Master of Science in Computer Engineering

Master Degree Thesis

An in-depth comparative analysis of Kubernetes authorization mechanisms for fine-grained access control

Supervisors
prof. Riccardo Sisto
prof. Fulvio Valenza
dott. Daniele Bringhenti
dott. Francesco Pizzato

Candidate
Ines Muka

Academic Year 2024-2025

Summary

Cloud computing delivers on-demand computing resources over the Internet, driving modern infrastructure with unparalleled scalability, cost-efficiency, and adaptability. Moreover, Kubernetes has become the de facto standard for cloud orchestration tools for this paradigm, automating the deployment and administration of containerized applications across clusters. This dynamic, multi-tenant environment enables shared resources across various users, teams, and microservices. However, it poses security vulnerabilities if permissions are improperly managed. Consequently, granular access control is essential, ensuring tenant isolation while maintaining the operational advantages of Kubernetes.

The goal of this thesis is to present an in-depth comparative analysis of Kubernetes authorization mechanisms for fine-grained access control. First, an analysis of native and open source Kubernetes authorization mechanisms has been carried out. In particular, the considered mechanisms include Role-Based Access Control (RBAC), Attribute-Based Access Control (ABAC), and Webhook for native solutions, alongside Open Policy Agent (OPA) representing Policy-Based Access Control (PBAC) and SpiceDB representing Relationship-Based Access Control (ReBAC) for the open source alternatives. Then, for comparison, they have been assessed against multi-tenant scenarios that closely mirror real-world conditions. Finally, this work is concluded with a complete cross-evaluation of the complexity, granularity, scalability, and performance of each authorization mechanism and the paradigm it represents.

As a result, this work provides a practitioner's guide to selecting, implementing, and optimizing authorization mechanisms. In future research work, this analysis could be extended to integrate real-time threat response and automated validation tools.

Acknowledgements

I want to start by thanking my supervisors, prof. Riccardo Sisto, prof. Fulvio Valenza, dott. Francesco Pizzato and dott. Daniele Bringhenti, for giving me the opportunity to work on this thesis, which acted as a solid bridge between my ending academic life and my new professional career.

Special thanks go to my parents, who have selflessly sacrificed their whole lives to support my dreams. Ma, Ba, this is also your achievement. Thanks to Ani, my brother, my all-time favorite role model, and the reason why I set foot on this engineering journey in the first place. Thank you, Ro, for the always-present support and positivity all the way through.

And a big thank you to my person. Fra, thank you for being by my side and reminding me what this is all about. I could not have done this without you. And without the new family you brought me into.

Last but not least, I want to thank my old and new friends. The ones that supported me from different countries scattered around the world, and the ones we shared a life of fuorisede here in Torino. You made it all worth it.

Contents

List of Figures

Listings

Chapter 1

Introduction

Cloud computing has profoundly transformed the landscape of today's infrastructure by delivering on-demand computing resources over the Internet. This paradigm shift allows unparalleled scalability, cost efficiency, and flexibility, allowing enterprises to allocate resources dynamically, optimize operational costs, and accelerate innovation. At the core of this shift lies **Kubernetes**, which has emerged as the de facto standard for container orchestration [1]. Kubernetes simplifies and automates the deployment, scaling, and management of containerized applications across clusters, offering a robust framework for managing complex distributed workloads. The dynamic, multi-tenant environment enables resource sharing among multiple users, teams, and microservices, thereby maximizing operational agility and resource utilization.

However, the complexity and dynamism of Kubernetes pose considerable security challenges. In multi-tenant setups, improper management of permissions can lead to vulnerabilities, such as unauthorized access, data breaches, and compromised tenant isolation. Therefore, granular access control is not just an improvement but a critical requirement, as evidenced also by **OWASP**, which consistently lists broken access control and broken authorization as the primary risks in several of their top 10 reports [2] [3]. Such a control would ensure that each tenant functions within a secure perimeter while maintaining the operational benefits of Kubernetes. Given the complex nature of modern applications, along with evolving threat scenarios, advanced authorization systems are required that balance security with usability, allowing fine-grained control without impeding the agility that defines cloud-native ecosystems.

The main goal of this thesis is to deliver an in-depth comparative analysis of Kubernetes authorization mechanisms for fine-grained access control. To achieve this, the research systematically evaluates both native and open source solutions. For native mechanisms, we will consider **Role-Based Access Control (RBAC)**, **Attribute-Based Access Control (ABAC)**, and **Webhook**, while for open source alternatives, we are choosing **Open Policy Agent (OPA)** to represent the **Policy-Based Access Control (PBAC)** paradigm and **SpiceDB** to represent the **Relationship-Based Access Control (ReBAC)** paradigm. Each mechanism is examined in multi-tenant scenarios that closely mirror real-world conditions, revealing their practical applicability, strengths, and limits. The evaluation concludes with a comprehensive cross-comparison of all mechanisms and the paradigms they represent across four dimensions: complexity, granularity, scalability, and performance. The outcome is a practitioner's handbook that assists in the selection, implementation, and optimization of authorization methods tailored to specific organizational needs.

Methodologically, this work integrates theoretical analysis with empirical validation. The theoretical framework analyzes the architectural principles, policy paradigms, and integration workflows for each mechanism. In contrast, empirical validation uses scenario-based testing to assess practical performance, including tenant isolation, dynamic policy enforcement, and resilience under load conditions. This dual methodology guarantees that the findings are both conceptually sound and practically applicable.

This thesis is organized to provide a logical progression from fundamental concepts to a detailed analysis. **Chapter 2** introduces Kubernetes architecture, core components and objects, and the principles of access control, thereby laying the necessary technical groundwork. **Chapter 3** clarifies the thesis objectives, specifying the scope and methodology. **Chapter 4** analyzes native Kubernetes authorization mechanisms (RBAC, ABAC, and Webhook) through architectural blueprints, configuration workflows, and scenario-based assessments. **Chapter 5** extends this research to open source alternatives (OPA and SpiceDB), highlighting their enhanced functionalities and integration paradigms. **Chapter 6** offers a comparative evaluation within the four-dimensional matrix, consolidating findings into strategic recommendations. **Chapter 7** concludes with a summary of significant contributions and recommendations for future research directions. This structure provides practitioners and researchers with an extensive framework to navigate the intricacies of authorization in Kubernetes.

Chapter 2

Kubernetes

This chapter provides an overview of Kubernetes architecture, including its long history and evolution. It will serve as a summary outlining several topics that we will focus on in the later chapters. Kubernetes, often referred to as K8s, is a large framework that would require more time and effort for a comprehensive analysis, so only an overview of the fundamental concepts and elements is presented here. For further details, refer to the official documentation [4].

2.1 A bit of history

Before we move on to presenting the intricate details of the workings of Kubernetes, let us start by giving a historical perspective on how Kubernetes actually came to be. Around 2004, Google built the **Borg** [5], a relatively small initiative that initially involved fewer than five individuals. The initiative was developed alongside an updated version of Google's search engine. Borg was a large-scale management system for internal clusters that executed hundreds of thousands of jobs from several applications across multiple clusters, each including up to tens of thousands of machines.

Later, in 2013, Google launched **Omega** [6], a robust and scalable scheduler for large compute clusters. Omega offered a parallel scheduler architecture centered on shared state, utilizing lock-free optimistic concurrency control to attain both implementation adaptability and performance scalability.

Finally, in mid-2014, Google introduced **Kubernetes** as an open source variant of Borg. Kubernetes was created by Joe Beda, Brendan Burns, Craig McLuckie, and more engineers at Google. The creation and architecture of Kubernetes were significantly shaped by Borg, with numerous first contributors having prior experience with Borg. The initial Borg project was developed in C++, whereas Kubernetes was implemented in the Go programming language.

Kubernetes v1.0 was released in 2015. Simultaneously with the release, Google collaborated with the Linux Foundation in creating the **Cloud Native Computing Foundation (CNCF)** [7]. Kubernetes has greatly expanded since then, reaching CNCF graduated status and being adopted by nearly all big organizations.

Currently, we can say with certainty that Kubernetes serves as the **de facto** standard for container orchestration [1]

2.2 Workload management evolution

Kubernetes is defined as a portable, flexible, open source platform for managing containerized workloads and services, enabling declarative configuration and automation. It possesses a vast, rapidly expanding ecosystem, with its services, support, and tools easily accessible. But to understand the importance of such a definition, we should first provide some historical context of the evolution of workload management [8]

Traditional deployment era

Initially, enterprises hosted applications on physical servers. The inability to set resource limits for applications on a physical server resulted in resource allocation challenges. For instance, when several applications operate on a physical server, one application could take over the majority of resources, thus leading to poor performance of the other applications. A viable alternative would be to deploy each application on a separate physical server. However, this approach did not scale effectively due to the underutilization of resources, resulting in high maintenance costs for enterprises managing several physical servers.

Virtualized deployment era

Virtualization was implemented as a solution of the issues of the previous era. It enables the execution of multiple virtual machines (VMs) on a single physical server's central processing unit (CPU). Virtualization enables the isolation of applications into virtual machines (VMs), hence enhancing security by preventing unrestricted access to one application's data by another.

Virtualization improves resource efficiency on a physical server and promotes scalability by enabling the seamless addition or updating of applications, hence reducing hardware expenses and offering many other advantages. It also allows for the representation of a collection of physical resources as a cluster of ephemeral virtual machines, where each virtual machine operates as a complete system, executing all components, including its operating system, atop the virtualized hardware.

Container deployment era

Containers resemble virtual machines (VMs), although they consist of more relaxed isolation properties, allowing applications to share the operating system (OS). Consequently, containers are seen as lightweight. Like a virtual machine, a container owns its own filesystem, allocation of CPU, memory, process space, and additional resources. Being separated from the underlying infrastructure, they are portable across many cloud environments and operating system distributions.

Containers have gained popularity due to their additional advantages, including:

- Agile application development and deployment, enhancing simplicity and efficiency in container image creation relative to virtual machine image usage.
- Continuous development, integration, and deployment ensure dependable and frequent container image creation and deployment, allowing rapid and efficient rollbacks due to image immutability.
- Separation of concerns between development and operations, generating application container images throughout the build/release phase instead of during deployment, thus separating apps from infrastructure.
- Observability includes not only operating system-level information and metrics but also application health and various other indicators.
- Uniformity of the environment across development, testing, and production, thus operating identically on a laptop as it does in the cloud.
- Portability of cloud and operating system distributions, since it is compatible with Ubuntu, RHEL, CoreOS, on-premises environments, major public clouds, and additional platforms.
- Application-centric management, which elevates the abstraction level from an operating system on virtual hardware to executing an application on an operating system utilizing logical resources.
- Decoupled, distributed, elastic, and autonomous microservices, meaning applications are segmented into smaller independent components that can be deployed and controlled dynamically, rather than relying on a monolithic architecture operating on a singular specialized computer.
- Resource isolation with consistent application performance.
- Resource utilization with enhanced efficiency and density.

