and trust assumptions within internal services. At the other end, a trusted system component at the edge transforms incoming authentication data into cryptographically signed, normalized identity structures. These structures abstract away the original protocol and data format, allowing internal services to remain agnostic to how authentication was performed. By providing tamper-resistant, verifiable representations of identity, they establish strong trust boundaries across service interactions and enable auditable access decisions, making them especially effective for enforcing least privilege in distributed environments.

[1] https://csrc.nist.gov/pubs/sp/800/207/final

Between these extremes exist intermediate patterns where internal services rely on simplified identity representations issued or transformed by upstream services but without cryptographic protections, requiring implicit trust between services.

Each pattern involves trade-offs between implementation complexity, security, trust, privacy, and operational overhead. Choosing the appropriate identity propagation approach depends on the system's security posture, scalability requirements, and the desired level of trust between internal components.

Understanding these trade-offs in concrete terms requires examining how identity propagation is commonly implemented in practice. The following chapters describe representative patterns along this spectrum, highlighting their characteristics, benefits, and limitations.

3.1 External Identity Propagation

In this pattern, the edge component forwards the externally received authentication data (e.g., an access token, ID token, session cookie, or certificate) directly to internal services without transformation. The internal services are responsible for the verification of the received authentication data, for extracting the identity context (such as user ID, or other attributes), and making access control decisions based on it. When an internal service needs to communicate with another service, it just forwards the authentication data further downstream. The aforesaid verification may require contacting a Verifier, which depending on the authentication protocol and data used, could be an authorization server that issued the token or, for example, an OCSP responder to check the revocation status of a certificate.

As already said above, the actual verification of the authentication data, represented by the dotted lines in steps 3 and 5 of the diagram above, depends on the type of authentication data used. For example, in the case of an opaque token, each service must call the appropriate identity provider, respectively, authorization server endpoint to retrieve the associated data. If the token is self-descriptive,

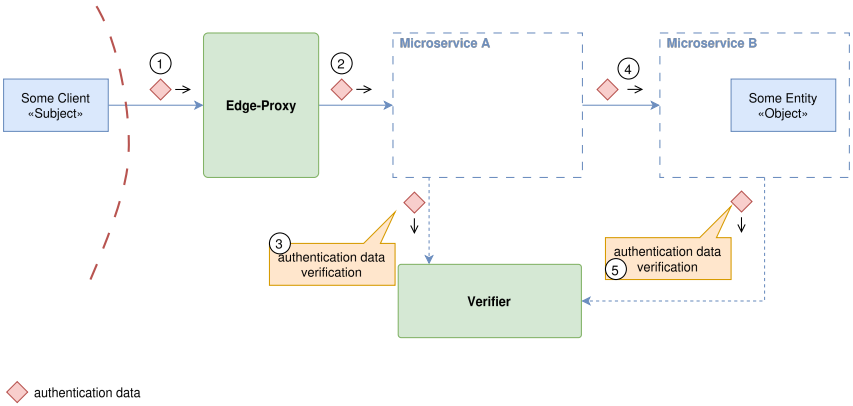

Figure 3.1: External Identity Propagation

such as a JWT[2], the service needs the corresponding key material to verify its signature, and so on.

Pros

- **Minimal edge logic required:** The edge mainly forwards the authentication data, reducing its complexity. It may also just verify the validity of the authentication data.
- **No additional infrastructure needed:** Internal services use the same authentication data as the edge, avoiding the need for internal signing or identity transformation.

Cons

- **Tight coupling to external protocols:** Each microservice must understand and correctly handle potentially multiple types of external authentication data and formats (e.g., OAuth 2.0[3], OIDC[4], cookies). As a result, services must

[2] https://www.rfc-editor.org/rfc/rfc7519
[3] https://www.rfc-editor.org/rfc/rfc6749
[4] https://openid.net/specs/openid-connect-core-1_0.html

support protocol-specific logic (e.g., JWT[5] parsing, OAuth 2.0 token validation, cookie decoding) and are exposed to external semantics, expiration rules, and revocation mechanisms, increasing implementation complexity and brittleness. Changes to external identity providers or protocols typically break internal service behavior.

- **Increased security risk:** If external authentication data is leaked, any internal service exposed, intentionally or not, can potentially be accessed directly using the leaked token.
- **Unsuitable for Zero Trust[6] or multi-tenant environments:** Trust assumptions and lack of verifiability conflict with the security guarantees required in these environments.
- **Privacy concern:** Because externally visible authentication data is reused internally, identifiers intended for internal use (e.g., subject IDs in JWTs) may become externally observable. This can violate privacy requirements by enabling cross-context linkability and may conflict with regulations such as the GDPR or the CCPA.

3.2 Simple Service-Level Identity Forwarding

This pattern builds on the previous one but introduces a lightweight form of internal identity abstraction. While the edge component still forwards the externally received authentication data (e.g., an access token, ID token, session cookie, or certificate) to internal services, each microservice no longer forwards this data unchanged. Instead, a microservice extracts the relevant identity information (e.g., user ID, roles, scopes) from the incoming request and creates a simplified representation of the identity, such as a plain JSON object, a self-signed JWT, or even a single value embedded in a query or path parameter, when making calls to downstream services. As with the previous pattern, the verification of the initially received authentication data may require contacting a Verifier, which depending on the authentication protocol and data used, could be an authorization server

[5] https://www.rfc-editor.org/rfc/rfc7519
[6] https://csrc.nist.gov/pubs/sp/800/207/final

that issued the token or, for example, an OCSP responder to check the revocation status of a certificate.

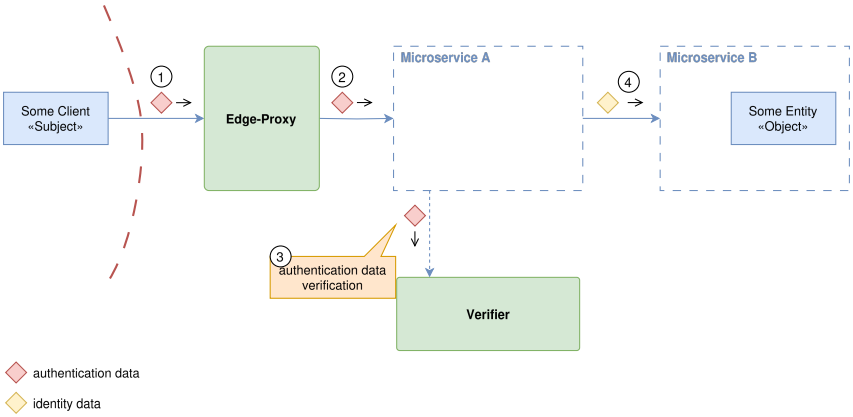

Figure 3.2: Simple Service-Level Identity Forwarding

This internal identity representation is not strongly cryptographically protected and often relies on implicit trust between services. As a result, downstream services must trust the integrity and correctness of the identity information forwarded by their upstream callers.

As with the previous pattern and as also said above, the actual verification of the received authentication data, represented by the dotted line in steps 3 of the diagram above, depends on the type of authentication data used. For example, in the case of an opaque token, the service must call the appropriate identity provider, respectively, authorization server endpoint to retrieve the associated data. If the token is self-descriptive, such as a JWT, the service needs the corresponding key material to verify its signature, and so on.

Pros

- **Simple and lightweight:** Requires minimal implementation effort and no complex cryptography or signing infrastructure.

- **Protocol abstraction:** Internal services operate on simplified identity representations, avoiding the need to parse or validate external authentication protocols.
- **Flexible identity forwarding:** Enables propagation of identity context without dependency on a central trusted issuer for every internal call.

Cons

- **High trust requirement:** Downstream services must trust upstream callers to provide unaltered and accurate identity information and related data.
- **Vulnerable to spoofing:** Lack of cryptographic protection makes identity data susceptible to tampering.
- **Unsuitable for Zero Trust[7] or multi-tenant environments:** Trust assumptions and lack of verifiability conflict with the security guarantees required in these environments.
- **Protocol complexity leakage:** If any internal service becomes externally exposed, support for full external authentication mechanisms is required to avoid API abuse.
- **Privacy concern:** Because externally visible authentication data is reused internally, identifiers intended for internal use (e.g., subject IDs in JWTs) may become externally observable. This can violate privacy requirements by enabling cross-context linkability and may conflict with regulations such as the GDPR or the CCPA.

[7] https://csrc.nist.gov/pubs/sp/800/207/final

What this pattern is really about

This pattern has a tendency to invite risks commonly associated with Insecure Direct Object References (IDOR)[a] resulting in data exposure. I hope I'm not stepping on anyone's toes – but I just have to say it.

It's easy to fall into the trap of passing user IDs through URLs, headers, or JSON payloads without verification – trusting upstream services entirely, with no signatures, no integrity checks, and no questions asked by downstream systems. While this might seem like a convenient internal optimization, it creates a fragile foundation.

Often the response is:

But it's internal – what should happen?

This is a common industry standard. Everyone does it.

Well... things do happen. Remote code execution, lateral movement, privilege escalation, and data leaks are all real possibilities once an attacker gains a foothold. *Internal* is not a security boundary – and never was. If that assumption breaks down – and it often does – these design choices can amplify the damage.

That said, if you choose it, go in with open eyes – and be honest about the trade-offs. You now know them.

[a] https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html#introduction

3.3 Token Exchange-Based Identity Propagation

This pattern builds upon the previous pattern by introducing a trusted intermediary, an authorization server, through use of the OAuth 2.0 Token Exchange[8], or the new OAuth 2.0 Transaction Tokens (draft)[9] protocol. A microservice that receives a request containing externally issued identity (e.g., an access token) exchanges it for a new, signed access token issued by the authorization server. This exchanged token is specifically scoped for a downstream internal service and is then propagated as part of the internal call. As with the previous patterns, the verification happens optionally with the help of a Verifier. The issuance of a new token is, however, the responsibility of the Secure Token Service (STS). The latter assumes the role of the Verifier for the verification of tokens it has issued. Both might be implemented by the same authorization server, but don't need to.

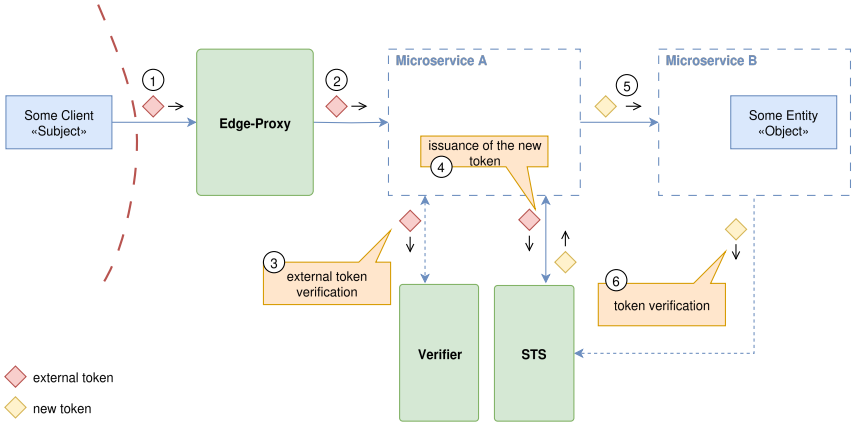

Figure 3.3: Token Exchange-Based Identity Issuance

Downstream services trust the token issued by the STS rather than the one used by the external client ("Some Client" in the diagram above). The pattern improves

[8] https://www.rfc-editor.org/rfc/rfc8693

[9] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-transaction-tokens

the trust model and strengthens identity guarantees, but is tightly coupled to the OAuth 2.0[10] protocol family and its associated token types.

The actual verification of all involved tokens, represented by the dotted lines in steps 3 and 6 of the diagram above, depends on the type of the token used. For example, in the case of an opaque token, each service must call the appropriate identity provider endpoint to retrieve the associated data. If the token is self-descriptive, such as a JWT, the service needs the corresponding key material to verify its signature.

Pros

- **Improved trust model:** Downstream services do not need to trust upstream service implementations, only the STS.
- **Cryptographically verifiable identity:** Issued tokens are signed by an STS, offering strong integrity guarantees.
- **Scoping and audience control:** Exchanged tokens can be restricted in scope and audience, reducing the risk of token misuse.

Cons

- **OAuth 2.0-specific:** Relies on OAuth 2.0 Token Exchange[11], respectively, on OAuth 2.0 Transaction Tokens (draft)[12], limiting its applicability to systems using that protocol family for externally visible authentication data.
- **Service-side complexity:** Application code must integrate with the STS to handle token exchange logic, and manage caching or retries.
- **Latency overhead:** The token exchange process introduces additional network round-trips per request flow unless aggressively optimized.
- **Operational dependency on the STS:** Introduces runtime dependency on the STS implementation availability and scalability.