Figure 2.1 illustrates the evolution of workload management, highlighting the differences between various forms of application deployments.

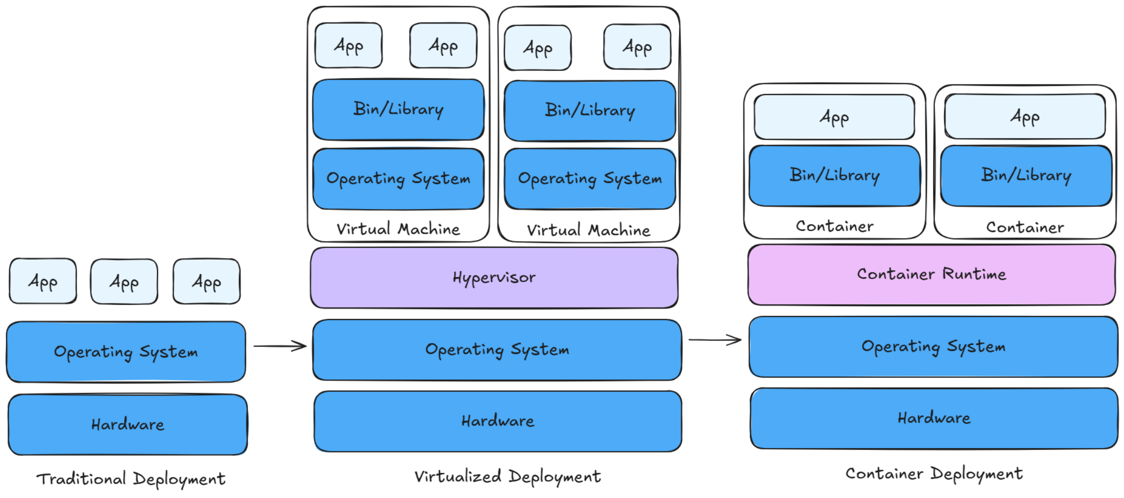

Figure 2.1. Evolution of applications deployments

Along with technological progress, there has been a shift in workload management techniques, transitioning from managing VMs as individual units to adopting a "cattle" paradigm, wherein VMs are regarded in a more generalized manner.

Even though their management remains closely linked to their lifecycle, the objective is to advance towards a decoupled methodology, like that utilized by Kubernetes: a declarative approach that states overall intentions, which the system then implements across all relevant resources, without requiring dealing with individual instances. This results in a more detached perspective in which resources are perceived as commodities that may be created, eliminated, and substituted as required.

2.3 Container orchestrators

Now that we have seen how workload management has evolved over time and what the present era of containerized deployment offers us, it brings us back to the core definition of Kubernetes and how it fits into the big picture.

As hundreds or thousands of containers get created, the necessity for effective management arises; therefore, container orchestrators have emerged. A container orchestrator is a system engineered to efficiently handle intricate containerized deployments across several machines from a centralized location. Currently, Kubernetes serves as the de facto standard for container orchestration, offering a framework for the resilient operation of distributed systems. It manages scaling and failover for our applications, offers deployment patterns, and provides additional functionalities.

Kubernetes offers [9]:

- **Service discovery and load balancing** Kubernetes may expose a container via its DNS name or its designated IP address. It can effectively load balance and distribute network traffic when there is considerable traffic to a container, ensuring deployment stability.
- **Storage orchestration** Kubernetes enables the automatic mounting of a preferred storage solution, including local storage and public cloud providers, among others.
- **Automated rollouts and rollbacks** Kubernetes allows you to articulate the desired state for your deployed containers, and it may adjust the actual state to align with the desired state at a regulated rate.
- **Automatic bin packing** Kubernetes is supplied with a cluster of nodes for executing containerized jobs. You specify the CPU and memory (RAM) requirements for each container, and it will optimally allocate containers to your nodes to maximize resource use.
- **Self-healing** Kubernetes restarts failing containers, replaces them, kills unresponsive containers based on user-defined health checks, and withholds advertisement to clients until they are prepared to serve.
- **Secret and configuration management** Kubernetes allows for storing and managing sensitive information, including passwords, OAuth tokens, and SSH keys. Secrets and application configurations can be deployed and updated without the necessity of rebuilding container images, hence preventing the exposure of secrets within your stack configuration.
- **Batch execution** Kubernetes can manage your batch and CI workloads, replacing failed containers if required, in addition to providing services.

- **Horizontal scaling** Kubernetes allows adjusting the scale of the application effortlessly with a command, via the user interface, or automatically in response to CPU utilization.
- **IPv4/IPv6 dual-stack** Allocation of IPv4 and IPv6 addresses to Pods and Services in a dual-stack configuration
- **Designed for extensibility** Integrate functionalities into your Kubernetes cluster without modifying the upstream source code.

2.4 Kubernetes architecture

Having looked at Kubernetes' role in the containerized deployment scene and its advantages as a container orchestrator, we can now explore its architecture in greater detail [10].

When we deploy Kubernetes, a **cluster** is set up. A cluster comprises **nodes** that execute containerized applications. At least one node handles the control plane and is referred to as **control plane** or **master**. Its function is to oversee the cluster and provide an interface for the user. Instead, the pods, which are components of the application, are hosted on the **worker node(s)**. The master oversees the worker nodes and the pods inside the cluster. There is also the case of **single-node** clusters, where the control-plane node also acts as a worker node. In production environments, the control plane typically operates over multiple machines, while a cluster generally spans several nodes, ensuring fault tolerance and high availability. Kubernetes adopts a modular architecture with various components that are responsible for different aspects of the cluster. The main ones are represented in Figure 2.2 and will now be briefly introduced.

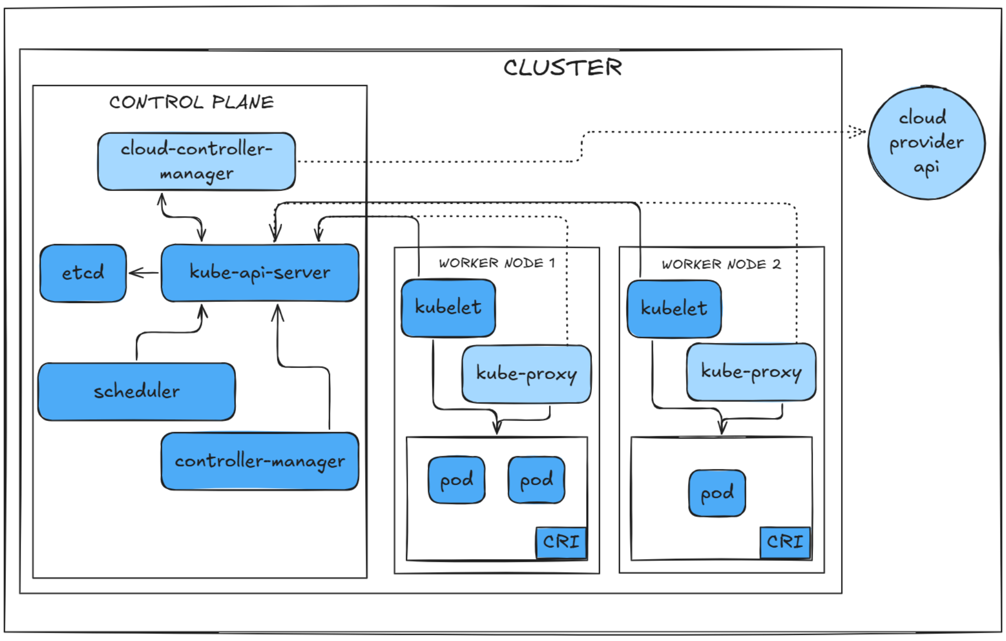

Figure 2.2. Kubernetes architecture

2.4.1 Control plane components

The components of the control plane take global decisions regarding the cluster, such as scheduling, and also identify and react to cluster events. Control plane components may be executed on any machine inside the cluster, but for the sake of simplicity, setup scripts often initiate all control plane components on a single system and do not execute user containers on that machine.

Some of the components of the control plane are as follows:

API server

The API server is probably the most important component in the Kubernetes control plane that interfaces with the Kubernetes API. It serves as the front end for the Kubernetes control plane. The default implementation of a Kubernetes API server is `kube-apiserver`. The `kube-apiserver` is engineered for horizontal scalability, meaning it expands by deploying additional instances. Multiple instances of kube-apiserver can be executed, allowing for traffic distribution among them.

etcd

`etcd` is a reliable and highly accessible key-value store, utilized as the backup store for all cluster data in Kubernetes. It keeps configuration data, state data, and metadata within Kubernetes. In order to guarantee data storage consistency among all cluster nodes for a fault-tolerant distributed system, etcd is based on the Raft consensus algorithm [11]. In a Kubernetes cluster, `etcd` is implemented as pods within the control plane. To enhance security and resilience, it may also be implemented as an external cluster.

Scheduler

It is a control plane component that monitors freshly produced pods lacking an assigned node and assigns a node to host them. The default scheduler offered by Kubernetes is known as `kube-scheduler`. However, it can be tailored by incorporating additional schedulers and specifying in the pods to use them. Scheduling decisions consider individual and collective resource requirements, hardware/software/policy limitations, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

kube-controller-manager

It is the component in charge of executing controller processes. It constantly compares the desirable state of the cluster, as defined by the object specifications, with the current state retrieved from `etcd`. Logically, every controller operates as an independent process, but in order to decrease complexity, they are all compiled into a singular binary and executed within a single process.

Several kinds of controllers exist. We can mention some of them here:

- **Node controller:** detects and addresses node failures.
- **Job controller:** monitors job objects that signify one-off tasks and subsequently generates pods to execute those tasks till completion.

- **EndpointSlice controller:** completes EndpointSlice objects to establish a connection between Services and Pods.
- **ServiceAccount controller:** generates default ServiceAccounts for newly created namespaces.

cloud-controller-manager

A component of the Kubernetes control plane that integrates cloud-specific control mechanisms. The cloud controller manager enables the connection of your cluster with your cloud provider's API, distinguishing the components that engage with the cloud platform from those that just engage with your cluster. The `cloud-controller-manager` exclusively operates controllers tailored to your cloud provider. If you are operating self-hosted Kubernetes or in a personal learning environment on your local machine, the cluster lacks a cloud controller manager.

Similar to the `kube-controller-manager`, the `cloud-controller-manager` merges multiple conceptually distinct control loops into a singular binary run as one single process. Horizontal scaling (executing many instances) enhances performance and increases fault tolerance.