[10] https://www.rfc-editor.org/rfc/rfc6749
[11] https://www.rfc-editor.org/rfc/rfc8693
[12] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-transaction-tokens

3.4 Protocol-Agnostic Identity Propagation

A note on terminology

By the way, Netflix refers to this pattern as "Token Agnostic Identity Propagation" in this blog post[a], which is a great name. That said, I've often found that as soon as the word "token" comes up, people instinctively think of OAuth 2.0 or OIDC, even though tokens can also refer to cookies, certificates, or other artifacts. To avoid that confusion, I chose a more neutral name here. Naming is hard.

Small advertisement: If you're looking for an off-the-shelf option that supports this pattern (and several others described in this primer), feel free to check out my project heimdall[b]. It goes well beyond this specific approach and is designed to be flexible and extensible for various scenarios.

[a] https://netflixtechblog.com/edge-authentication-and-token-agnostic-identity-propagation-514e47e0b602

[b] https://github.com/dadrus/heimdall

The external request is authenticated at the system edge by a trusted component, which then generates a cryptographically signed (and/or encrypted) data structure representing the external entity's identities and attributes (e.g., user ID, roles, permissions) – typically a self-contained, verifiable structure, such as a JWT or a proprietary signed format. By doing so, the edge component assumes the role of a Secure Token Service (STS). This signed identity structure, hereafter referred to as a token, is propagated downstream to internal microservices. Internal services trust the signature from the edge issuer and use the token to make access control decisions.

As with the previous pattern, the verification of the original authentication data may require contacting a Verifier. The implementation of the Verifier depends on the protocol and data format used – e.g., it could be an authorization server that issued a token, or it could be an OCSP responder, used to check the revocation status of a certificate. Unlike in previous patterns, only the edge component is

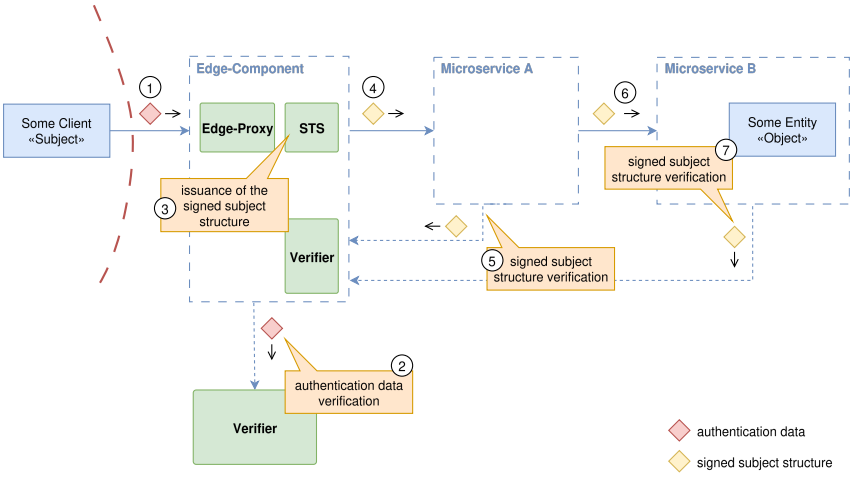

Figure 3.4: Protocol-Agnostic Identity Propagation

responsible for that verification. The specific verification process depends on the aforesaid type and format of the authentication data, denoted by the dotted line in step 2.

Further downstream, the microservices validate the signed token issued by the trusted edge-component. Each microservice must have access to the corresponding verification key to validate the authenticity of this token. The corresponding verification steps are denoted by the dotted lines in steps 5 and 7. This is where the trusted component at the edge assumes the role of a Verifier.

It's worth noting that the edge-component roles shown in the diagram above — Edge Proxy, STS, and Verifier — may all be implemented within a single technical component, or split across multiple cooperating services. For example, a proxy might delegate the authentication data and token issuance related logic to another service via a mechanism typically named as *forward auth* or *external auth*. That service could implement the STS and the Verifier logic by itself, or, in turn, delegate token issuance to an existing authorization server using mechanisms such as the OAuth 2.0 Token Exchange[13], as described in the previous pattern.

[13] https://www.rfc-editor.org/rfc/rfc8693

Pros

- **Cryptographic trust:** Signed tokens provide strong guarantees about the integrity and authenticity of the propagated identity.
- **Decoupling from external authentication data and context:** Internal services neither handle external protocols nor need to differentiate whether requests originate from first- or third-party actors, simplifying their logic and trust assumptions.
- **Rich identity context:** Allows inclusion of fine-grained identity and authorization metadata.
- **Secure across trust boundaries:** Suitable for multi-tenant and Zero Trust[14] environments.
- **Separation of external and internal identities:** Enables mapping externally known identifiers to distinct internal representations, preventing direct exposure of internal identifiers and thereby enhancing privacy by reducing correlation and tracking risks across domains.

Cons

- **Key management complexity:** Requires secure handling and rotation of signing keys to maintain trust.
- **Token size overhead:** Signed tokens issued by the edge component may be large, increasing network overhead.
- **Revocation challenges:** Once issued, tokens may be valid for many services until expiration, complicating immediate revocation. This can, however, be mitigated by issuing short-lived tokens and tailoring subject structures to individual downstream services.
- **Increased complexity at the edge:** The edge component must handle external authentication data verification as well as internal token generation and signing, making it a critical security component.

[14] https://csrc.nist.gov/pubs/sp/800/207/final

3.5 On Privacy By-Design

Privacy concerns – particularly around cross-context linkability and the risk of exposing internal identifiers – affect all identity propagation patterns, though their severity depends on how externally received authentication data is handled.

Implementation of patterns like **External Identity Propagation** and **Simple Service-Level Identity Forwarding** typically directly reuse externally visible authentication data within the system. This increases the risk that internal identifiers (e.g., `sub` claims in JWTs) become externally observable, enabling correlation of user activity across contexts. Such reuse undermines core privacy goals like pseudonymisation and data minimisation and conflicts with principles of integrity and confidentiality – all central to privacy-by-design thinking.

In contrast, patterns like **Token Exchange-Based Identity Propagation** and **Protocol-Agnostic Identity Propagation** help enforce privacy boundaries by transforming or isolating authentication data before it's used internally. That doesn't mean these patterns – or their specific implementations – are immune to privacy risks. They simply make it easier to adopt techniques such as opaque tokens, session-referencing cookies, or identifier mapping, which reduce unnecessary exposure of user-specific identifiers. Even so, mapped identifiers can still reveal the existence of a persistent relationship with the system, which may be problematic in certain contexts. Still, these patterns embody privacy-by-design principles more effectively – and as a positive side effect, tend to align well with legal requirements such as the GDPR (Art. 5(1)(b, c, f), Art. 25, Art. 32, Recitals 26 and 30), CCPA, and similar frameworks.

4 Authorization Patterns

While some basic access control can be applied to anonymous or unauthenticated subjects, the most meaningful authorization requires a reliable understanding of the subject's identities and associated attributes. Having covered these foundational topics in previous chapters, we now turn to how access control decisions are made and enforced across services in distributed systems.

The corresponding architectural approaches can be described by authorization patterns. These patterns define where Policy Decision Points (PDPs), Policy Enforcement Points (PEPs), and Policy Information Points (PIPs) are placed within a system and how they interact. They also govern how subject and object identities, along with related attributes, flow between these components – and where policies are stored and accessed.

Choosing the right patterns is critical, as it directly impacts the system's security posture, performance, scalability, and maintainability. The following subchapters explore the most common ones used in distributed architectures and outline their trade-offs.

4.1 Decentralized Service-Level Authorization

In this pattern, most of the functional components from the reference architecture are implemented directly within each microservice. Even the Policy Information Points (PIPs) may be embedded into the service logic (e.g., via database or configuration entries) if the microservice is responsible for all relevant attributes itself. However, this is rarely the case, and most microservices must integrate with other services to retrieve required attributes, treating those other services as external PIPs.

The access control rules are typically implemented using native language constructs (e.g., `if`/`else` statements), either inline with business logic functions or via abstraction mechanisms such as interceptors.

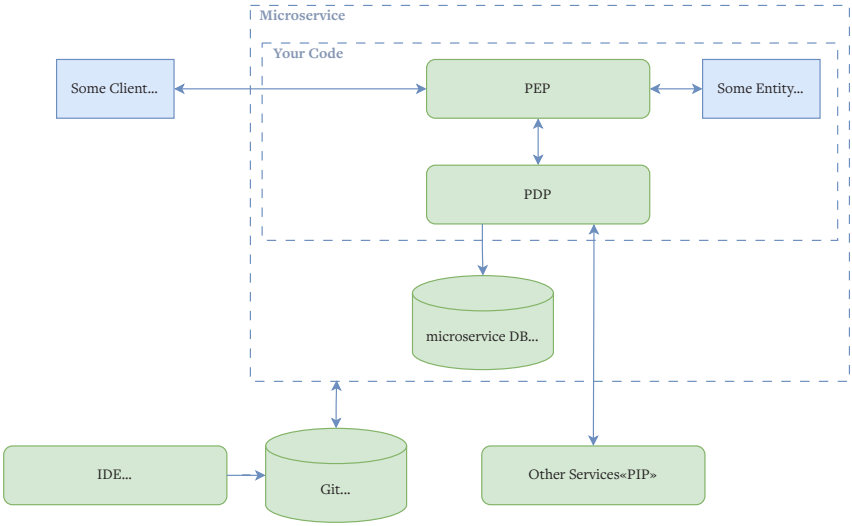

Figure 4.1: Decentralized Service-Level Access Control

When a microservice receives a request containing authorization data (e.g., end-user context or resource identifiers), it evaluates whether access should be granted. This may involve querying other services (PIPs) for additional attributes before reaching a decision and enforcing it (implicitly). Alternatively, some services may use asynchronous communication patterns (e.g., periodic syncs or event-driven updates) to pre-fetch required data in advance, improving performance and resilience.

When adopting this approach, the following trade-offs should be considered:

Pros

- **Familiar development model**: Developers can use the same language and tools they already know.
- **Framework support**: Many libraries and frameworks exist for many languages to reduce boilerplate and simplify integration.
- **Rapid prototyping**: Policy logic is implemented directly in code, enabling quick experimentation and iteration.

- **Team autonomy**: Fits well with independent team ownership; each team can choose its approach.
- **High performance**: Policy evaluation is done in-memory within the microservice.
- **Full context awareness**: The service has access to runtime data, business logic, and domain models, enabling fine-grained, context-rich and nuanced decisions.
- **Failure isolation**: If all required attributes are available locally or cached, failures in external systems do not impact decision-making.

Cons

- **Scattered logic**: Authorization requirements tend to spread across multiple services, leading to code duplication, increased complexity, and maintenance overhead. Over time, this results in a slow and error-prone policy lifecycle, significantly reducing time to market. This is a classic "Hardcoded Rules" antipattern.
- **Role explosion**: Business stakeholders typically describe authorization requirements using roles – for example, "a user with role X can do Y". Without introducing an abstraction layer between business roles and the actual implementation, systems often accumulate many similar but inconsistent roles. Roles also tend to evolve or change names over time. This leads quickly to role explosion, again slowing the policy lifecycle and increasing the risk of errors. This is known as the "Code Against the Role" antipattern.
- **Weak governance**: Autonomous teams may interpret and implement policies differently, making consistent governance for the whole environment nearly impossible. This may result in enforcement gaps and unpredictable behavior.
- **No central auditability**: When authorization logic is distributed across services, it becomes nearly impossible to answer "before-the-fact" questions such as "Who has access to what, and when?" – a key requirement in compliance and security contexts.
- **Inconsistent monitoring**: Logging and audit trails vary widely across services and are often incomplete or incompatible. This hampers the ability to detect abuse, investigate incidents, or analyze system-wide access patterns.

- **Coverage gaps**: Many frameworks do not expose ways to integrate access control into certain auto-exposed endpoints. Teams may also forget to secure these paths entirely. Documentation of the frameworks is also often inconsistent or misleading. All of that leads to unintended public exposure of sensitive endpoints.

These cons often lead to "accept by default" behavior, ultimately leading to broken access control vulnerabilities.

4.2 Centralized Service-Level Authorization

This pattern aims to address the first three drawbacks of the previous pattern – to reduce complexity, improve time to market, and establish governance over policy definitions – by decoupling policy logic from service code and supporting its own lifecycle management. In this model, authorization rules are defined independently of the microservice code. This separation allows policies to be reviewed, versioned, and audited without being tied to the specific implementation languages of the microservices. These policies can reside in a dedicated policy repository, which explains the "centralized" in the pattern name, or they can be colocated with the service code in the same repository. The essential aspect is that policies are decoupled from the service code, rather than intertwined with it. The actual enforcement of the access decisions still takes place locally to each microservice.

The PDP can be implemented as a library (e.g., Casbin[1]) embedded in the service's codebase, as a local sidecar process (e.g., Open Policy Agent[2]), or even be external, centrally managed PDP – shared across a domain (in Domain Driven Design sense), scoped to a business unit, or truly central depending on organizational needs. Authorization rules are now defined using the PDP's domain-specific language (e.g., Rego in the case of OPA), rather than being hardcoded into the service logic.

[1] https://casbin.org/
[2] https://www.openpolicyagent.org/

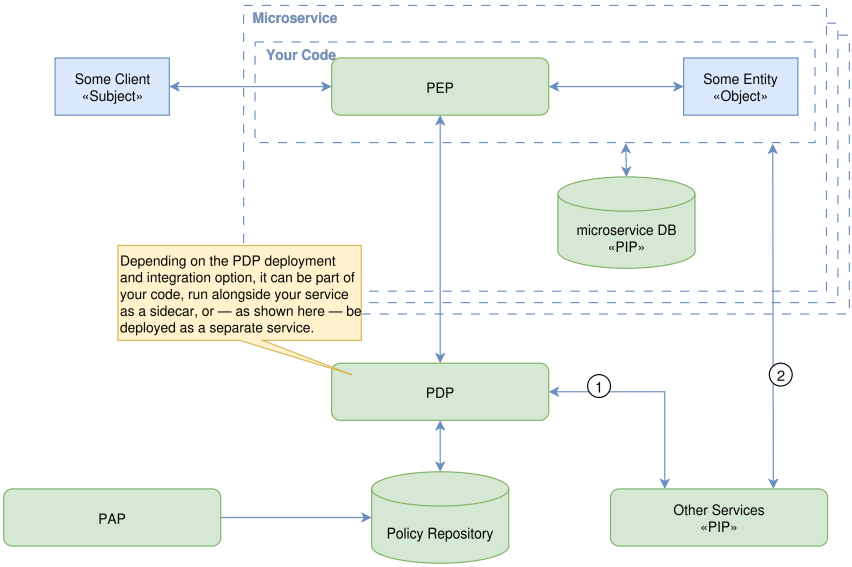

Figure 4.2: Centralized Service-Level Authorization

The microservice continues to act as the PEP, calling into the PDP to make access decisions during request handling. To make an authorization decision, the PDP requires attributes, which – depending on the PDP deployment options mentioned above – may either be available within the service or retrieved from external sources (PIPs). Some PDPs support data-fetching logic within the policy itself, allowing them to directly retrieve the necessary attributes at runtime. This is represented by 1 and 2 in the diagram above. Both connections are conceptual and represent logical communication paths.

Although this pattern significantly improves the maintainability and consistency of access control logic, it also introduces new challenges and does not resolve all the limitations inherent in the previous pattern. It's important to note that aspects such as performance, failure resilience, and auditability – including support for "before-the-fact" audit – largely depend on the type of PDP and its integration approach (e.g., embedded, sidecar, or external). These trade-offs are discussed separately in **PDP Deployment & Integration Options**.

Pros

- **Policy governance:** Policies can be centrally defined, versioned, reviewed, and audited, independent of the service's implementation language.
- **Policy layering:** The model allows for both global (e.g., security team–defined) and local (e.g., service team–defined) policies to coexist. This enables clearer separation of concerns and better alignment with organizational structure and responsibilities.
- **Improved monitoring:** All decisions can be consistently logged and monitored, assuming proper instrumentation.
- **Team autonomy:** Teams remain responsible for their services and their policies, with local enforcement and minimal external dependencies. This aligns well with independent team ownership and domain-driven design principles.
- **Enhanced testability:** Authorization logic can be tested independently of the microservice business logic.

Cons

- **Policy distribution complexity:** Policies are now decoupled from the code, so mechanisms are needed to deploy the correct version of each policy to the appropriate service instances.
- **Context sharing:** PDPs do not inherently have access to the microservice context. Developers must design mechanisms to assemble and pass the right attributes into the PDP for evaluation.
- **Coverage gaps:** Some frameworks expose endpoints by default, often without offering hooks for policy enforcement. Teams may also just forget to add the required logic to some endpoints. Combined with poor or misleading documentation, this can result in unintentionally exposed functionality and missed access control. Common examples include health and metrics endpoints (e.g., Spring Boot Actuator), auto-generated documentation routes (e.g., FastAPI or OpenAPI UIs), or static routes in frameworks like e.g., Express.js.
- **Incomplete enforcement observability:** While policy decisions are consistently logged, there's often no visibility into whether those decisions were correctly enforced across all code paths. Missing instrumentation or scattered enforcement logic makes it difficult to validate effective protection, investigate incidents, analyze system-wide access patterns or detect abuse.

Due to these remaining gaps, “accept by default” behaviors remain a real risk, leading to broken access control vulnerabilities.

4.3 Edge-Level Authorization (Classic)

This pattern aims to address several shortcomings of service-level access control patterns, particularly inconsistent enforcement, policy sprawl, and limited observability. Instead of tying PEP-related logic in each service, access control is moved to the system’s perimeter – typically implemented via API gateways, ingress controllers, or reverse proxies.

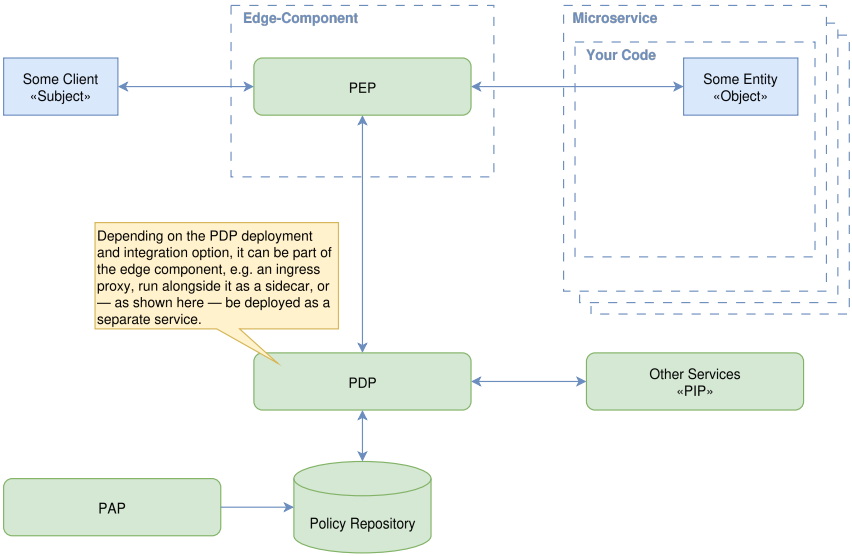

Figure 4.3: Edge-Level Authorization (Classic)

Since authorization must follow authentication, this pattern tightly couples authentication and authorization at the network boundary. Gateways or proxies serve as the PEP, and either evaluate policies locally, using embedded logic, or delegate decisions to an external PDP.

All external traffic flows through the edge component, making this the first pattern that guarantees every inbound request is observed and subject to access control logic. As with the previous pattern, aspects such as performance, failure resilience, and auditability are not covered here but are discussed in **PDP Deployment & Integration Options** instead.

Pros

- **Consistent enforcement:** All inbound requests pass through a centralized enforcement point, ensuring uniform application of policies and reducing the likelihood of unprotected endpoints ("no accept by default").
- **Policy governance:** Policies can be centrally defined, versioned, reviewed, and audited, independent of the service's implementation language.
- **Policy layering:** The model allows for both global (e.g., security team–defined) and local (e.g., service team–defined) policies to coexist. This enables clearer separation of concerns and better alignment with organizational structure and responsibilities.
- **Best observability:** All external access attempts are visible and can be logged centrally, supporting effective monitoring, alerting, and forensics.

Cons

- **Socio-technical challenges:** In many organizations, API gateways are operated by infrastructure or platform teams, meaning development teams cannot directly manage authorization policies or authentication configurations. This separation of responsibilities requires close coordination between developers and operations/security, which often reduces delivery velocity due to communication and process overhead, especially in complex ecosystems with many roles, evolving access control rules, and the need for flexible authentication flows.
- **Policy distribution complexity:** Policies are decoupled from the code, so mechanisms are needed to deploy the correct version of each policy for the appropriate service instances.
- **Authentication limitations:** Edge components only support a single authentication configuration per listener or route group. Supporting multiple identity providers, per-endpoint authentication flows, or more advanced patterns –

such as dynamic consent, step-up authentication, or conditional logic based on subject actions – is difficult or impossible without custom logic or deep integration.

- **Context sharing:** Edge components only have access to request-level attributes (e.g., headers, paths, IPs). This makes it difficult to evaluate fine-grained, object-level, or business-context-sensitive access decisions.
- **Enforcement blind spots and defense-in-depth violations:** Since the edge only governs ingress traffic, any internal traffic (e.g., service-to-service calls) or network misconfigurations may bypass enforcement entirely – violating the defense-in-depth principle and creating a single point of failure.
- **Single point of failure:** Misconfigurations, or performance degradation can impact large parts of the system at once.

4.4 Edge-Level Authorization (Modern)

This pattern evolves the classic edge-level authorization approach to overcome its key limitations. While enforcement still occurs at the perimeter via proxies or gateways, this approach allows per-service customization through service-specific rules – declarative definitions of how identity and context are gathered, how authorization is performed, and how decisions are propagated – forming explicit *Authorization Contracts*. These contracts manifest as structured, signed data (e.g., JWT claims or enriched signed headers) that the edge proxies or gateways relay to downstream services. This explicit propagation of authorization context ensures that internal service-to-service calls rely on a trusted, verifiable authorization boundary, addressing common concerns around enforcement blind spots and defense-in-depth violations typically associated with edge-only models. By making authorization an explicit API-level contract, teams can confidently decentralize enforcement without creating single points of failure or gaps in access control.

Instead of embedding rigid policy logic or centralizing control in infrastructure teams, this pattern emphasizes composability, autonomy, and observability, enabling each team to define how their endpoints are protected, while still benefiting from centralized governance and enforcement guarantees. As with the previous

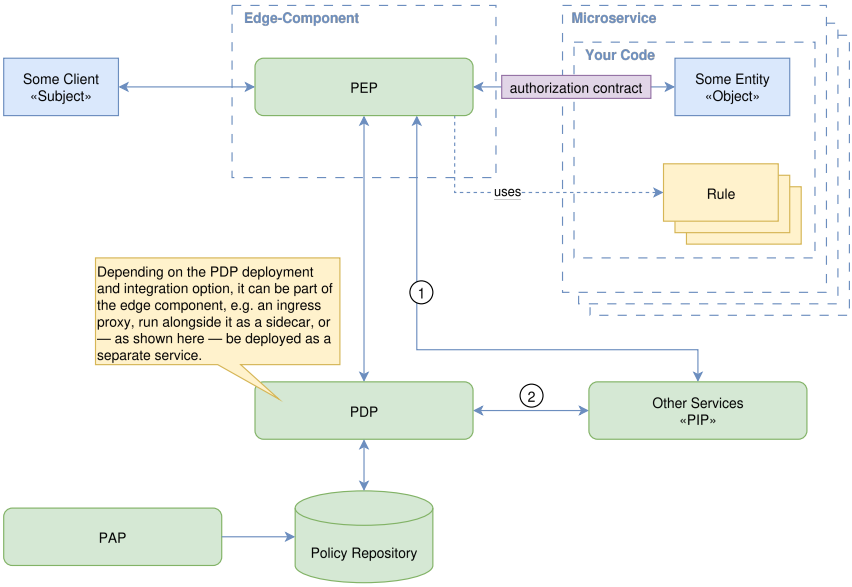

Figure 4.4: Edge-Level Authorization (Modern)

pattern, aspects such as performance, failure resilience, and auditability are not covered here but are discussed in **PDP Deployment & Integration Options** instead.

Pros

- **Consistent enforcement:** Uniform application of policies at a centralized point prevents unprotected or overlooked endpoints.
- **Policy governance:** Policies remain versioned, reviewed, and auditable, often authored centrally but can be referenced declaratively in service-specific contracts.
- **Best observability:** All external access attempts are visible and can be logged centrally, supporting effective monitoring, alerting, and forensics.
- **Rapid prototyping:** Through *Authorization Contracts*, teams can experiment with different authorization models (e.g., embedded JWT claims, header-based roles, etc.) without relying on the infrastructure components.