The subsequent controllers may have dependencies on cloud providers:

- **Node controller:** verifies with the cloud provider whether a node has been removed from the cloud following its failure to respond.
- **Route controller:** configures routes within the base cloud infrastructure
- **Service controller:** creates, modifies, and deletes cloud provider load balancers.

2.4.2 Node components

Node components operate on each node, managing running pods and delivering the Kubernetes runtime environment.

Container runtime

An essential element that enables Kubernetes to run containers properly. It oversees the execution and lifecycle management of containers within the Kubernetes ecosystem. Kubernetes accommodates `container runtimes`, including containerd, CRI-O, and any other implementation of the Kubernetes Container Runtime Interface (CRI). Basically, we need a properly working `container runtime` on each node within our cluster so that the `kubelet` can launch Pods and their respective containers.

kubelet

An agent operates on every node within the cluster. It ensures that containers are operational within a pod. The `kubelet` receives a collection of PodSpecs via multiple ways and guarantees that the containers specified in those PodSpecs are operational and in good health. The `kubelet` is not in charge of containers that were not instantiated by Kubernetes.

kube-proxy (optional)

`kube-proxy` is a network proxy that operates on each node within your cluster, enabling a part of the Kubernetes Service concept. It regulates network configurations on nodes, where these network regulations provide communication to your Pods from network sessions both within and outside your cluster. It utilizes the operating system's packet filtering layer when it is present and accessible, or alternatively, `kube-proxy` directly routes the traffic by itself. Instead, if we are using a network plugin, then `kube-proxy` is unnecessary on the nodes within our cluster.

Addons

`Addons` utilize Kubernetes resources to implement cluster functionalities that are not yet native to Kubernetes. As these offer cluster-level functionalities, namespaced resources for addons must be under the `kube-system` namespace. Some examples are DNS, Web UI, network plugins, monitoring, and logging.

2.5 Kubernetes objects

After we have seen the architectural components of Kubernetes, we can move on to focus on the real building blocks that make up the ecosystem; Kubernetes objects are entities that remain persistent within the Kubernetes ecosystem [12]. Kubernetes leverages these entities to express the state of the cluster. They can specifically define:

- Which containerized applications are operational and on which nodes
- The resources accessible to those applications
- The policies governing the behavior of those applications, including restart rules, upgrades, and fault tolerance.

A Kubernetes object serves as a "record of intent". Upon its creation, the Kubernetes system persistently tries to sustain the object's existence. By constructing an object, you are explicitly defining the desired state of your cluster's workload within the Kubernetes system. To interact with Kubernetes objects, whether for creation, modification, or deletion, you must utilize the Kubernetes API. One practical way is to use the `kubectl` command-line interface to execute the requisite Kubernetes API calls on your behalf.

Nearly every Kubernetes object comprises two nested fields that dictate the object's configuration: `spec` and `status`. For objects with a `spec`, it is necessary to establish this at object creation by detailing the qualities you wish the resource to possess, describing its desired state. Instead, `status` defines the present state of the object, provided and refreshed by the Kubernetes system and its components. The Kubernetes control plane consistently and proactively regulates the actual state of each object to align with the desired state you provided.

Kubernetes establishes various object types that serve as its basic building blocks. A Kubernetes resource object typically comprises the following fields:

- `APIVersion`: denotes the versioned schema of this object's representation.
- `Kind`: a string value denoting the REST resource this object signifies

- `ObjectMeta`: metadata pertaining to the object, including its name, annotations, labels, and other relevant details.
- `ResourceSpec`: user-defined, it delineates the intended state of the item.
- `ResourceStatus`: populated by the server, it indicates the current condition of the resource.

The permitted operations on these resources consist of the standard CRUD actions:

- **Create:** `create` generates the resource within the storage backend. Upon the creation of a resource, the system enforces the specified state.
- **Read:** includes three variations.
  - `get`: obtain a specific resource object by its name.
  - `list`: obtain all resource objects of a particular type inside a namespace, with the option to limit results to those that meet a selector query.
  - `watch`: stream updates for an object(s) as modifications occur.
- **Update:** includes two variations.
  - `replace`: substitute the current specification with the supplied one.
  - `patch`: implement a modification to a designated field.
- **Delete:** `delete` removes a resource. Child objects of the deleted resource may or may not be subject to garbage collection by the server, depending upon the specific resource.

We will also outline the most important objects needed for the chapters that follow.

Labels and Selectors

Although not classified as objects, these elements are essential for object description, so we address them first. Labels are key-value pairs applied to a Kubernetes object, utilized for the organization and identification of a subset of objects. Selectors are the grouping constructs that enable the selection of a collection of objects sharing the same label.

Namespaces

Namespaces serve as virtual partitions inside the cluster. Kubernetes, by default, generates four namespaces:

- `kube-system`: comprises items generated by the Kubernetes system, mostly control-plane agents.
- `default`: comprises items and resources generated by users and is the one entity used by default
- `kube-public`: accessible to all users, including unauthenticated individuals, and utilized for specific functions such as sharing public information about the cluster.
- `kube-node-lease`: responsible for preserving items related to heartbeat data from nodes.

Dividing the cluster into many namespaces is an effective strategy for enhancing its virtualization.

Pods

Pods represent the fundamental processing units within Kubernetes. A pod is a logical aggregation of one or more containers that utilize the same network and storage and are concurrently scheduled within the same pod. Pods are ephemeral and lack auto-repair capabilities; hence, they are typically governed by a controller that manages replication, fault tolerance, and self-healing.

ReplicaSet

ReplicaSets manage a collection of pods, enabling the scaling of the current pod count implementation. Upon the deletion of a pod within the set, the ReplicaSet detects an inconsistency between the current number of replicas, as indicated in the Status, and the desired count, as stated in the Spec, prompting the creation of a new pod. Typically, ReplicaSets are not utilized directly. Instead, Kubernetes offers a higher-level abstraction known as a Deployment that we are considering next.

Deployment

Deployments oversee the creation, alteration, and deletion of pods. A deployment automatically generates a ReplicaSet, which then produces the specified number of pods. Consequently, an application is generally deployed within a deployment rather than in an individual pod.

Service

A service is an abstract mechanism for exposing an application operating on a collection of Pods as a network service. It may have varying access scopes based on its ServiceType:

- `ClusterIP`: A service accessible solely from within the cluster. It is the default type.
- `NodePort`: Exposes the service on a fixed port of each Node's IP address. The NodePort Service is accessible externally by reaching `<NodeIP>:<NodePort>`.
- `LoadBalancer`: Exposes the service externally using a cloud provider's load balancer.
- `ExternalName`: Associates the service with an external entity, enabling local applications to use it.

ConfigMaps

ConfigMaps are Kubernetes objects utilized for the storage of container configuration data in key-value pairs. ConfigMaps facilitate the development of lighter and more portable images by isolating configuration data from the remainder of the container image.

Secrets

A secret is an object that holds a limited quantity of sensitive information, including a password, token, or key. This information may alternatively be included in a pod specification or within a container image. But utilizing a secret eliminates the necessity of embedding sensitive information within your application code. This way, as secrets can be generated separately from the pods that utilize them, the risk of exposing the secret and its data throughout the processes of creation, viewing, and modifying Pods is minimized. In some ways, secrets resemble ConfigMaps but are specifically designed to store sensitive information.

ServiceAccount

A service account is a non-human account that offers a unique identity within a Kubernetes cluster. Application pods, system components, and entities within as well as outside the cluster can utilize the credentials of a designated ServiceAccount to authenticate as that ServiceAccount. This identity is beneficial for several contexts, such as authenticating with the API server or enforcing identity-based security regulations. They are namespaced, lightweight, and portable. Upon the creation of a cluster, Kubernetes automatically generates a ServiceAccount object designated as default for each namespace within the cluster. When a pod is deployed in a namespace without a manually assigned ServiceAccount, Kubernetes automatically allocates the default ServiceAccount of that namespace to the pod.

At this point, we have covered the fundamentals of Kubernetes, and we are well prepared to dive deep into the details of access control and authorization that are going to be our main focus throughout this thesis.

2.6 Kubernetes access control

Since our work is going to be focused on the authorization mechanisms in Kubernetes, we see it as necessary to first introduce it from the perspective of access control fundamentals [13], and then we will move on to focusing more on the authorization implementation.

Both human users and Kubernetes service accounts may be granted authorization for API access. Upon receiving a request, the API undergoes multiple phases: **authentication**, **authorization**, and **admission control**, before granting access, as demonstrated in Figure 2.3.

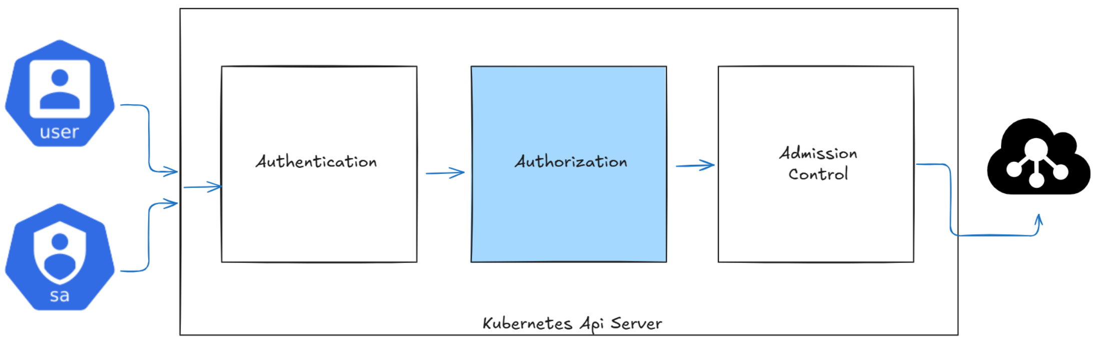

Figure 2.3. Access control in Kubernetes

We start with a short introduction of the authentication mechanism [14]. Authentication is the procedure by which one's identity is verified using certain credentials. In Kubernetes, after the establishment of TLS for secure communication, the HTTP request proceeds to the authentication phase. The API server is configured to execute one or more authenticator modules, specified in the cluster startup or the following configuration. They consist of client certificates, passwords, plain tokens, bootstrap tokens, and JSON Web Tokens (for service accounts). Several authentication modules may be defined, where each is attempted sequentially until one succeeds. Requests that cannot be authenticated are denied with an HTTP status code of 401. Otherwise, the person is authorized under a specific username, which is exposed for the following steps.