- **Context sharing:** The proxy can fetch contextual data from arbitrary PIPs, enabling context-sensitive decisions based on domain-specific attributes, object metadata, or subject state.
- **Service autonomy:** *Authorization Contracts* empower microservice teams to define their own access control needs declaratively, supporting domain-driven service ownership without duplicating enforcement logic.
- **Authorization context propagation:** The system can rewrite identity and authorization responses from the PDP into formats that match each service's expectations (e.g., structured JWTs, plain or signed headers), decoupling service-specific logic from authorization protocols.
- **Secure by default:** The use of declarative contracts and centralized enforcement reduces misconfiguration risks and prevents implicit access grants.

Cons

- **Policy distribution complexity:** Ensuring the correct version of a policy is evaluated in the context of the specific service version requires additional coordination. This mainly depends on PDP capabilities and tooling.
- **Contract governance:** While *Authorization Contracts* empower teams with autonomy, it requires clear guidelines and automated validation tools to prevent misconfiguration or misuse.
- **Single point of failure:** Misconfigurations, or performance degradation can impact large parts of the system at once.

4.5 Sidecar-Level Authorization

This pattern is a variant of **Edge-Level Authorization** (**Modern**), where the PEP is not deployed at a shared system perimeter, but instead colocated with each microservice as a dedicated sidecar proxy. All inbound traffic to the service is routed through this proxy, which is responsible for authorization decision enforcement, and propagation of authorization context to the actual microservice.

Conceptually, the sidecar proxy assumes the same role as an edge component in the previous pattern, but at a finer granularity: one enforcement point per service rather than one per environment or cluster. The proxy remains responsible for

enforcing the resulting decisions and for rewriting or injecting authorization contexts (e.g., structured headers or signed tokens) before forwarding the request to the application.

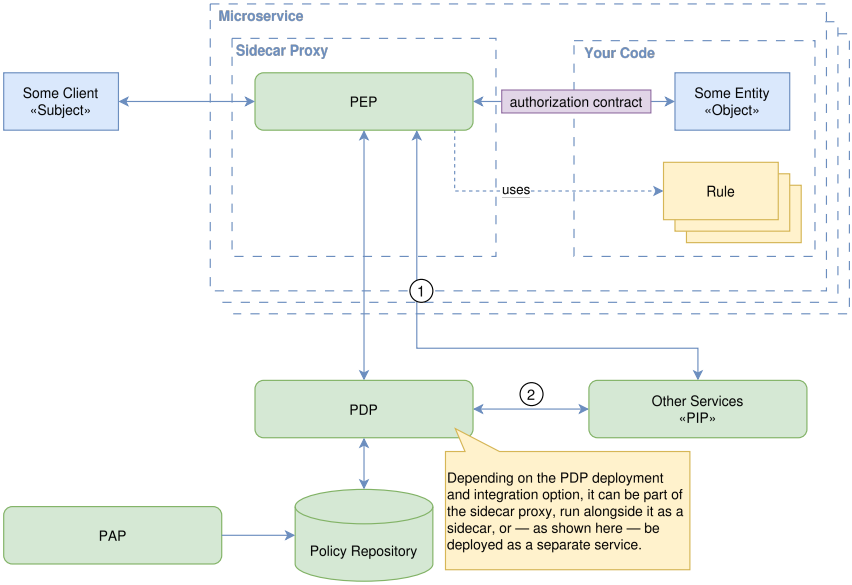

Figure 4.5: Sidecar-Level Authorization

As with the previous pattern, policy evaluation itself may happen locally within the proxy, via an embedded or sidecar PDP, or be delegated to an external PDP service. This approach shares also many of the same advantages and drawbacks as the edge-level model, while reducing the single-point-of-failure characteristics typically associated with centralized edge components.

Pros

- **Consistent enforcement:** Uniform application of policies at a centralized point prevents unprotected or overlooked endpoints.

- **Policy governance:** Policies remain versioned, reviewed, and auditable, often authored centrally but can be referenced declaratively in service-specific contracts.
- **Rapid prototyping:** Through *Authorization Contracts*, teams can experiment with different authorization models (e.g., embedded JWT claims, header-based roles, etc.) without relying on the infrastructure components.
- **Context sharing:** The proxy can fetch contextual data from arbitrary PIPs, enabling context-sensitive decisions based on domain-specific attributes, object metadata, or subject state.
- **Service autonomy:** Microservice teams can define their own access control needs declaratively, supporting domain-driven service ownership without duplicating enforcement logic.
- **Authorization context propagation:** The system can rewrite identity and authorization responses from the PDP into formats that match each service's expectations (e.g., structured JWTs, plain or signed headers), decoupling service-specific logic from authorization protocols.
- **Secure by default:** Since authorization decisions are enforced before requests reach application code, application of this pattern reduces misconfiguration risks and prevents implicit access grants.

Cons

- **Operational complexity:** Each service introduces an additional runtime component, increasing deployment, configuration, and operational overhead.
- **Fragmented observability:** Observability becomes fragmented, since monitoring is limited to individual services unless all services in a given context adopt the same pattern.
- **Resource overhead:** Sidecar proxies consume CPU, memory, and network resources, which can be significant in large-scale environments with many small services.
- **Policy distribution complexity:** Ensuring the correct version of a policy is evaluated in the context of the specific service version requires additional coordination. This mainly depends on PDP capabilities and tooling.

- **Contract governance:** While *Authorization Contracts* empower teams with autonomy, it requires clear guidelines and automated validation tools to prevent misconfiguration or misuse.
- **Partial adoption risks:** If only some services adopt the pattern, security guarantees become uneven, and architectural assumptions about enforcement consistency may no longer hold.

4.6 PDP Deployment & Integration Options

The choice of PDP deployment – embedded, as a sidecar, or external – significantly impacts performance, auditability, and supported authorization models. The table below summarizes the key trade-offs:

Aspect	Embedded PDP	Sidecar PDP	External PDP
Location	as a library	as local sidecar process	separate PDP service
Latency	no impact	very low latency	higher latency due to network hops
Before-the-Fact Audit	limited	limited	possible system-wide
Access Control Models	PBAC, e.g., Casbin	PBAC, e.g., OPA	PBAC, ReBAC, and NGAC (e.g., OPA, OpenFGA, SpiceDB)
Dependencies	none (self-contained)	none (self-contained)	Relies on PDP service availability

4.6.1 On Data Source Integration

The need to fetch or inject data required for policy evaluation introduces operational challenges across all authorization patterns – including **Decentralized Service-Level Authorization**. This responsibility may lie with the PEP (e.g., a service or a proxy) or the PDP itself. Accessing PIPs at runtime can complicate network configurations, conflict with segmentation or firewall policies, and broaden the system's attack surface. These concerns require careful architectural consideration, which is also something the next chapter aims to support you with.

5 Decision Dimensions for Authorization Patterns

The discussion of **Authorization Patterns** might suggest that **Decentralized Service-Level Authorization** should be avoided due to drawbacks such as scattered logic and limited auditability. However, this is not always the case. The suitability of an authorization pattern depends on the system context. This context can be analyzed along several key dimensions that guide the choice of appropriate patterns and help keep the system secure, manageable, and responsive, as outlined in this chapter.

Cost considerations

You might be wondering whether **cost** should be treated as a separate dimension – and I completely agree. After all, we don't do security for its own sake, but to support (or even enable) the business. That's why cost is always present in the background. Throughout this primer, I've tried to reflect that – whether by discussing operational overhead, maintenance effort, or other practical trade-offs.

5.1 Policy Characteristics

Policy characteristics define how policies are authored, maintained, and updated, influencing their management and distribution. Two key dimensions, **ownership** and **change latency**, guide these processes, which are critical for operationalizing authorization systems.

5.1.1 Policy Ownership

This dimension identifies who owns and maintains a particular policy. Ownership matters in two ways: it often correlates with how composable or layered the

policies need to be, and it also defines governance[1] boundaries, determining who is authorized to create, modify, and deploy policies.

- **Microservice Team:** Policies authored and maintained by the team responsible for a specific microservice. These are typically focused on local enforcement logic and closely tied to internal service semantics. For example, a recommendation service defines request filters that exclude certain products based on internal scoring thresholds or active experiments.
- **Domain Level:** Policies shared across services within a business domain, often requiring coordination between teams. These policies may be abstracted and reused across multiple services, like a subscription domain enforces business rules about grace periods, usage limits, or billing thresholds that are referenced by billing, customer portal, and notification services.
- **Central** (**Organization Level**)**:** Policies governed by a central security, compliance, or platform team. These typically apply across domains or services and provide the foundation upon which more granular policies are built, like an organizational policy that defines acceptable data residency constraints or standard access conditions for administrative APIs.

5.1.2 Policy Change Latency

This dimension describes how quickly a policy change must be reflected in the system once introduced. It should not be confused with the **frequency** of policy changes (how often they occur), or with **input data freshness** (how quickly attribute updates must be reflected in policy decisions). While policy change frequency influences governance and authoring processes, the latency dimension defines how fast policies must be deployed and propagated across services to take effect.

[1] The term governance refers to a set of responsibilities and processes that keep authorization reliable. It defines who can create, change, or deploy policies, configurations, and the data those policies rely on. It also includes practical concerns – such as who owns a policy, who can update the PDP or its inputs, which attributes PEPs may send, and how changes are reviewed or audited. Put simply, governance is about keeping policies and data consistent so deployed rules don't break and the system remains reliable and robust. In this article, whenever governance is mentioned, it refers to some or all of these aspects, depending on the context.

- **Immediate:** Policies must take effect as soon as they are changed (seconds to minutes). Example: A financial system introduces a temporary block on a specific payment method due to detected processing errors. The rule itself (block this method) must be enforced immediately across all services to prevent further transactions.
- **Fast:** Policies should be applied within hours to days. Example: A sales team requests an update to discount eligibility rules for enterprise customers. Once approved and authored, the new policy should be effective by the next business day.
- **Delayed:** Policies can be applied on a longer timescale (weeks or more). Example: A data retention policy update mandated by new legislation is scheduled for enforcement with the next release.

5.2 Policy Distribution Strategies

The **policy change latency** dimension, described in the previous chapter, defines how quickly policy changes must take effect once introduced. These latency requirements directly influence how policies are delivered to PDPs to ensure they are available for evaluation in the system, which is what this chapter addresses. Here, we discuss the two primary strategies along with their respective trade-offs.

5.2.1 Out-of-Band Delivered Policies

Policies are proactively sent to the PDP and stored locally for evaluation. This strategy suits policies that tend to have immediate to fast change latencies, requiring agile, incremental updates without disrupting service availability.

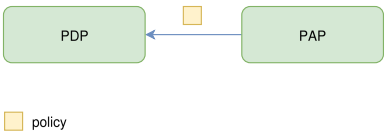

Figure 5.1: PAP sends policy to the PDP

Pros

- Enables applying policy changes dynamically without redeploying PDPs, supporting high availability.

Cons

- Requires robust synchronization mechanisms to deploy the correct versions of required policies to each PDP instance.
- Demands governance mechanisms to ensure that policy deployment aligns with ownership boundaries. E.g., microservice teams should only be able to update their own policies, while domain or central teams retain control over shared or organizational policies.

This is where **policy ownership** becomes a critical factor, as it directly shapes the enforcement and governance model for policies and who is allowed to deploy which one.

5.2.2 Embedded Policies

Policies are embedded directly within the PDP (e.g., as code, or as static configuration) and cannot be updated without restarting or redeploying the PDP. This approach is best suited for policies that have delayed change latencies. Stability and operational simplicity are typically prioritized over agility in such cases.

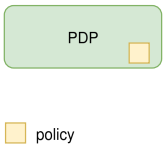

Figure 5.2: Embedded policy

Pros

- Simplifies policy management, as policies are bundled with the PDP.

Cons

- Increases deployment overhead, as changes involve rebuilding and/or redeploying the PDP.
- Limits scalability for needs with frequent policy adjustments.
- Introduces governance challenges, since the team deploying the PDP effectively decides which policies get bundled and activated, even if those policies are owned by different teams or organizational units.

5.3 Data Characteristics

Data characteristics define how information used during policies evaluation is sourced and managed. The first subchapters focus on the input side and introduce three key dimensions: **input data locality**, **input data cardinality**, and **input data freshness**. The last chapter covers the characteristics of the output data – the **output data cardinality**.

Locality describes the scope within which data is relevant and shared, cardinality determines how much data must be managed, and freshness defines how often that data must be refreshed or fetched in real time. Taken together, these dimensions shape the feasibility and efficiency of authorization system design.

5.3.1 Input Data Locality

Locality defines the boundaries of data relevance and reuse, from tightly scoped to broadly shared:

- **Service-Local Data:** Data relevant only within a single service, not reused elsewhere. For example, service-specific configuration flags affecting authorization decisions only inside that service, or ephemeral session attributes used exclusively by the service's internal logic.
- **Domain-Level Data:** Data shared across multiple services within the same bounded context or domain. Examples include ownership metadata of documents in a document management domain, customer account status (e.g.,

frozen, active, under review) used by both billing and support services, or time-based availability windows for booking or scheduling services.
- **Organization-Level Data:** Relevant across domains or the entire system, such as regulatory classification of data (e.g., "EU personal data"), tenant-level subscription tier or plan.

5.3.2 Input Data Cardinality

Cardinality refers to the number of distinct attributes across all subjects or resources. It determines how easily data can be cached or distributed in an access control systems.

- **High:** Many distinct data items, often tied to individual requests or users (e.g., a real-time risk score or geoip information).
- **Medium:** Moderate number of distinct data items typically shared across sets of subjects or resources (e.g., project IDs).
- **Low:** Few distinct data items. For example, environment labels (e.g., "production", "staging"), or business unit identifiers (e.g., "HR", "Finance", "R&D").

5.3.3 Input Data Freshness

This measures the maximum acceptable delay between an attribute value changing, and that change being reflected in authorization decisions.

- **High:** Changes must be reflected immediately or within seconds to maintain accurate authorization (e.g., real-time risk scores, breach detection flags).
- **Medium:** Changes should be reflected within minutes to hours, balancing freshness and performance (e.g., feature toggles, subscription tiers).
- **Low:** Changes can be reflected with delays of hours to days without significant impact.

5.3.4 Output Data Cardinality

As written in the story chapter, policy decisions often include more than just simple `"permit"` or `"deny"` responses. They may carry **structured outputs** that

shape the final data set accessible to a subject – such as lists of permitted object IDs, query filters, or advices.

These outputs fall into two broad categories:

- **Metadata**: Optional guidance or instructions to the PEP (e.g., log this access, display a warning).
- **Decision Data**: The core result of policy evaluation – potentially including constraints, and similar information describing what access is allowed.

While the PDP returns the decision, its structure and size – the **output cardinality** – are defined by the **policy logic**, which reflects the needs of the consuming application. For example, if an application must render only the documents a user is allowed to see, the policy may be implemented to return a list of permitted document IDs, increasing output cardinality.

That way, the output cardinality can be grouped into three levels:

- **Low**: Simple decisions with minimal metadata, such as `{ "result": true }` or `{ "decision": "permit" }`.
- **Medium**: Decisions include multiple structured attributes or small lists. Example: `{ "allowed_projects": ["A", "B"] }`.
- **High**: Large or complex result sets, such as thousands of object IDs. These often require pagination or streaming. Example: `{ "resources": ["doc1", "doc2", ..., "doc5000"] }`.

5.4 Policy Input Data Distribution Strategies

While the **input data freshness** dimension defines how quickly data changes must be reflected in access control decisions, **input data cardinality** can, depending on the PDP type, limit how much information can be stored or cached in practice, and with that also the ability to fully achieve that reflection. This challenge is especially relevant in PBAC systems.

This tension highlights a broader challenge for all approaches relying on embedded or external PDPs: how to make the right data available at evaluation time without overwhelming the system. To address this challenge, different strategies for distributing input data to PDPs have emerged. Each comes with distinct trade-offs, and their suitability depends on the PDP type (e.g., PBAC, ReBAC, NGAC) as well as on system requirements for performance, scalability, and freshness.

Each strategy addresses different operational concerns, and no single approach works universally. Mature systems often combine them, guided by data characteristics, performance targets, and architectural constraints.

5.4.1 On-Demand Data Pull

The PDP fetches data from PIPs at the time of policy evaluation, typically via APIs or database queries. PDPs supporting this option typically allow for configurable caching of the pulled data.

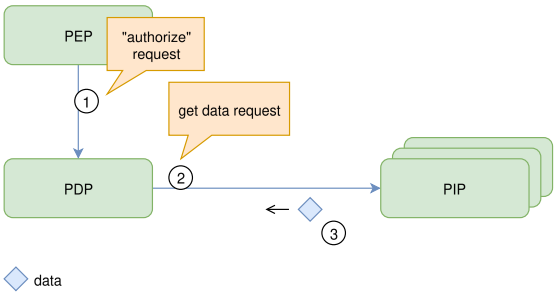

Figure 5.3: On-Demand Data Pull

Pros

- Ensures data freshness by retrieving the latest attributes values from PIPs at evaluation time.
- Enables handling of high-cardinality data without preloading large datasets into the PDP.

- No need for data synchronization mechanisms, since the PDP always queries the source directly.
- Since the PDP does not need to maintain a local copy of data, the memory or storage demand of the PDP is low.
- Governance responsibility is at the policy author – the policy defines where the data is retrieved from.

Cons

- Increases latency due to network calls to PIPs during evaluation, which negatively impacts performance, especially for high-throughput systems.
- Introduces dependencies on external systems, reducing resilience if PIPs are slow or unavailable, potentially leading to cascading failures, degraded service or fallback decisions.
- Limits the usable PDP types, as ReBAC and NGAC implementations typically don't support this strategy.
- Degrades system performance when attributes are accessed repeatedly, especially for high-throughput systems.

While caching (if supported by the PDP) can mitigate some of these drawbacks, it undermines the freshness guarantee, potentially leading to incorrect authorization decisions. Moreover, caching also negates the low-storage advantage listed above – especially for high cardinality data.

5.4.2 Out-of-Band Data Push

Data is proactively sent to the PDP in advance, and stored in memory or a local data store for faster access during evaluation.

Pros

- Improves performance by storing data locally (e.g., in memory or a local database), enabling faster policy evaluation.
- Enhances resilience, as the PDP can operate independently of PIP availability, allowing PDP instances to remain lightweight and focused on evaluation, which improves their scalability.

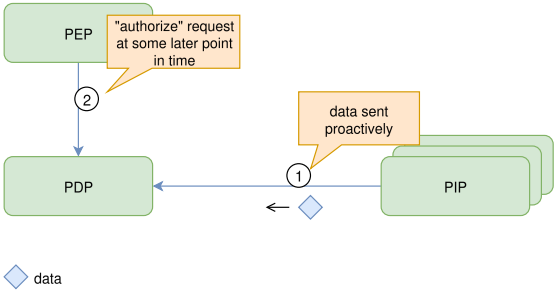

Figure 5.4: Out-of-Band Data Push

- ReBAC/NGAC PDP types typically require access to complete relationship graphs or contextual data sets, which are infeasible to retrieve on-demand or pass inline. This strategy enables those models.
- Reduces load on the PDP by shifting data synchronization to other system components, allowing PDP instances to remain lightweight and focused on evaluation, which improves their scalability.

Cons

- Requires robust data synchronization mechanisms to push updates to the PDP instances in real-time or near-real-time, especially for data with high-freshness requirements.
- Increases memory or storage demands on the PDP, which is usually problematic for high-cardinality data.
- Introduces governance complexity, as mechanisms, who can write to the event/topic the PDP listens to, or who can invoke the PDP's API for updates, and which specific data each party is allowed to send, must be established.

5.4.3 Request-Time Data Injection

Required data is passed directly in the request from the PEP to the PDP – an approach often referred to as *inline data passing*. Early-stage standardization efforts (OpenID AuthZEN Initiative[2]) aim to make this interaction more consistent and interoperable.

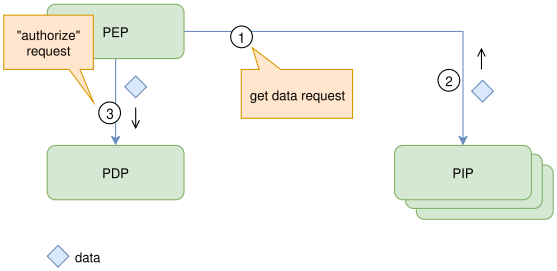

Figure 5.5: Request-Time Data Injection

Pros

- Ensures data freshness by providing the latest attributes values from PIPs.
- Enables handling of high-cardinality data without preloading large datasets into the PDP.
- Reduces load on the PDP by shifting data synchronization to other system components (the PEP), allowing PDP instances to remain lightweight and focused on evaluation, which improves their scalability.
- Since the PDP does not need to maintain a local copy of data, the memory or storage demand of the PDP is low.
- Typically, the only option for ReBAC and NGAC systems to provide attributes which are not stored in their databases.

Cons

- Increases request size, as additional data is included in the decision request, potentially impacting network performance.

[2] https://openid.net/authzen-authorization-api-1-0-implementers-draft-approved/

- Places the burden on the PEP (e.g., microservice or edge component) to collect and validate data from PIPs, increasing complexity in the calling component.
- Risks inconsistent data if the PEP fails to provide all required attributes or if data collection is misconfigured, potentially leading to incorrect decisions.
- Can degrade system performance when attributes are accessed repeatedly by the PEPs.
- Introduces governance complexity, as PEP configuration becomes a concern – it determines which attributes are fetched and sent to the PDP, as changes directly impact authorization decisions.

While caching (if supported by the PEP) can mitigate some of these cons, it introduces the risk of stale data, potentially leading to incorrect authorization decisions.

5.4.4 Embedded Data

Data is baked directly into the PDP's configuration, rather than being pulled or pushed dynamically.

Figure 5.6: Embedded Data

Pros

- Zero runtime dependencies on external PIPs – the PDP is fully self-contained, which simplifies deployments.
- No synchronization concerns; the data is always available and consistent.

Cons

- Useful for static or rarely changing information only (e.g., “environment”: “prod”, “region”: “EU”), and impractical for medium- or high-freshness data.
- Introduces governance challenges similar to those described in embedded policies, as the team deploying the PDP effectively decides which data get bundled and used, even if those data elements are owned by different teams or organizational units.

5.5 Policy Output Data Handling Patterns

As stated earlier, access control requirements often go beyond simple cases like “can subject X read object Y?”. In practice, most requests hitting a PEP involve multiple, context-sensitive decisions. This is especially true for read operations, such as deciding whether to render “edit” or “delete” buttons based on a user’s permissions.

These decisions can often be handled via batch requests, where the PEP sends multiple access queries in a single call, and the PDP evaluates them at once, returning one simple decision per item. However, access requests involving larger output data sets, such as for rendering a list of articles Alice is allowed to see, can, depending on the chosen approach, significantly affect output data cardinality and directly influence the choice of the possible authorization patterns.

The following subchapters describe the typical patterns used in such cases.

5.5.1 PDP as Filter, aka Brute-Force Lookup

In this pattern, the PEP retrieves all potentially relevant data from a PIP (e.g., a database or API) and iterates over each item, querying the PDP to check whether access is permitted. If allowed, the item is included in the final result (e.g., rendered HTML or returned JSON).

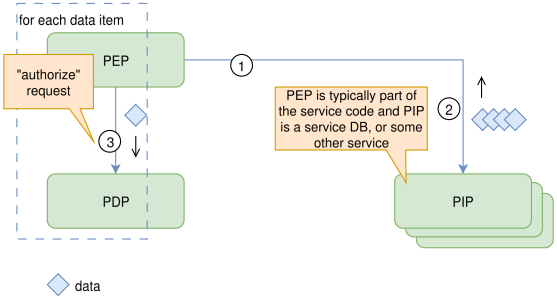

Figure 5.7: PDP as Filter

Pros

- Simple to implement.
- Works with any PDP.
- Easy to debug and monitor.

Cons

- High latency and poor scalability for high-cardinality queries due to repeated PDP calls.
- Increases resource consumption by retrieving more data than needed.
- Tightly couples the PEP with the service's business logic and makes externalizing the PEP (e.g., into a proxy) impossible.
- Changes to access policies typically require service redeployments or even refactorings.

The first two cons might be partially resolved by making use of batch queries if the PDP supports that.

5.5.2 Authorized Data Set

In this pattern, the PEP makes a single request, and the PDP returns a complete set of allowed resources (e.g., object IDs). The PDP constructs this result based on policy logic and available attributes.

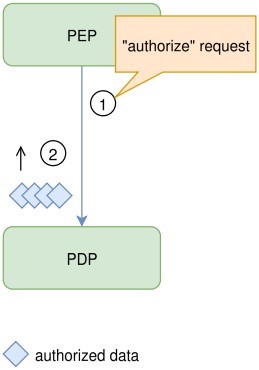

Figure 5.8: Authorized Data Set

Pros

- Reduces round-trips by returning all results at once.
- Simplifies PEP logic, as the PDP handles the complexity of determining the authorized dataset.
- Well-suited for ReBAC or NGAC PDPs, which can leverage internal data models to compute permitted resources.