Next in line is the authorization mechanism [15]. Once the request is verified as coming from a specific user, it must undergo authorization. A request has to include the requester's username, the desired action, and the object impacted by the action. The request is approved if a current policy indicates that the user possesses permission to carry out the action requested. Kubernetes authorization demands the utilization of standard REST attributes in order to engage with pre-existing organization-wide or cloud-provider-wide access control systems. Using REST formatting is essential also because these control systems may interface with APIs beyond the Kubernetes API. Kubernetes supports many authorization modules, which we are going to specify in more detail later on. If several of these are specified upon cluster startup or later configuration, Kubernetes evaluates each module sequentially, and if any of them grants authorization, the request may then proceed. If all modules reject the request, the request gets denied with an HTTP status code of 403.

The last mechanism in line is called Admission Control [16]. These modules are software components that can alter or deny requests. Besides the attributes visible to authorization modules, admission control modules are allowed to access the object's contents that are being created or updated. Admission controllers respond to requests that create, modify, delete, or proxy an object. They do not respond to requests that solely read objects. When multiple admission controllers are configured, they are invoked sequentially. In contrast to authentication and authorization modules, a rejection by any admission controller module results in the instant denial of the request. Upon being successfully processed through all admission controllers, a request undergoes validation using the designated procedures for the respective API object, and after that, gets written in the object store.

Now that we have an idea of how the authorization mechanism is positioned in the access control process in Kubernetes, we can move on to providing more details on how this mechanism is implemented. Starting with the authorization modes supported by the Kubernetes API server:

AlwaysAllow

As the name suggests, this mode permits all requests, bypassing authorization. Since authorization mechanisms generally give either a denial or a no-opinion outcome, activating the AlwaysAllow mode will permit all the requests after all other authorizers provide a "no opinion" response. The AlwaysAllow mode should not be utilized on a Kubernetes cluster accessible from the Internet or in environments where there is no full trust in all possible API clients, including the workloads deployed. The sole acceptable context is maybe within a testing scenario.

AlwaysDeny

As the name suggests, this mode denies all requests. The sole acceptable context is maybe within a testing scenario.

Node

It is a specialized authorization mode that only authorizes kubelet-generated API requests.

RBAC

Role-based access control is a mechanism for controlling access to computer or network resources based on the roles of individual users within an organization. We are going to see this authorization mode in full detail later.

ABAC

Attribute-based access control establishes a paradigm of access control in which access permissions are given to users via policies that bring together different attributes. We are going to see this authorization mode in full detail later.

Webhook

This mode does a synchronous HTTP callout, stalling the request until the third-party HTTP service replies to the query. We are going to see this authorization mode in full detail later.

One note at this point about the `system:masters` group, which is a predefined Kubernetes group that provides unrestricted access to the API server. Users appointed to this group possess complete cluster administrator rights, bypassing any boundary of authorization established by the RBAC or Webhook mode. Therefore, it is advised not to add users to this group. If you indeed have a need to allow cluster-admin rights to a user, you may do so by making a ClusterRoleBinding to the built-in cluster-admin ClusterRole.

We believe this covers, for now, the broad view of Kubernetes architecture, objects, and the basis of access control and authorization. We will go into detail about the different native authorization mechanisms and the open source solutions in Kubernetes in the following chapters.

Chapter 3

Thesis objectives

Having established our motivation and general goal for this thesis, as well as presented the principles of Kubernetes to lay out our groundwork, we will now outline our thesis objectives in more detail, describing how we will carry out this research and what the expected contributions are.

The evolution of cloud computing and container orchestration has elevated Kubernetes to a pivotal role in modern infrastructure. However, the security implications of its multi-tenant architecture require robust authorization mechanisms capable of enforcing fine-grained access control. While Kubernetes offers native solutions and the ecosystem provides open source alternatives, a systematic comparison of their efficacy, scalability, and operational viability remains underexplored. This thesis addresses this gap by conducting an exhaustive comparison of Kubernetes authorization mechanisms, including both native and open source alternatives, going beyond theoretical explanations and into assessing mechanisms in scenarios that replicate real-world multi-tenant needs to finally evaluate them systematically.

This thesis employs a three-phase methodology grounded in theoretical rigor, empirical validation, and comparative synthesis. The initial phase establishes a thorough theoretical foundation by analyzing the architecture and fundamental principles of each authorization method and the paradigm they represent. This involves analyzing the implementation mechanics for each of the chosen authorization mechanisms, be they native to Kubernetes or open source alternatives. So we will be exploring the general workings of the mechanism alongside more specific aspects. Like, for example, for Role-Based Access Control (RBAC), this will include role-binding semantics, privilege escalation safeguards, and role aggregation while emphasizing its inability to enforce deny rules or context-aware policies. Moving on to Attribute-Based Access Control (ABAC), we will examine it with a focus on attribute-policy mapping procedures and the operational limitations of static file management, while specifically addressing the difficulties in policy conflict resolution and version control. The assessment of Webhooks covers HTTP callback workflows and security prerequisites, highlighting their dependency on the availability of external services and network resilience. Moving on to open source alternatives, we will focus specifically on those that offer a paradigm extension. Open Policy Agent (OPA) is analyzed for its Policy-As-Code paradigm utilizing Rego, facilitating dynamic context-aware decision-making, while SpiceDB's graph-based

Relationship-Based Access Control (ReBAC) paradigm is evaluated for its efficient navigation of resource hierarchies and inherited permissions. This phase clarifies how each mechanism interprets access requests, processes policy logic, and integrates with Kubernetes' API server, revealing underlying strengths or limitations and how they impact the granular access control.

The second phase shifts to empirical validation using scenario-based practical analysis. These meticulously designed scenarios try to replicate real-world challenges across isolated Kubernetes clusters, each configured with a single authorization mechanism. So we will encounter namespace isolation tests for RBAC's role-binding efficacy, policy order tests for seeing its impact on ABAC's correctness, label-matching testing policy for Webhook's flexibility, time-based testing policy for OPA's Rego capabilities, attribute-based capabilities in the relationship schema of SpiceDB to test the fusion of ReBAC with ABAC, and many more. This empirical methodology connects theoretical insights with practical applications, making it easier to evaluate not only their individual strengths and limitations, but also useful for the subsequent cross-comparison among all solutions. To maintain methodological clarity, these theoretical and empirical investigations are conducted in parallel for each mechanism.

Lastly, the third phase conducts a cross-comparative evaluation using a four-dimensional matrix, composed of complexity, granularity, scalability, and performance, to assess trade-offs. Complexity is determined via setup procedures, tooling dependencies, and development efforts. Granularity is evaluated through policy expressiveness, including aspects like support for context-aware decisions, label selectors, and explicit denial rules. Scalability is assessed in scenarios of role/policy explosion and the capabilities of the architecture to support a large number of resources, users, or requests. Performance is evaluated by decision-making speed and overhead or network-induced latencies based on the architectural characteristics. Mechanisms are ranked dimensionally, specifying them as the most and least optimal choices. The concluding summary converts these findings into a practitioner's guide, aligning organizational needs with solutions and trying to give recommendations on how to choose the most optimal one.

After having explained our methodology in detail, we can state the thesis contributions more clearly. Three fundamental contributions are made to Kubernetes security, particularly regarding fine-grained access control. First, it establishes a comprehensive taxonomy of authorization paradigms, systematically categorizing native mechanisms, clarifying architectural principles, policy semantics, and integration workflows. Second, it delivers an extensive empirical validation utilizing scenario-based analysis. The findings from this are further explored through a cross-comparison analysis among all present mechanisms, based on a four-dimensional evaluation matrix spanning complexity, granularity, scalability, and performance. Last, the work synthesizes these theoretical and empirical insights into a practitioner's handbook, enabling informed selection, practical implementation, and optimization reasoning for the authorization mechanisms, enabling practitioners to manage Kubernetes' security challenges while maintaining innovation speed.

What remains for us is to deep-dive into our analysis and achieve our defined objectives.

Chapter 4

Native k8s authorization mechanisms

In this chapter, we are going to provide a deep dive into native Kubernetes authorization mechanisms. We covered the high-level description of access control in Kubernetes and the authorization mechanisms supported natively by it in the Kubernetes chapter. Therefore, we can go into more details on the main triplet of these mechanisms, RBAC, ABAC, and Webhook, not just providing a high-level view introducing their workings and characteristics, but proceeding with a scenario-based analysis and results drawn from it. All the high-level descriptions of the authorization mechanisms are true to the official documentation of Kubernetes [17].

4.1 Role-Based Access Control (RBAC)

Role-based access control is one of the authorization mechanisms present in the group of native authorization mechanisms in Kubernetes. As the name suggests, RBAC is a mechanism for controlling access to computer or network resources based on the roles of individual users within an organization.

RBAC authorization uses the `rbac.authorization.k8s.io` API group to facilitate authorization decisions, enabling dynamic policy configuration using the Kubernetes API. The RBAC API defines four types of Kubernetes objects: **Role**, **ClusterRole**, **RoleBinding**, and **ClusterRoleBinding**. RBAC objects can be described or modified using command-line tools like `kubectl`, similar to any other Kubernetes object.

An RBAC Role or ClusterRole comprises rules that define a collection of permissions. Permissions are only additive, with no space for ”deny” rules. The difference between the two stands on the fact that a Role specifies permissions inside a specific namespace, which you are constrained to pick upon creation, whereas a ClusterRole is not namespace-scoped. Consequently, ClusterRoles serve multiple purposes. They can be utilized to:

- Establish permissions on namespaced resources and allow access inside specific namespace(s).

- Establish permissions on namespaced resources and allow access across all namespaces.
- Establish permissions for resources that are scoped to the cluster.

So the choice is clear: if you need to define a position within a namespace, utilize a Role. Instead, if a role across the cluster is needed, ClusterRole is the correct object.

As we mentioned before, the **set of permissions** is defined through **rules**. In the rule, we specify the allowed apiGroups, their **resources** (or non-resources), and **verbs** that can be applied to these resources. In the Kubernetes API, the majority of resources are represented and accessible via a string denoting their object name, such as `pods` for a Pod. RBAC denotes resources utilizing the identical naming found in the URL of the corresponding API endpoint. Certain Kubernetes APIs include a subresource, like the logs of a Pod that in RBAC role is represented as `pods/log` using a slash to differentiate the resource from the subresource. For specific requests, you may also reference resources by their names using the `resourceNames` list. Upon specification, requests may be confined to particular instances of a resource. But you cannot limit `create` or `delete collection` requests based on their resource name. This constraint arises because the name of the new item is unknown at the time of authorization. To restrict listing or watching by `resourceName`, clients have to include a `metadata.name` field selection in their request that aligns with the designated `resourceName` to obtain authorization.

Instead of specifying individual resources, apiGroups, and verbs, one may utilize the wildcard (represented with `*`) to denote all such entities. For `nonResourceURLs`, the wildcard can be used as a suffix glob match. But the utilization of wildcards in resource and verb entries may lead to excessively liberal access to critical resources. The principle of least privilege must be implemented, utilizing specified resources and actions to guarantee that only the necessary rights for the workload's proper operation are granted.