Cons

- Externalizing the PEP to e.g., an external proxy is only feasible for low to medium cardinality output data sets.
- Might complicate error handling and monitoring of data access.
- Require pagination or streaming for bigger output data sets.
- Results in complex policies for PDPs implementing PBAC approaches.
- Not supported by every PBAC PDP implementation.

5.5.3 Authorization Filter

In this pattern, the PEP calls the PDP, which returns a filter expression (e.g., a SQL WHERE clause, query predicate, or attribute-based condition). The PEP then applies this filter during data retrieval (e.g., in a database query) to fetch only the authorized data.

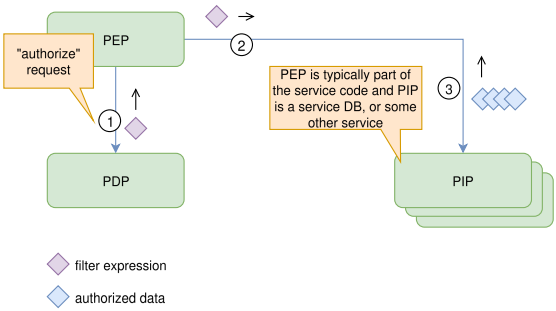

Figure 5.9: Authorization Filter

Pros

- Highly efficient for large datasets – filtering happens at the PIP (data source).
- Scales well with high output cardinality.
- Reduces PDP load.
- PDP does not need to not know all data sets.
- Enables flexible PEP placement – as part of the service, or as an external proxy.

Cons

- Not supported by every PDP (ReBAC und NGAC PDPs do not support that at all).
- Might complicate error handling, monitoring of data access, and diagnosing related issues.

5.6 Performance

Last but not least, performance plays a critical role in the design of authorization systems – especially in latency-sensitive environments. From the end-user's perspective, Time to First Byte (TTFB) is one of the most influential metrics, as described in Phil Walton's article on user-centric performance metrics[3].

[3] https://web.dev/articles/user-centric-performance-metrics

TTFB represents the time it takes for the first byte of a response to reach the client and reflects the perceived responsiveness of a system. It implicitly defines the **latency budget** available for upstream processes – including authorization decisions. This concept is further reinforced by the speed and human perception thresholds[4] discussed in *High Performance Browser Networking*[5].

The following factors strongly influence architecture decisions – such as PDP placement and data handling – and directly impact whether the system can meet that latency budget:

- **Policy evaluation latency:** The time a PDP takes to compute a decision depends on the number and complexity of policies and the input data cardinality – i.e., how many attributes must be evaluated.

- **Data retrieval latency:** When attributes are fetched on-demand (see Policy Input Data Distribution Strategies), latency depends on the number of PIPs involved, the volume of data (input data cardinality), and its locality. This can add significant variability to response time.

- **Policy output handling:** The output data cardinality and the selected output handling pattern affect the time required to process and apply the result.

- **PDP integration overhead:** Overheads include network latency (ranging from ca. 300µs on loopback to >200ms for cross-region communication), DNS resolution, TLS handshake, and data serialization/deserialization. Protocol choices (e.g., HTTP/1.1 vs. HTTP/2 vs. gRPC) can further influence this. For in-process PDPs, these costs are minimized, though serialization costs may still apply.

- **Runtime resource contention:** ("Busy Neighbor" effect) PDPs are typically CPU- and memory-intensive. Co-located resource-hungry processes can significantly degrade performance if compute and memory isolation aren't enforced. This is especially relevant when integrating a PDP into an edge component (either embedded or as a sidecar), which is often optimized for high IOPS

[4] https://hpbn.co/primer-on-web-performance/#speed-performance-and-human-perception
[5] https://hpbn.co/

throughput. In such cases, embedding a PDP introduces trade-offs between CPU-bound policy evaluation and I/O-heavy request processing.

- **Caching and memoization:** Many PDPs implement decision caching or partial evaluation to avoid repeated computation for deterministic inputs. These optimizations can reduce latency but can lead to outdated decisions and require robust cache invalidation logic.

Additional considerations include:

- **Connection reuse and pooling:** Using persistent connections, connection pooling, or multiplexed protocols (like gRPC or HTTP/2) helps amortize integration overhead and reduce connection setup time.

- **Fallback strategies and timeouts:** Systems must decide how to behave when the PDP is slow or unavailable. Strategies such as *fail-closed*, *fail-open*, or *graceful degradation* are architectural decisions that directly impact perceived performance and security posture.

5.7 What's next

I know, this was a pretty heavy part. But it laid the groundwork needed to answer the big question: Which authorization patterns are actually right for you?

All the pieces are already here, but to make the answer more tangible, in the next chapter I'll bring them together into a visual guide – a kind of map or decision tree that makes the connections clearer. From there, we'll build a broader pattern language that connects authentication, authorization, and identity propagation patterns to show how they influence each other. Finally, we'll turn to the practical side: what these choices mean for real-world implementations and which open-source tools can help put them into practice.

6 Practical Considerations & Recommendations

In the earlier chapters of this primer, I've walked you through core concepts, explored many different authentication, identity propagation, and authorization patterns, and we even took a few side steps to uncover what else might be relevant. Along the way, I deliberately avoided making concrete recommendations, because I wanted to ensure you had all the information needed to make educated decisions.

Now, as promised, it's time to put theory into practice. In this last chapter, I'll finally derive recommendations from the insights shared so far and provide practical guidance for implementing authorization, identity propagation, and authentication in real-world microservice systems – a kind of "how-to" for everything we've discussed.

6.1 Authorization Patterns Recommendations

This subchapter maps the decision dimensions to the authorization patterns, using the trade-offs of the corresponding patterns as the primary guiding principle. While multiple patterns may technically be applicable in a given context, some introduce security, operational, performance, or maintenance overheads that make them less desirable in practice. The recommendations below aim to balance these concerns, helping to avoid common pitfalls and promote architectural consistency. Deviations may be valid in specific cases, but should be intentional – not accidental.

The diagram below illustrates the recommended patterns based on the given dimensions.

- If the input data locality required for the decision is service-local, Decentralized Service-Level Authorization is ideal – regardless of other dimensions – since the data's isolated scope avoids all the drawbacks discussed earlier.

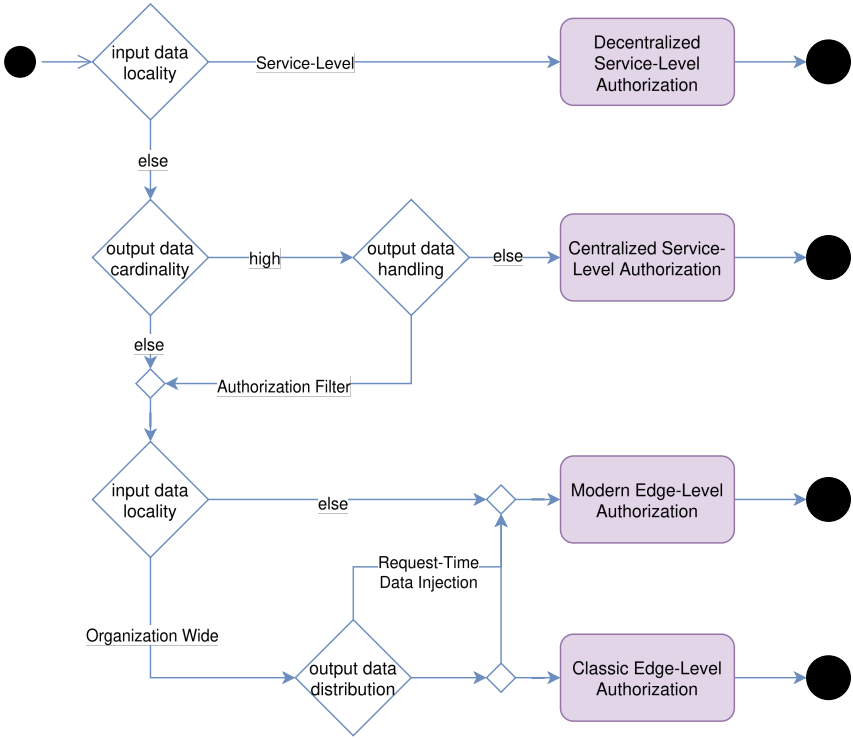

Figure 6.1: Recommended Authorization Patterns

- If the input data locality extends beyond Service-Local – i.e., the same data is shared across multiple services – and the output data cardinality is high while using Authorization Filters is not feasible, then Centralized Service-Level Access Control is better suited to the context.

- In all other cases, Modern Edge-Level Authorization tends to offer the best trade-offs.

- Classic Edge-Level Authorization may still be suitable when input data is organization-wide, output cardinality is low, and all relevant data is either pulled by the PDP at request time at request time or pushed to the PDP out-of-band in advance.

Implications of pattern selection

Pattern selection is not an isolated decision. The described patterns form a broader pattern language, where one pattern often implies or necessitates the use of another. For instance, selecting Modern Edge-Level Authorization introduces the concept of an *Authorization Contract*, which must be verified within each service. These contracts represent service-local data, and verifying them naturally leads to adopting Decentralized Service-Level Authorization inside the respective services.

Example: The Blog Platform

To illustrate how multiple authorization patterns may compose into a coherent solution, let's return to the earlier story of Alice and the blog platform.

The system defines two access requirements:

- **Listing articles:** Every user is allowed to see the list of available articles, including the title, publication date, author, and a short excerpt.
- **Reading articles:** Access to the full content depends on the user's subscription level and the number of full articles already read that day.

These requirements map naturally to different authorization patterns:

- For listing articles, the access logic relies solely on local data stored within the article service. Since the input data is entirely service-local, the Decentralized Service-Level Authorization pattern is ideal – no orchestration or coordination with other services is required.
- Reading a full article, however, requires accessing data managed by multiple services: the subscription service (to verify Alice's plan) and a usage-tracking service (to check her daily quota). Because the input data is not local and the output cardinality is low – the system makes a decision about a single article – Modern Edge-Level Authorization is a better fit. The payload of the *Authorization Contract* introduced here might, for example, look similar to: `{ "sub": "0f4a6554-9069-483d-bc8b-86c6943f22f2", "iat": 1757378963,`

`"requested_article": "<uuid>", "allowed_representation": "<full | excerpt>", ... }`, which then leads to verifying this contract within the article service using Decentralized Service-Level Authorization.

Example: A Document Management System

Let's now shift the service landscape slightly to explore the applicability of the remaining patterns. Imagine Alice now wants to access her employer's document management system.

- **Listing documents:** Users can only list documents related to the projects, they are a team member of
- **Reading documents:** Users can only read documents related to the projects, they are a team member of

These map to the following patterns:

- Listing documents requires access to the project members service. Given the typically high output cardinality, Centralized Service-Level Authorization is the best fit.
- While reading a document could use the same pattern, Modern Edge-Level Authorization is a better fit. It simplifies the implementation of the document-rendering service and ensures that all exposed endpoints – not just the document delivery one – are consistently subject to access control.

Last but not least, the performance requirements and input data cardinality strongly influence the PDP choice – PBAC, ReBAC, or NGAC – and integration approach – embedded vs. external. However, this decision may also be shaped by the available tooling for policy and policy input data distribution – which brings us to the next chapter.

6.2 Data and Policy Distribution in Practice

Building on the concepts introduced in Policy Input Data Distribution Strategies and Policy Distribution Strategies, this chapter demonstrates how the out-

of-band data push and out-of-band delivered policies approaches translate into concrete architectures for real-world PDP deployments. These architectures – whether based on embedded PDPs or standalone PDP services – incorporate specific control-plane components to manage initialization, configuration, and runtime updates, ensuring that PDPs remain synchronized and deliver accurate authorization decisions in dynamic environments.

These control plane components are:

- **Configuration Repository:** Stores the desired configuration for each PDP instance, including detailed references to required policies – such as their repository locations and version information – as well as PIP integration settings, including endpoints, supported protocols, credentials, and other communication-specific parameters.
- **Distributor:** A control-plane component responsible for distributing configuration that enables Aggregators to obtain and apply data and policy artifacts. It retrieves configuration from the Configuration Repository and monitors it for changes. Whenever an Aggregator connects or updated configuration becomes available, the Distributor pushes the applicable configuration to that Aggregator. Depending on the implementation, it may also act as a relay for data updates from PIPs, forwarding only the relevant updates to each Aggregator based on their configured subscriptions.
- **Aggregator:** A control-plane component responsible for configuring a PDP instance with the required policies and data. The Aggregator acts as a client of the Distributor, connecting to it to receive its configuration and any updates. Based on this configuration, it retrieves policies and data from designated sources – policy repositories for policies and PIPs for data – and monitors these sources to ensure the PDP remains synchronized with the desired state. Monitoring of policies depends on the capabilities of the policy repository and typically involves polling. For data updates, Aggregators may either pull directly from PIPs or receive change notifications via decoupled mechanisms such as message buses or webhooks. Event-based delivery is often preferred due to its scalability and resilience, but it's not strictly required.