Simple samples of Role and ClusterRole manifests are as follows:

Listing 4.1. RBAC Role object sample manifest

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: secret-reader
  namespace: namespace-a
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "list", "watch"]
```

Listing 4.2. RBAC ClusterRole object sample manifest

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cross-namespace-secret-reader
```

```
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "list", "watch"]
```

A **role binding** applies the permissions specified in a role to an individual or group of users. It contains a list of **subjects** (users, groups, or service accounts) and a reference to the role being bound to. A RoleBinding allows permissions within a designated namespace, but a ClusterRoleBinding extends that access throughout the entire cluster. A RoleBinding can reference any Role inside the same namespace or, alternatively, can reference a ClusterRole, in this way binding that ClusterRole to the namespace of the RoleBinding. Instead, to associate a ClusterRole with all namespaces in your cluster, utilize a ClusterRoleBinding.

To elaborate a bit more on the subjects, as stated before, they may include groups, users, or ServiceAccounts. Kubernetes denotes usernames as strings. It is the responsibility of the cluster administrator to set up the authentication modules to generate usernames in your desired format. Be cautious about the prefix `system:`, as it is designated for Kubernetes system use. Ensure that no individuals or groups accidentally possess names that begin with `system:`. Aside from this specific prefix, the RBAC authorization mechanism imposes no format requirements for usernames.

In Kubernetes, authenticator modules supply group information. Groups, similar to users, are denoted as strings, which have no requirement of format rules, except that the prefix `system:` is designated for reserved usage. ServiceAccounts are designated with names that begin with `system:serviceaccount:`, and are associated with groups that have names starting with `system:serviceaccounts:`.

Simple samples of RoleBinding and ClusterRoleBinding manifests are as follows:

Listing 4.3. RBAC RoleBinding object sample manifest

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: user-a-secret-reader
  namespace: namespace-a
subjects:
- kind: User
  name: user-in-A
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: secret-reader-a
  apiGroup: rbac.authorization.k8s.io
```

Listing 4.4. RBAC ClusterRoleBinding object sample manifest

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
```

```
metadata:
  name: admin-cluster-secret-reader
subjects:
- kind: User
  name: cluster-admin-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cross-namespace-secret-reader
  apiGroup: rbac.authorization.k8s.io
```

To put it in a visualized manner, Figure 4.1 shows how everything is interconnected in RBAC.

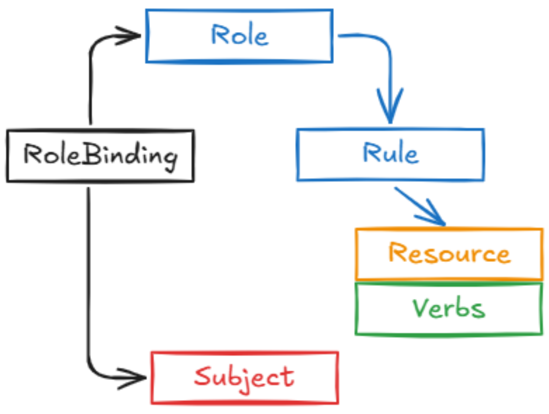

Figure 4.1. RBAC model

RBAC API in Kubernetes offers some special solutions that other native mechanisms do not support.

First, it supports the **aggregation of ClusterRoles**. Multiple ClusterRoles can be consolidated into a single, unified ClusterRole. A controller, operating within the cluster control plane, monitors ClusterRole objects that possess an `aggregationRule` configuration. The `aggregationRule` specifies a label selector utilized by the controller to identify other ClusterRole objects that should be combined into the rules field of this particular object. Since the control plane replaces any settings you explicitly input in the rules field of an aggregate ClusterRole, and then to modify or append rules, adjust the ClusterRole objects identified by the `aggregationRule`. ClusterRole aggregation. It is used for the default user-facing roles, enabling the cluster administrator to add rules for custom resources, like those provided by CustomResourceDefinitions (CRD) or aggregated API servers, therefore expanding the default roles.

Next, it includes the newly mentioned **default user-facing roles** as part of the default roles and role bindings. API servers provide a collection of default ClusterRole and ClusterRoleBinding objects. Many of these are prefixed with `system:` indicating that the resource is directly governed by the cluster control plane. Every default ClusterRole and ClusterRoleBinding is annotated with `kubernetes.io/bootstrapping=rbac-defaults`. This way it can support **auto-reconciliation**, API discovery roles, user-facing roles, core component roles, and other component roles. With auto-reconciliation, upon each start-up, the API server updates default cluster roles by including any absent permissions and updates default cluster role bindings by adding any missing subjects. This enables the cluster to fix unintentional modifications and helps in keeping the roles and role bindings updated when permissions and subjects evolve in future Kubernetes releases. As for **API discovery roles**, default cluster role bindings provide both unauthenticated and authenticated users access to API information considered safe for public disclosure, including CustomResourceDefinitions. We can disable anonymous unauthenticated access by incorporating the `--anonymous-auth=false` flag into the API server settings. As for user-facing roles, we have to know that there exist some default ClusterRoles without the prefix `system:` intended as user-facing roles. They contain super-user roles (`cluster-admin`), roles designated for cluster-wide assignment via ClusterRoleBindings, and roles designed for specific namespaces through RoleBindings (`admin`, `edit`, and `view`). User-facing ClusterRoles utilize ClusterRole aggregation to enable administrators to add rules for custom resources within these ClusterRoles. The same thing is applied to core component roles and other cluster roles.

Last, RBAC API supports **privilege escalation prevention** and **bootstrapping**. This enforcement happens at the API level; therefore, it is applicable even in the absence of the RBAC authorizer. There are clear restrictions on role creation and updating, and role binding creation. A role can only be created or updated if at least one of the following conditions is met:

- You possess all the permissions associated with the role, applicable at the same scope as the object being altered (cluster-wide for a ClusterRole, or within the same namespace or cluster-wide for a Role).
- You are explicitly authorized to execute the escalate verb on the roles or clusterroles resource within the `rbac.authorization.k8s.io` API group.

A role binding can only be created or updated if at least one of the following conditions is met:

- You possess all the permissions associated with the role, applicable at the same scope as the object being altered (cluster-wide for a ClusterRoleBinding, or within the same namespace or cluster-wide for a RoleBinding).
- You are explicitly authorized to execute the bind verb on the Roles or ClusterRoles resource within the `rbac.authorization.k8s.io` API group.

The last thing to mention regarding RBAC is how it can be an upgrade from ABAC. One of the differences between these two mechanisms that makes the transition difficult is that ABAC, with its permissive policies, allows full API access for all the service accounts, while RBAC, even though it gives scoped access to control-plane components, nodes, and controllers, denies permissions to service accounts outside of the `kube-system` namespace. Although significantly more secure, this may disturb existing workloads that anticipate automatic API authorization. This can be fixed in two ways: either by running both authorizers in parallel or replicating the permissive policies of ABAC by using RBAC role bindings.

We have now concluded by providing a high-level overview of the workings of RBAC in Kubernetes. We have familiarized ourselves with the way it works in principle and are prepared to advance to the next phase of exploring RBAC in action. Prior to engaging in a scenario-based analysis where we will explore the practical implementation of RBAC, we are going to examine further specifics regarding its configuration for use within the cluster.

4.1.1 Configuration

RBAC authorization mode is certainly the easiest to configure in our cloud-agnostic Kubernetes scenario. Since the Kubernetes API server comes by default with RBAC configured as one of the authorization modes under the `--authorization-mode` flag, we do not need to carry out extra configuration in cluster startup. This is true for cluster setups with **kubeadm**, as well as **kind** clusters. Actually, under this category also falls the Node authorization mode. And it is important to note that these two authorization modes, RBAC and Node, cannot be removed or added in a later configuration, only reordered. This is a very crucial point, because in the following ABAC and Webhook authorization modes, we are going to deal with a significantly complex configuration at cluster startup. Therefore, all we have to do to use RBAC authorization mode for our following scenario-based analysis is set up the cluster itself.

In kubeadm, we can set up our cluster by carrying out the following commands in a terminal [18]:

```
#intiate cluster
sudo kubeadm init --pod-network-cidr=a.b.c.d/e --service-cidr=f.h.i.j/k
#configure kubectl
mkdir -p $HOME/.kube
sudo cp /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
#install a CNI
curl -L --remote-name-all https://github.com/cilium/cilium-cli/releases/\
download/v0.18.3/cilium-linux-amd64.tar.gz
sudo tar xzvfC cilium-linux-amd64.tar.gz /usr/local/bin
cilium install

#run this command in the worker node to join it to your cluster
kubeadm join control-plane-ip:port --token itValue \
--discovery-token-ca-cert-hash itValue
#or for a one-node cluster
#run this to be able to schedule workloads in control plane node
kubectl taint node --all node-role.kubernetes.io/master:NoSchedule-
```

In kind, we can simply set up our cluster with the following command [19]:

```
#create one-node cluster
kind create cluster
#create a multi-node cluster
kind create cluster --config kind-config-file.yaml
```

Where the configuration file passed in the command looks like this:

Listing 4.5. Kind configuration file

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
```

The only extra configuration in this case has to do with the authentication mechanism instead. For our RBAC scenarios, we have chosen to authenticate our users via the means of **X.509 certificates**. We can carry out the following commands to prepare our environment for our scenario-based analysis that is next on our agenda [20]:

```
#generate private key:
openssl genrsa -out user.key 2048
#create CSR:
openssl req -new -key user.key -out user.csr -subj \
"/CN=user/O=which-namespace"
#encode CSR in base64
cat user.csr | base64 | tr -d "\n"
#make sure to copy this to insert in the CertificateSigningRequest
#create a CertificateSigningRequest Kubernetes object
#(file structure shown later)
kubectl create -f certificate-signing-request.yaml
#approve the certificate
kubectl certificate approve user-csr
#you can check CSRs (should show approved and issued state)
kubectl get csr
#retrieve and store the approved certificate
kubectl get csr user-csr -o jsonpath='{.status.certificate}' | base64 -d \
> user.crt
#finally set user credentials
kubectl config set-credentials user --client-certificate=user.crt \
--client-key=user.key
```

And as promised, the file structure for the CSR object manifest looks like the below.

Listing 4.6. CertificateSigningRequest object manifest

```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
    name: user-csr
spec:
    groups:
    - system:authenticated
    request: <base64-encoded-csr>
    signerName: kubernetes.io/kube-apiserver-client
    usages:
    - client auth
```

Last, we are going to quickly review some command-line utilities to manage RBAC objects. Even though in our scenarios we are going to deploy these types of objects by applying manifests, for the sake of completeness, we are going to show here also the individual commands that make it possible to dynamically configure and manage these objects and, as a consequence, RBAC accsess control.