The following setup illustrates this approach, showing how a PDP can be provisioned with policies and data while supporting runtime updates.

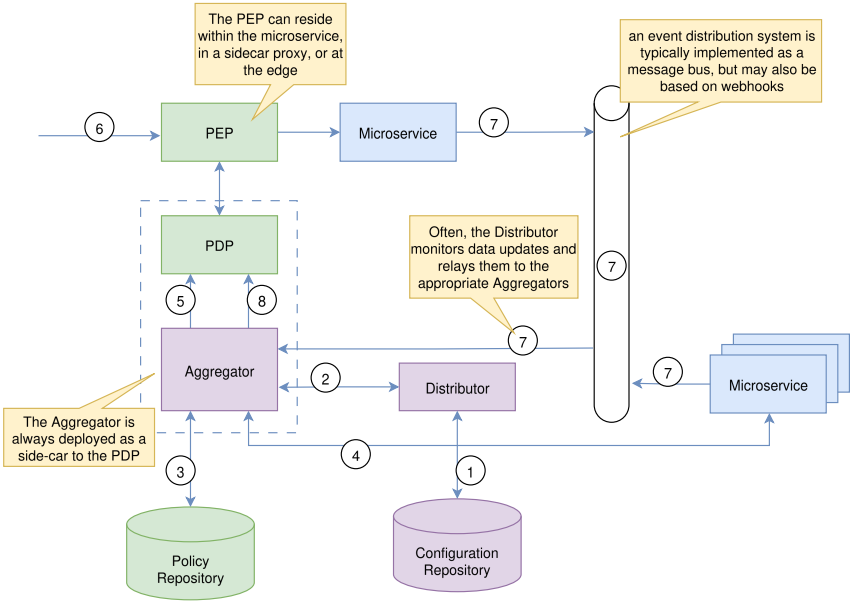

Figure 6.2: Embedded PDP Data & Policy Distribution

1. The Distributor starts, retrieves configurations from the Configuration Repository, and waits for Aggregator connections.
2. An Aggregator starts, connects to the Distributor, and receives its configuration.
3. The Aggregator pulls policies from the specified Policy Repository.
4. It fetches initial data sets from the designated PIPs.
5. The Aggregator configures the PDP with the retrieved policies and data.
6. When a PEP intercepts a request, it queries the PDP for an authorization decision. If the request is allowed and forwarded to the microservice, it may result in updates to microservice-managed data.
7. Resulting update events are sent to the event distribution system and received by the interested Aggregators.
8. Aggregator updates the PDP's data sets accordingly.

Similar setups have been successfully adopted in large-scale production environments. For example, Netflix presented a comparable design at KubeCon 2017[1]. Their terminology differs slightly: the component shown as *Aggregator* in the diagram above is called the "AuthZ Agent", and instead of letting each agent independently collect required data, Netflix introduced a central "Super PIP" (which they call the "Aggregator") positioned between the event distribution system and the agents. This component preprocesses and routes relevant data updates, while the AuthZ Agents remain responsible for configuring and updating the embedded PDP instances.

An open-source project that implements a similar architecture is OPAL - Open Policy Administration Layer[2], which allows managing OPA[3] and Cedar-Agent[4] instances. Compared to the diagram above, OPAL delegates responsibility for relaying data updates to the *Distributor*, which pushes relevant changes to each *Aggregator* instance.

Applicability beyond PBAC

Although both examples above use PBAC PDP engines, the architectural principles described here are not specific to PBAC engines. The control-plane components – configuration repository, distributor, and aggregator – as well as the mechanisms for policy and data provisioning, apply equally to other PDP types, such as ReBAC and NGAC. Unlike PBAC PDPs, which typically store policies and data only in memory, ReBAC and NGAC PDPs maintain persistent storage. In such deployments, the distributor is typically implemented as a CI/CD pipeline to handle automated provisioning and policy updates and does not manage runtime data updates.

This out-of-band data push approach introduces an important challenge: Consider a microservice (e.g., Service A) that updates its own database after a successful request and emits a corresponding domain event[5] intended to notify PDP-

[1] https://www.youtube.com/watch?v=R6tUNpRpdnY
[2] https://github.com/permitio/opal
[3] https://www.openpolicyagent.org/
[4] https://github.com/permitio/cedar-agent
[5] https://microservices.io/patterns/data/domain-event.html

related infrastructure (some of the interested Aggregators via an event bus). If the event is lost, delayed, or not processed correctly, the PDP's internal state may become outdated. As a result, future authorization decisions – possibly in other services – may be based on stale or incomplete data, leading to incorrect access grants or denials.

This situation reflects a classic **distributed transaction problem**: changes in the microservice and the state change in the PDP must eventually converge, but there's no atomic commit across both systems. Since traditional distributed transactions are often impractical or undesirable in such architectures, solutions may range from simple reliable event delivery mechanisms, like the Transactional Outbox[6] to more sophisticated patterns like Saga[7] if acknowledgement of event delivery is required.

6.3 Policy Input Data Governance

A note of caution

Neglecting proper governance of policy input data can break the entire authorization architecture: even small changes – renaming a field, changing a type, or omitting an attribute – can prevent policies from being evaluated correctly, and without governance, fixing such issues becomes like looking for a needle in a haystack.

As in the previous chapter, this chapter builds on the concepts introduced in Policy Input Data Distribution Strategies, but focuses on challenges common to all strategies that were not addressed earlier:

- **Structural and semantic consistency:** ensuring that supplied data matches the assumptions encoded in policies, such as the existence of user identifiers, values for resource ownership, etc., as a renaming of a field, change of type, or

[6] https://microservices.io/patterns/data/transactional-outbox.html
[7] https://microservices.io/patterns/data/saga.html

omission of an attribute may prevent policies from being evaluated correctly, creating the risk of incorrect authorization decisions.
- **Consumer visibility and coordination:** knowing which policies depend on which attributes so that producers can coordinate safely with policy owners before making schema or semantic changes.

These challenges are inherent to distributed architectures. Whether in Big Data pipelines spanning multiple data sources and transformations, or if multiple microservices are communicating to each other to execute some business function, in both domains, data or messages can break consumers if schemas or semantics change unexpectedly, and accountability for who relies on which data is unclear.

To address this, explicit agreements – often called **data contracts** in Big Data domain or **consumer driven contracts** in microservice architectures – codify the shared expectations between producers and consumers and define the "API of data" being exchanged. These types of contracts typically define schema, semantics, and quality guarantees, and also provide mechanisms for coordinated change management.

Adapting the same principles to authorization architectures bring similar benefits:

- **Communicating the Data API**: Contracts act as a shared reference between PIPs (data producers) and policy authors, clarifying which attributes are required and how they are structured.
- **Protecting Consumer Expectations**: Contracts can include domain constraints, value ranges, or other guarantees, helping policy assumptions remain valid as data evolves.

That would also ensure that data supplied to PDPs – whether pulled on-demand, pushed out-of-band, or passed inline – is complete, correctly typed, and semantically valid before reaching the PDP.

Standards such as the emerging Open Data Contracts Standard[8] provide structured ways of defining such contracts, while related tooling like the Data Contract

[8] https://bitol-io.github.io/open-data-contract-standard/v2.2.2/home/

CLI[9] supports validation and can also be used for enforcement. Alternatively, open-source governance platforms such as Apache Atlas[10] can be adapted to manage metadata, lineage, and schema evolution, or tools like Pact[11] can serve as a practical step toward implementing such contracts by codifying consumer expectations and validating producer behavior – all helping ensure that exchanged data meets structural and semantic requirements.

6.4 Interplay Between Authorization, Authentication, Identity Propagation Patterns, and Zero Trust

Authentication (who you are), authorization (what you're allowed to do), and identity propagation (how the results of authentication are securely carried forward) each address distinct concerns, yet they are deeply interconnected. The usage of one affects the requirements of the others, and vice versa. And only by aligning them consistently can identity, access, and trust be continuously verified and enforced across the system. As a natural consequence, the system as a whole comes to embody the principles of Zero Trust:

- **Never trust by default**: Treat every request as untrusted, even inside the same network perimeter.
- **Always verify everything**: Continuously and adaptively authenticate and authorize all requests, taking real-time signals, like user behavior or device state into account.
- **Least privilege**: Grant subjects, be it a user, device, or e.g., a service, only the permissions they need, minimizing attack surface.
- **Micro-segmentation**: Divide networks and systems into isolated micro zones to limit lateral movement.
- **Assume breach**: Operate as if attackers are already inside - monitor, log, and audit continuously.

[9] https://cli.datacontract.com/
[10] https://atlas.apache.org/
[11] https://pact.io/

- **Protect data**: Strongly encrypt sensitive information in transit and at rest to ensure confidentiality and integrity.

To achieve this alignment, it helps to determine the authorization approach first, then select identity-propagation mechanisms that can reliably convey attributes about the subject, and only then select the proper authentication patterns. This order prevents earlier decisions from imposing constraints that would undermine a secure, scalable, and maintainable system, while leaving room for deliberate deviations – for example, shifting enforcement closer to a service when downstream identity propagation makes it necessary.

Building on the outcome from the Authorization Patterns Recommendations, the application of this principle leads to the following recommendations:

- **Decentralized Service-Level Authorization**
  - When a service needs to communicate to downstream services, a stable, canonical representation of the external subject is required so that each hop can evaluate requests consistently. Applying the Protocol-Agnostic Identity Propagation pattern at the edge supports the required issuance of a signed subject structure – a special purpose *Authorization Contract* – which would travel with the request across the call chain.
  - However, this pattern alone does not address all limitations of Decentralized Service-Level Authorization: every downstream endpoint remains accessible to any authenticated subject. Additionally, endpoints intended to be public would now require authentication. Combining this pattern with Modern Edge-Level Authorization configured with a default-deny rule ensures that no endpoint is reachable unless explicitly permitted. Services that need to expose endpoints can now define allow rules: purely public endpoints can bypass authentication and the deny-all rule, while endpoints requiring authentication can disable only the deny-all rule.
  - Endpoints that do not consume the canonical contract (e.g., health checks, actuator APIs) require additional protection to prevent access from malicious peers within the same network. This protection can be provided through either the Kernel-Level Authentication pattern or the Service-Level

Proxy-Mediated Authentication pattern, both of which establish workload identity for every inbound connection.

- For services without downstream dependencies, identity propagation is unnecessary, but maintaining a default-deny posture is still recommended. This can be enforced through the same edge-level patterns or, alternatively, by using Sidecar-Level Authorization – a localized form of Modern Edge-Level Authorization – together with Service-Level Proxy-Mediated Authentication to validate the caller and determine its subject for internal processing.

- **Centralized Service-Level Authorization:** The same approach as for Decentralized Service-Level Authorization applies.

- **Modern Edge-Level Authorization:**

  - If a service needs to communicate with downstream services, a stable subject representation across hops is required. In this case, it is necessary to deviate from the result of the Authorization Patterns Recommendations and fall back to Centralized Service-Level Authorization. This deviation is valid because adding custom claims to, or changing the canonical subject entirely to build the *Authorization Contract* (the default behavior of Modern Edge-Level Authorization) would break downstream services that rely on it for their own authorization. Using a centralized service-level approach preserves consistency across the call chain while maintaining enforcement, though modern edge-level authorization mechanisms are still used, but limited to the bare minimum.
  - If no downstream calls are needed, the service can make use of Modern Edge-Level Authorization to its full extent, and Edge-Level Authentication is a natural fit to establish the external subject.
  - In either case, pairing with Kernel-Level Authentication or Service-Level Proxy-Mediated Authentication ensures that workload identity is established and inter-service communication – here between the edge and the service – is protected.

- **Classic Edge-Level Authorization:**

- Since no service-local data is used, this pattern naturally combines with Edge-Level Authentication and, as with other patterns, should be complemented with Kernel-Level Authentication or Service-Level Proxy-Mediated Authentication.
- If there is a need to communicate with downstream services, Protocol Agnostic Identity Propagation can be adopted as well.
- However, as discussed in the Classic Edge-Level Authorization chapter, this pattern has inherent socio-technical limitations. Therefore, Modern Edge-Level Authorization is recommended instead.

When authentication decisions limit identity propagation

While service-level code-mediated or proxy-mediated authentication are commonly used patterns to validate and establish the external subject, these approaches practically not only restrict secure identity propagation to token exchange-based only, which is actually designed to narrow the authorization scope of a requester in third-party contexts and is not intended for first-party use. They also tightly couple microservice code to OAuth 2.0/OIDC, making multi-principal subjects difficult to implement in practice, and entirely exclude multi-protocol scenarios.

6.5 Authentication, Identity Propagation, and Authorization Patterns in Practice

Building on the story of Alice and the blog platform and the example in Authorization Patterns Recommendations, this chapter illustrates how the recommended patterns can be implemented in practice.

6.5.1 Extended Access Requirements

- **Listing articles:** Every user may view a list of articles, including the title, publication date, author, and a short excerpt.

- **Reading articles:** Access to the full content depends on the user's subscription tier and the number of full articles already read that day. If the quota is exceeded or the user is anonymous, only an excerpt is shown. An exception applies for authors: an authenticated user may always read articles they wrote. Existing tiers are:
  - Free tier: up to 2 articles per day
  - Basic tier: up to 20 articles per day
  - Professional tier: unlimited
- **Writing articles:** Only professional-tier users may write. Before publication, an article must pass a harassment-content analysis. If rejected, the user is notified and warned. Warnings appear in the user's private profile.

6.5.2 Resulting Services

To support these requirements, the following services may be implemented:

- **Articles service** – manages article storage and retrieval, as well as the number of the read articles, with latter being cleared each night.
- **Subscription service** – tracks user subscription tiers.
- **Analysis service** – performs harassment analysis and stores warnings
- **Identity Provider** (**IdP**) – handles registration, login, password reset, etc.
- **Payment provider** – processes subscription fees.
- **Wiring application** – assembles the UI and orchestrates calls to the other services, using appropriate UI integration patterns.

6.5.3 Mapping Requirements to Patterns

- Listing articles Decentralized Service-Level Authorization
- Reading a full article Modern Edge-Level Authorization to create the *Authorization Contract*, enforced within the article service using Decentralized Service-Level Authorization
- Writing an article secure identity propagation from the edge through the article service to the analysis service via Protocol-Agnostic Identity Propagation + Centralized Service-Level Authorization

- Reading warnings Decentralized Service-Level Authorization
- Performing harassment analysis same pattern as above
- Wiring application UI checks only whether the user is authenticated; Decentralized Service-Level Authorization suffices
- In all cases Modern Edge-Level Authorization protects endpoints so they cannot accidentally become public.
- To secure service-to-service traffic (article analysis, article PDP, etc.) Kernel-Level Authentication is used to ensure workload identity.

6.5.4 Possible OSS Stack

Implementing Kernel-Level Authentication typically requires Kubernetes. Projects such as Cilium[12], Istio[13] (ambient mode), Linkerd[14], or other service-mesh implementations provide strong workload identity and mutual authentication for inter-service traffic.

For the IdP, social login via Google or Apple can cover registration and sign-in flows. To completely stay with the OSS stack, projects like Keycloak[15], Zitadel[16], Ory Kratos[17], or further can be used.

Modern Edge-Level Authorization and Protocol-Agnostic Identity Propagation can be implemented with the help of open-source projects such as Heimdall[18], Oathkeeper[19], Pomerium[20], or similar. In the walkthrough below I'll use heimdall simply because I maintain it, and it's the easiest way for me to illustrate the patterns. If Istio serves as the service mesh, Istio Gateway can act as the ingress, with heimdall integrated via Istio's `DestinationRule`.

[12] https://cilium.io/
[13] https://istio.io/
[14] https://linkerd.io/
[15] https://www.keycloak.org/
[16] https://zitadel.com/
[17] https://github.com/ory/kratos
[18] https://github.com/dadrus/heimdall
[19] https://github.com/ory/oathkeeper
[20] https://github.com/pomerium/pomerium

Because social login with Google requires an OIDC client functionality and heimdall (like many similar projects) does not implement it, an additional component is needed. oauth2-proxy[21] is a well-known option for that purpose.

As the PDP, OPA, with OPAL acting as the control-plane component to distribute policies and data to OPA instances, could be used. However, any other PDP and matching control-plane solution could be used in the same way.

To ensure that authorization decisions always reflect the most recent state, an event bus is required. Services publish relevant events, which are then consumed by OPAL and distributed to the PDP instances.

In this example:

- The **articles service** emits an event each time a user reads an article. The event contains the heimdall-issued JWT together with the user's updated "read articles" counter.
- The **subscription service** emits an event whenever a user changes their subscription tier.

This event-driven approach lets OPA react almost instantly to changes when evaluating policies. For reliable delivery, the event bus could be implemented with Apache Kafka[22], or lighter alternatives such as NATS[23], RabbitMQ[24], or similar.

6.5.5 Establishing a Canonical Subject and Enforcing a Deny-by-Default Posture

To establish a canonical subject and enforce a deny-by-default posture with heimdall one would define a so-called default rule:

```
default_rule:
  execute:
    # requires all requests "being authenticated" via google
```

[21] https://github.com/oauth2-proxy/oauth2-proxy
[22] https://kafka.apache.org/
[23] https://nats.io
[24] https://www.rabbitmq.com/

```
    - authenticator: google

    # denies all requests
    - authorizer: deny_all_requests

    # creates the canonical representation of the external subject
    - finalizer: jwt

  on_error:
    # triggers authentication flow if the above google authenticator
    # fails and the request was sent by a browser
    - error_handler: authenticate_with_google
      if: >
        type(Error) == authentication_error &&
        Request.Header("Accept").contains("text/html")
```

Each step in the two pipelines above (`execute` and `on_error`) references mechanisms from a predefined catalogue. This catalogue is part of heimdall's configuration and can be tailored to the needs of a particular system. If required, a step can also customize the behavior of the chosen mechanism, as shown in the next chapter. Other projects similar to heimdall, may require full configuration for every step, or may implement a similar catalogue-based approach.

6.5.6 Service-Specific Rules and Authorization Contracts

Each service can now define deviations as needed. For example, the wiring service would define a rule to expose public endpoints serving HTML and related content and another one to allow authenticated and anonymous requests to yet an additional endpoint:

```
apiVersion: heimdall.dadrus.github.com/v1alpha4
kind: RuleSet
metadata:
  name: "wiring app rules"
spec:
  rules:
    # allow authenticated or anonymous requests to the / route for GET requests
    - id: wiring-app:main-page
      match:
        routes:
          - path: /
        methods: [ GET ]
```

```
    execute:
      - authenticator: google
      - authenticator: anonymous
      - authorizer: allow_all_requests
      # jwt finalizer which creates the canonical representation of the
      # external subject and the error handler are reused from the default
      # rule

  # allow all GET requests to any css, js, or ico resources under / route
  - id: wiring-app:public-resources
    match:
      routes:
        - path: /:resources
          path_params:
            - name: resources
              type: glob
              value: "{*.css,*.js,*.ico}"
      methods: [ GET ]
    execute:
      - authenticator: anonymous
      - authorizer: allow_all_requests
      # jwt finalizer which creates the canonical representation of the
      # external subject and the error handler are reused from the default
      # rule
```

The code used to render the html page behind the `/` route can use any standard JOSE library and the public key from heimdall's `.well-known/jwks` endpoint to validate the issued JWT. This is a very simple application of the Decentralized Service-Level Authorization pattern. All services using Protocol-Agnostic Identity Propagation will see the same JWT structure and perform identical verification. And in case of the implementation to write articles, the articles service can simply pass the received JWT downstream to the analysis service along with the article to be verified.

Reading articles makes use of a wider range of Modern Edge-Level Authorization capabilities and establishes an own *Authorization Contract* by extending the JWT created by heimdall with some custom claims:

```
apiVersion: heimdall.dadrus.github.com/v1alpha4
kind: RuleSet
metadata:
  name: "articles service rules"
spec:
  rules:
```

```
- id: articles-service:read-article
  match:
    routes:
      - path: /articles/:article_id
    methods: [ GET ]
  execute:
    - authenticator: google
    - authenticator: anonymous
    - authorizer: allow_all_requests
    # since the actual enforcement is done in the implementation of the
    # articles service a contextualizer is used here instead of an
    # authorizer
    - contextualizer: opa
      config:
        values:
          policy: articles/allow
          action: read
          # the Subject object is created by the executed authenticator
          subject: "{{ .Subject.ID }}"
          # article_id captures the value from the request path defined in
          # the match expression above
          object: "{{ .Request.URL.Captures.article_id }}"
    # extend the JWT configured in the default rule with custom claims. A
    # complete rewrite is also possible instead.
    - finalizer: jwt
      config:
        values:
          requested_article: "{{ .Request.URL.Captures.article_id }}"
          allowed_representation: "{{ .Outputs.opa.result }}"

# other rules, e.g. for requests to write an article
```

With that in place the implementation of the read article functionality can make use of these custom claims after verifying the JWT received along the request without the need to call OPA directly.

The article write functionality verifies the JWT issued by heimdall as already described for the other services in this example, calls OPA to understand whether writing of articles is allowed and enforces it. If allowed, the corresponding UI representation is rendered to the user. When ready, the user submits the article, resulting in the same checks, followed by a call from the article service to the analysis service for the harassment analysis. Since that check can take a while, the user is redirected to some page explaining the progress. The corresponding rule for heimdall would look similar to the `wiring-app:main-page` shown at the

beginning of this chapter, but without a fallback to the anonymous authenticator.

7 Final Words

This concludes the primer, but the work of applying these patterns and principles is just the beginning. You now have a framework for making informed decisions about authorization, identity propagation, and authentication – a practical toolkit for real-world distributed systems.

Remember: there's no one-size-fits-all solution. Every system comes with unique requirements, constraints, and trade-offs. The patterns and recommendations shared here are meant to guide you, not dictate exact implementations. Treat them as a foundation on which to experiment, adapt, and refine your own approaches.

I hope this book inspires you to think critically about security architecture and empowers you to design systems that are not just functional, but secure, scalable, and maintainable.

8 About us

INNOQ is a technology consulting company. Honest consulting, innovative thinking, and a passion for software development means: We deliver successful software solutions, infrastructure and products.

We specialize in the following areas:

- Strategy and Technology Consulting
- Software Architecture and Development
- Data & AI
- IT Security
- Development of Digital Products
- Digital Platforms and Infrastructures
- Knowledge Transfer, Coaching and Trainings

With around 150 employees across offices in Germany and Switzerland, we support companies and organizations in designing and implementing complex initiatives and in improving existing software systems.

We are actively involved in open-source projects and the iSAQB® e.V., and we share our knowledge and experience at conferences and meetups, as well as through numerous books and professional publications.

Visit us at **www.innoq.com**

About the author

Dimitrij Drus

Dimitrij is a Senior Consultant at INNOQ with many years of experience in the architecture and security of distributed and embedded systems. His engagements in customer projects typically include supporting and mentoring teams in the practical implementation of the security and architectural concerns that matter, placing particular emphasis on helping development teams embrace security and removing barriers so that it becomes an integral part of good software quality rather than a perceived obstacle. Beyond client work, Dimitrij is active in the open-source community — including heimdall, an identity-aware proxy — and enjoys sharing his knowledge through training, blog posts, and conference talks.

Dimitrij Drus set out to rethink the OWASP Microservice Security Cheat Sheet – and shared his insights in a widely discussed blog series. Here's what others said about it:

"To all you out there interested in application security and microservices enthusiasts! If you are passionate about building secure distributed systems, I've got an interesting blog series for you.

Over the past couple of weeks I dived deep into a 7-part blog series by Dimitrij Drus on updating the OWASP Microservice Security Cheat Sheet. In my opinion, Dimitrij's work here is nothing short of brilliant. He is one of the most knowledgeable security experts I know and has taken tremendous effort to write up and reconsider a lot of the current state of microservice security.

Kicking off with sharp critique of the current sheet's gaps (shallow patterns, contradictions, outdated advice sometimes fueling risks like "accept-by-default"), he proposes to rethink it with the following key pillars in mind:

- NIST-based core concepts (subjects, policies, first/third-party contexts). Authentication patterns (service/edge/kernel-level, Zero Trust trade-offs).
- Identity propagation (methods and trade-offs)
- Authorization patterns (decentralized to modern edge, PDP options).
- Decision dimensions (policy/data traits, distribution, performance).
- Practical recs (e.g., blog platform with OPA/Heimdall/K8s), governance, and Zero Trust integration."

Jan Larwig, Maintainer of OAuth2 Proxy

This primer compiles the full series – clearly structured and refined. For software architects, developers, and anyone interested in practical approaches to securing distributed systems.

innoq.com