```
#create a Role allowing to perform these verbs on these resources
kubectl create role name-of-role --verb=verb1,verb2 \
--resource=resource1,resource2
#create a ClusterRole
kubectl create clusterrole name-of-clusterrole --verb=verb1,verb2 \
--resource=resource1,resource2
#create a RoleBinding between a Role and user
kubectl create rolebinding user-role-binding --role=name-of-role \
--user=name-of-user --namespace=namespace-of-role
#create a RoleBinding between a ClusterRole and user
kubectl create rolebinding user-clusterrole-binding \
--clusterrole=name-of-clusterrole --user=name-of-user \
--namespace=which-namespace
#create a ClusterRoleBinding between a ClusterRole and a service account
kubectl create clusterrolebinding user-clusterrole-clusterbinding \
--clusterrole=name-of-clusterrole \
--serviceaccount=namespace:name-of-serviceaccount
#apply a manifest of RBAC objects
kubectl auth reconcile -f my-rbac-rules.yaml
#delete any Role, ClusterRole, RoleBinding, ClusterRoleBinding
#with the following command adjusted to the case
kubectl delete role name-of-role
```

As for testing our policies, we can do this by using the `kubectl auth can-i` commands, where we can impersonate whatever user we want. They look like the following:

```
#kubectl auth can-i verb resource -n namespace --as user
#returns yes or no depending if the established policies allow it or not
kubectl auth can-i get secrets -n namespace-a --as user-in-A
```

At this point, we have carried out all the configurations needed, so we can move on to implementing our scenario-based analysis.

4.1.2 Scenario-based analysis

In this section, we will examine various scenarios demonstrating the implementation of RBAC. We will observe close to real-life applications of RBAC while highlighting the strengths and limitations of its mechanism and the paradigm itself.

Scenario 1

In this scenario, we are going to show a simple case of RBAC authorization where users have access to the secrets only in their respective namespaces. It involves two distinct namespaces (`namespace-a` and `namespace-b`), each containing two secrets (in `namespace-a`: `secret-a1`, `secret-a2`; in `namespace-b`: `secret-b1`, `secret-b2`), accessible only to users associated with the respective namespace. We establish two roles (`secret-reader-a` and `secret-reader-b`) that possess permissions exclusively to `get`, `list`, and `watch` over secrets as resources within their designated namespaces. We have already authenticated our two users (`user-in-A` and `user-in-B`), one within each namespace, and so we can bind them to their corresponding roles, `secret-reader-a` and `secret-reader-b`.

Figure 4.2 illustrates an outline of the scenario.

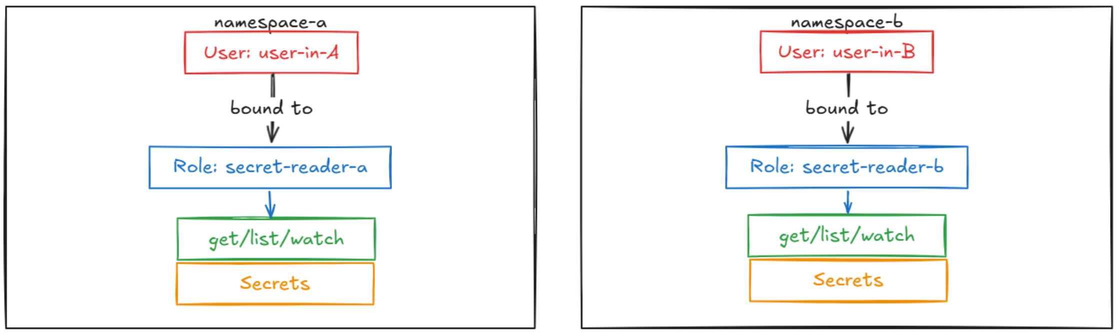

Figure 4.2. RBAC scenario 1

We have all our objects ready in manifests, and once we have deployed everything in our cluster, we can move on to testing our scenario to see if it behaves as we want it to.

```
kubectl auth can-i get secrets -n namespace-a --as user-in-A #return yes
kubectl auth can-i get secrets -n namespace-b --as user-in-B #return yes
kubectl auth can-i get secrets -n namespace-b --as user-in-A #return no
#if we run
kubectl get secrets -n namespace-b --as=user-in-A
#returns Error from server (Forbidden): secrets is forbidden:
#User "user-in-A" cannot list resource "secrets" in API group ""
#in the namespace "namespace-b"

#if we run
kubectl get secrets --as=user-in-A
#returns Error from server (Forbidden): secrets is forbidden:
#User "user-in-A" (or "user-in-B") cannot list resource "secrets"
#in API group "" in the namespace "default"
```

We tested that the users are able to see all secrets belonging to their namespace (the namespace appointed by the role they are bound to), so our implementation is behaving correctly as expected. With this scenario, we were able to see how easy it is to set authorization with RBAC and deploy it in the cluster.

Scenario 2

In this scenario, we are going to show another simple case of RBAC authorization, where a user has limited access only to a specific secret in a namespace. Similar to scenario 1, there are two distinct namespaces (`namespace-a` and `namespace-b`), each containing two secrets (in `namespace-a`: `secret-a1`, `secret-a2`; in `namespace-b`: `secret-b1`, `secret-b2`). Unlike Scenario 1, this scenario involves a single user with limited access, permitted to view only a certain secret within their namespace. We do this by establishing a role (`restricted-secret-reader`) that is permitted only to `get`, `list`, and `watch` `secret-a1` within the namespace `namespace-a`. This is achieved by using the `resourceNames` associated with the role. Consequently, the behavior and execution of the verbs are modified. The user `restricted-user-in-A`, bound to this role, must explicitly indicate the name of the secret they are permitted to access when executing the `kubectl get` command; otherwise, an error will occur.

Figure 4.3 illustrates an outline of the scenario.

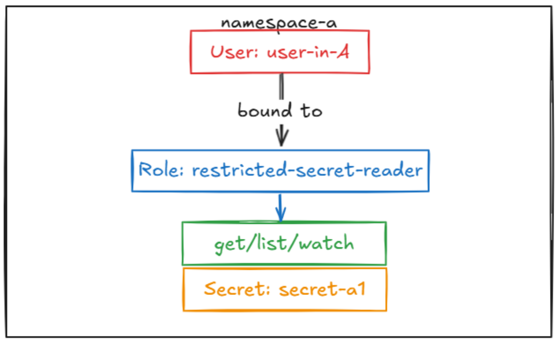

Figure 4.3. RBAC scenario 2

The following commands test the behavior:

```
kubectl get secrets secret-a1 -n namespace-a --as=restricted-user-in-A
#gets you the secret
kubectl get secrets -n namespace-a --as=restricted-user-in-A
#returns an error: Error from server (Forbidden): secrets is forbidden:
#User "restricted-user-in-A" cannot list resource "secrets" in API
#group "" in the namespace "namespace-a".
```

We mentioned this behavior in the RBAC implementation: when `list` or `watch` operations are restricted by `resourceName`, to obtain authorization, clients are required to include in their requests for `list` or `watch` a `metadata.name` field selector corresponding to the specified `resourceName`. Here we can see more clearly that when utilizing `resourceNames`, the `list` verb fails to filter the results to display just the resources to which one has access, hence limiting the ability for a comprehensive listing action altogether.

Also, just to check that `restricted-user-in-A` has only access in `namespace-a`, we test with the following command:

```
kubectl auth can-i get secrets -n namespace-b --as=restricted-user-in-A
#returns no
```

It is important to note that this scenario was initially designed to implement a form of discrimination based on resource details, preferably a label rather than a name. This is due to the knowledge that certain resources are ephemeral and may cease to exist only to be regenerated under a different name; thus, it is mostly conceptually meaningless to differentiate based on the name of the resource. However, this scenario revealed a limitation of RBAC, being its inability to restrict resources based on their labels. This will only be possible in the Webhook scenario later on.

Scenario 3

In this scenario, we are going to show the creation of the admin role in two different ways, one that is limited in certain namespaces and one that has access cluster-wide. As in the prior cases, there are two distinct namespaces (`namespace-a` and `namespace-b`), each containing two secrets (in `namespace-a`: `secret-a1`, `secret-a2`; in `namespace-b`: `secret-b1`, `secret-b2`). We create a ClusterRole named `cross-namespace-secret-reader` that has the permission to `get`, `list`, and `watch` secrets as resources. Although these permissions are now defined at the cluster level, this does not imply that they are automatically granted across all namespaces. To demonstrate this, we have a user named `admin-user` and bind it with RoleBindings in each namespace with the ClusterRole. We grant this user access to many namespaces by establishing unique RoleBindings for each namespace, thereby scoping access to those specific namespaces. This allows us to explicitly specify the namespace to which it has access, but it becomes challenging to manage when numerous RoleBindings require human configuration.

Figure 4.4 illustrates an outline of the scenario.

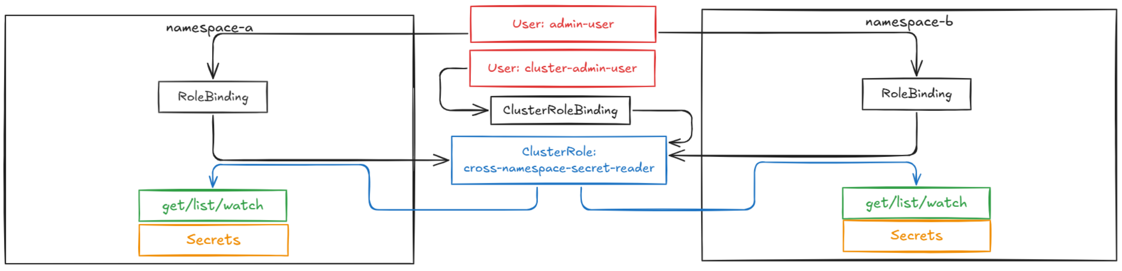

Figure 4.4. RBAC scenario 3

We can test this setting with the following commands:

```
kubectl auth can-i get secrets -n namespace-a --as=admin-user#returns yes
kubectl auth can-i get secrets -n namespace-b --as=admin-user#returns yes
kubectl auth can-i get secrets -n kube-system --as=admin-user#returns no
```

To grant full access listing all secrets across all namespaces with a single command, we bind a new user, `cluster-admin-user`, with a ClusterRoleBinding to the previously specified ClusterRole. The new user now possesses access to secrets in all existing namespaces and any future namespaces that may be created. This authorization is extremely powerful and presents a significant risk. We can see a simple example when running the following command:

```
kubectl get secrets --as=cluster-admin-user --all-namespaces
#returns all secrets cluster-wide
```

It indeed can read all four secrets deployed from the two namespaces, as well as secrets from the `kube-system` namespace (or, in our case, also the `cilium-secrets` namespace due to our cluster configuration). To mitigate risk, we have to reconsider our strategy, avoiding permitting cluster access unless absolutely essential and instead adhering to the principle of the least privilege. In this case, we observed the simplicity of granting cluster-wide access, as well as the potential risks that may arise from improperly configured policies.

Scenario 4

This scenario adopts a more realistic perspective on behavior in applications deployed within Kubernetes. We have an application deployment that reads environmental variables from a ConfigMap and does so via a ServiceAccount. Its access permissions are restricted to read-only at runtime and must not be allowed for alteration in any manner. Instead, we have an administrator who can `create`, `update`, `patch`, and `delete` the ConfigMaps, in addition to having read access.

Prior to clarifying the process for achieving this, we will briefly review the concept of service account [21]. A ServiceAccount in Kubernetes is a non-human account that offers a unique identity within the cluster. They are crucial in authenticating with the API server or enforcing security policies based on identity. Upon the creation of a cluster, Kubernetes automatically generates a ServiceAccount object named `default` for each namespace within the cluster. By default, this account possesses no permissions, aside from the basic API discovery permissions (essentially read-only access) that Kubernetes allocates to all authenticated principals when RBAC is enabled. Defining customized ServiceAccounts enables more precise access control using RBAC by associating them with specific roles or cluster roles. The question that arises is, what benefits do service accounts provide us? Similar to how users require identities to use Kubernetes, apps operating within the cluster require identities to interact with Kubernetes resources in a regulated manner. This also facilitates the separation of human identities and machine identities, hence simplifying the management of permissions and auditing as necessary. This way, automated techniques for interacting with the Kubernetes API without the necessity of human credentials are more easily accomplished. Overall, it makes it possible for fine-grained access control to be accomplished with RBAC, in addition to the fact that ServiceAccounts are scoped to namespaces.

Let us return to our scenario. We create a simple deployment in `namespace-a` named `sample-app`, which possesses a distinct identity via a ServiceAccount referred to as app-service-account. The `sample-app` must read configuration from ConfigMap (we are talking about data like `DATABASE_URL` and `LOG_LEVEL`), but must not have the capability to modify them. Both ConfigMap and ServiceAccount are deployed within the same namespace, named as `namespace-a`. Subsequently, we create an RBAC Role named `configmap-reader`, which possesses the capability to `get`, `list`, and `watch` ConfigMaps within `namespace-a`. We bind this role with RoleBinding to the `app-service-account`.

Figure 4.5 illustrates an outline of the scenario.

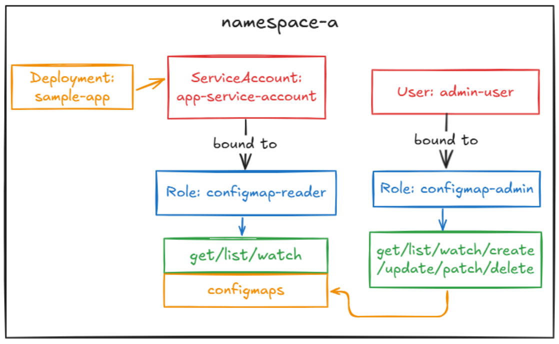

Figure 4.5. RBAC scenario 4

We can test whether the service account can access the ConfigMap with the following command:

```
kubectl auth can-i get configmaps \
--as=system:serviceaccount:namespace-a:app-service-account
\-n namespace-a #returns yes

kubectl auth can-i update configmaps \
--as=system:serviceaccount:namespace-a:app-service-account \
-n namespace-a #returns no
```

To gain the necessary permissions for modifying (associated with verbs like `create`, `update`, `patch`, `delete`) ConfigMaps, rather than just reading (associated with verbs like `get`, `list`, `watch`), we bind our new user, `admin-user` via a RoleBinding to a Role named `configmap-admin`, which includes all previously mentioned permissions for ConfigMaps within `namespace-a`. We can verify that the admin user is capable of modifying the ConfigMap with the following command:

```
kubectl auth can-i update configmaps --as=admin-user -n namespace-a
#returns yes
```

So with this scenario, we were able to show how service accounts can be utilized in achieving a more fine-grained RBAC policy.

Scenario 5

This scenario extends Scenario 4 to demonstrate that ServiceAccount and RBAC permissions are namespace-scoped. In contrast to the prior case, we have introduced an additional ConfigMap in `namespace-b`, which bears the same name as the one in `namespace-a`, although with different data. The identical ServiceAccount name (`app-service-account`) has been used in both namespaces. The RBAC Role and RoleBinding are applied exclusively to `namespace-a`, hence leaving `namespace-b` without explicitly granted permissions. This configuration demonstrates that the ServiceAccount in `namespace-a` may access the ConfigMap within that namespace; however, the identically named ServiceAccount in `namespace-b` cannot access ConfigMaps, as having the same name does not imply it is the same ServiceAccount. The `admin-user` rights are consistent with the previous scenario, indicating that it can manage just the ConfigMaps in `namespace-a`, but not in `namespace-b`.

Figure 4.6 illustrates an outline of the scenario.

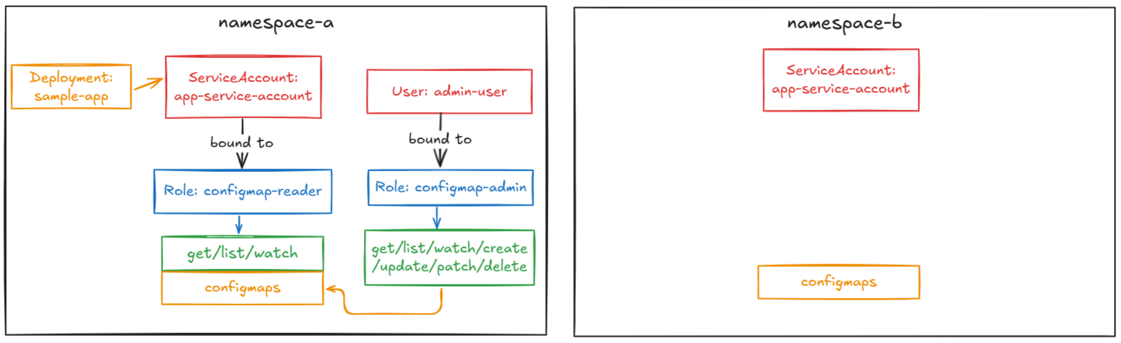

Figure 4.6. RBAC scenario 5

We can test this implementation with the following commands:

```
kubectl auth can-i get configmaps \
--as=system:serviceaccount:namespace-a:app-service-account \
-n namespace-a #returns yes
kubectl auth can-i get configmaps \
--as=system:serviceaccount:namespace-a:app-service-account \
-n namespace-b #returns no
kubectl auth can-i get configmaps \
--as=system:serviceaccount:namespace-b:app-service-account \
-n namespace-b #returns no
kubectl auth can-i update configmaps --as=admin-user \
-n namespace-a #returns yes
kubectl auth can-i update configmaps --as=admin-user \
-n namespace-b #returns no
```

In this scenario, we demonstrated the powerful interaction of RBAC with service accounts, which facilitates fine-grained access control; yet, it simultaneously introduces complexity in granting access across namespaces.

Scenario 6

This scenario aims to demonstrate a behavior common to multi-team or multi-environment settings within Kubernetes, where shared configurations exist across environments. One of the cases, and the one we are going to consider here, is an application deployment in one namespace that requires access to ConfigMaps from another namespace. We will configure the ServiceAccount associated with our `sample-app` deployment to be granted read permissions for ConfigMaps in namespaces other than its own. The `sample-app` deployment and the ServiceAccount `cross-namespace-reader` are located in `namespace-a`, while the required ConfigMap is situated in `namespace-b`. Additionally, an RBAC Role named `configmap-reader` is created in `namespace-b`, granting permissions to `get`, `list`, and `watch` resources as ConfigMaps. This Role is associated with a RoleBinding named `cross-namespace-configmap-reader` in the destination namespace,

namespace-b, linking to the ServiceAccount cross-namespace-reader from its original namespace-a, specifying this namespace clearly in the subject reference along with the name of the service account.

Figure 4.7 illustrates an outline of the scenario.

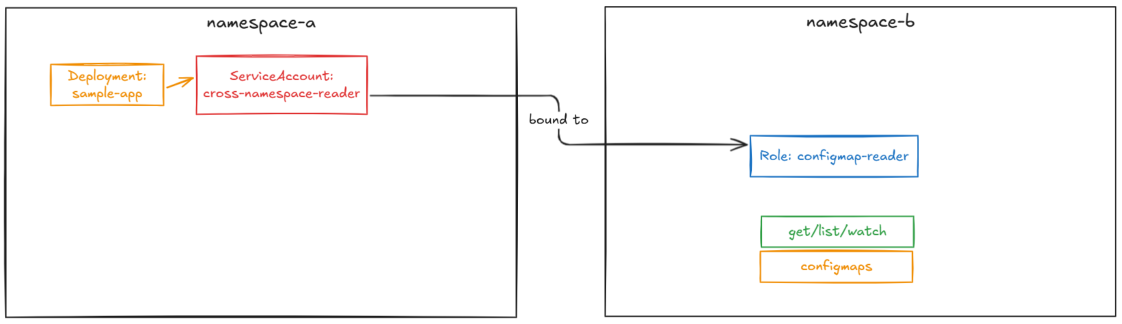

Figure 4.7. RBAC scenario 6

We can verify this works with the following command:

```
kubectl auth can-i get configmaps \
--as=system:serviceaccount:namespace-a:cross-namespace-reader \
-n namespace-b #returns yes
```

In this scenario, we successfully addressed the limitation established in the preceding one. While the process for obtaining this access level is comprehensible, it may become challenging when managing numerous cross-namespace policies.

Scenario 7

In this scenario, we are going to focus more on restricting resources and subresources via RBAC authorization. In this case, we are proceeding to RBAC regulations for pod-related activities. A typical real-world case involves providing log access without execution authorization.

Let us start with a brief overview of logging and execution functionalities. Both are pod-specific activities executed as subresources of the Pod resource, namely `pods/log` and `pods/exec`. Container runtimes collect logs from streams (`stdout` and `stderr`) of processes of containers, which are subsequently obtained by the `kubelet` (node agent) and made accessible via the Kubernetes API server through the `pods/log` subresource. Log access can be granted via RBAC by assigning a read-only role. Exec creates an interactive connection to the container within a pod, enabling bidirectional streaming. The client (in this case, `kubectl`) initiates an exec session request over the Kubernetes API, which validates the request and forwards it to the relevant `kubelet`, which subsequently sets up a connection, via the container runtime, to the container. To give executive access using RBAC, it is necessary to configure permissions on the subresource named `pods/exec`.

In our situation, we intend to permit log access without allowing execution. However, it is known that RBAC provides only permissive rules, with no capacity to define denying rules. The question that arises is, how shall we configure this to achieve the level of access desired?

First, let us clarify why this level of access is required. To adhere to the security principle of least privilege, you may have the ability to inspect logs for diagnostic purposes, but not necessarily possess the authority to execute commands within active containers. This would also assist in audit compliance, which often demands a distinction between observation (logging) and modification (execution). This rule set also prevents accidental alterations and safeguards data. Logs are, by default, read-only and generally comprise operational data that is not inherently sensitive. However, the `exec` command permits potentially destructive operations, and given that pods may contain sensitive information, this presents an elevated risk.

We will now proceed to the implementation of this scenario. Initially, we need a Pod to serve as a sample, along with a ServiceAccount. Following that, we create an RBAC Role that defines permissions to `get`, `list`, and `watch` pods and their logs. Instead, for `pods/exec`, we are deliberately leaving the verbs in the rules empty, so tricking RBAC into implementing a form of deny rule. Last, we associate this Role with RoleBinding to the ServiceAccount, remembering that all of these are confined to the `namespace-a` scope.

Figure 4.8 illustrates an outline of the scenario.

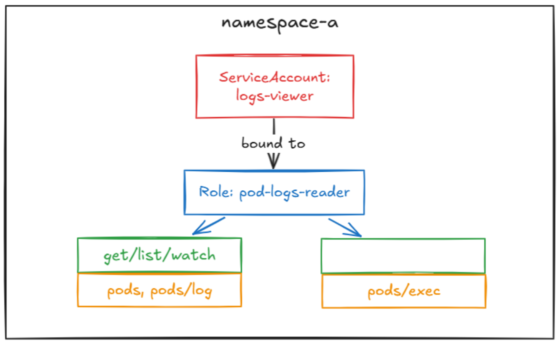

Figure 4.8. RBAC scenario 7

We can test with the following command:

```
kubectl auth can-i get pods/log \
--as=system:serviceaccount:namespace-a:logs-viewer \
-n namespace-a #returns yes

kubectl auth can-i create pods/exec \
--as=system:serviceaccount:namespace-a:logs-viewer -\
n namespace-a #returns no
```

However, it is essential to recognize that we are presuming our pod is accessed exclusively via the ServiceAccount to be more realistic. Let us hypothetically imagine that an attacker acquires valid `kubectl` credentials (via a compromised machine, theft of kubeconfig files, etc.), and these credentials possess broader rights than those granted to ServiceAccounts. If this attacker tried, it would effectively execute, bypassing the poorly constructed RBAC rules, which may be well set for service accounts but not for human users. In this case, it turned out necessary to implement user-level RBAC with stringent restrictions applicable to all human users. We can test the attacker's situation with the following command:

```
kubectl exec -n namespace-a <name-of-the-pod>\
--as=authenticated-user -- date #would actually execute
```

In this scenario, we successfully illustrated the rules governing the use of resources and subresources, as well as the particular risks associated with inadequately constructed RBAC authorization.

Scenario 8

In Scenario 7, we observed the implementation of log access without execution rights, presuming that we accessed the pods utilizing ServiceAccount. However, towards the conclusion of the scenario, we examined a hypothetical circumstance in which an attacker acquires valid `kubectl` credentials (via a compromised machine, theft of kubeconfig files, etc.), and these credentials possess broader permissions than those granted to ServiceAccounts. If this attacker attempted to exploit this vulnerability, it would successfully execute, bypassing the poorly constructed RBAC rules, which may be appropriately configured for service accounts but not for human users. In this instance, it became necessary to implement user-level RBAC with stringent restrictions applicable to all human users.

In this scenario, we will demonstrate how to accomplish it. The RBAC Role will stay unchanged, giving read-only permissions to `pods/logs` and deliberately leaving empty the verbs applicable on `pods/exec`, renamed as `restricted-pod-viewer`, and will be bound to subjects identified as the group called `system:authenticated` using RoleBinding, hence applying to all authenticated users. This would prohibit all human users from executing commands within the pod.

Figure 4.9 illustrates an outline of the scenario.

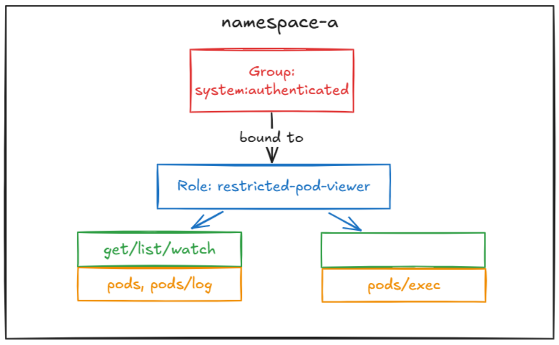

Figure 4.9. RBAC scenario 8

Following that, we use an authenticated test-user to evaluate our configuration using the following command:

```
kubectl auth can-i create pods/exec -n namespace-a --as=test-user
#returns no
```

It is important to understand that by establishing this RoleBinding with `system:authenticated` as a subject, we are granting permissions to all authenticated users without altering or canceling any pre-existing rights they may own. This remains a limitation to RBAC. For improved security, we must either adopt a multi-layered strategy incorporating Pod Security Standards, Admission Controllers, OPA Gatekeeper, etc., or establish role-based user accounts alongside separated admin credentials, break-glass procedures, and identity management integration.

Scenario 9

This scenario examines the construction of RBAC authorization that restricts actions based on the permitted verb, and we demonstrate this in a common strategy setting employed in production environments to avert service interruptions by removing pods. We will set up a role that permits the development team to `create`, `update`, and `patch`, in addition to having reading access, but prohibits deletion in the production environment. Additionally, to be more realistic, we have a lower-level environment where the development team faces fewer restrictions and is permitted to delete as well. This demonstrates their ability to operate autonomously and safely. The separation between creation and deletion permissions is a standard practice in production environments for various reasons. Firstly, it safeguards against unintentional downtime, therefore contributing to the maintenance of a stable environment. The decommissioning process is rendered more restricted, requiring elevated permissions for the removal of production resources, in this case, pods; however, this principle can be applied more broadly to the entire set of resources. Ultimately, it enables secure CI/CD pipelines wherein automated deployments can build, but not eliminate, pods or other resources. However, when evaluating lower-level environments, such as development or testing, we should offer greater flexibility while still preserving isolation to prevent interference with production.

To execute this scenario, we initially create two namespaces, named `production` and `testing`, to be consistent with the narrative, and then deploy a Pod and a ServiceAccount within each namespace. Following that, we construct a Role within the `production` environment, whose rules apply to pods and permit the verbs `get`, `list`, and `watch` (for read access), as well as `create`, `update`, and `patch`, while excluding `delete`. Next, we associate this Role via RoleBinding to the ServiceAccount that identifies our pod in the `production` environment. We also establish an RBAC role within `testing` that applies to pods and permits the verbs `get`, `list`, and `watch` (for read access), as well as `create`, `update`, and `delete`. Next, we associate this Role with RoleBinding to the ServiceAccount that identifies our testing pod.

In practical applications, this pattern would typically be extended to encompass higher-level resources, like Deployments or StatefulSets, rather than being applied straight to pods; nonetheless, the underlying concept remains the same.

Figure 4.10 illustrates an outline of the scenario.

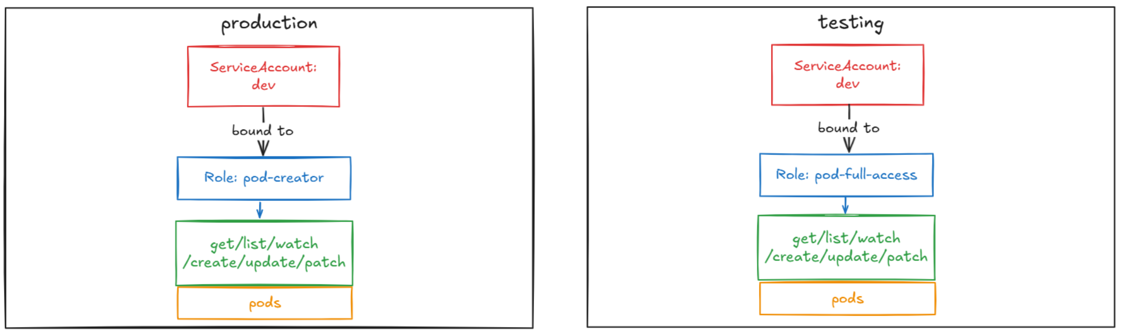

Figure 4.10. RBAC scenario 9

We can test our setup by running the following commands:

```
kubectl auth can-i create pods --as=system:serviceaccount:production:dev\
-n production #returns yes
kubectl auth can-i delete pods --as=system:serviceaccount:production:dev\
-n production #returns no
kubectl auth can-i delete pods --as=system:serviceaccount:testing:dev \
-n testing #returns yes
kubectl auth can-i delete pods --as=system:serviceaccount:testing:dev \
-n production #returns no since no cross-namespace access
```

In this case, we successfully demonstrated the construction of RBAC authorization that restricts based on permitted verbs.

Scenario 10

This scenario demonstrates how to aggregate multiple ClusterRoles into a single, unified ClusterRole, enabling the creation of more complex roles through a combination of smaller, specialized ones. A controller operating within the control plane monitors ClusterRole objects that possess a `aggregationRule`. This `aggregationRule` specifies a label selector for the controller to utilize in matching other ClusterRole objects for integration into the rules field. Be aware that the control plane will replace any values manually configured in the rules field of the aggregate ClusterRole. To modify or add rules, alterations should be made directly to the ClusterRole objects picked by the `aggregationRule`. The default user-facing roles also utilize ClusterRole aggregation. As a cluster administrator, you can set rules for custom resources to add to the default ones. To do this, you include `rbac.authorization.k8s.io/aggregate-to-admin: "true,"` `rbac.authorization.k8s.io/aggregate-to-edit: "true,"` or `rbac.authorization.k8s.io/aggregate-to-view: "true"` within the labels field.

In our case, we establish three base ClusterRoles: `pod-reader`, `log-reader`, and `events-reader`, each granting access to examine pods, pod logs, and Kubernetes events, respectively. Every base role bears a common label: `rbac.example.com/aggregate-to-monitoring: "true"`. We create an aggregating ClusterRole named `monitoring-role`, which has an empty rules field, and use the `aggregationRule` to indicate the base ClusterRoles to be aggregated according to the label specified.

A ServiceAccount named `monitoring-account` has been deployed in the monitoring namespace and is bound to our aggregating ClusterRole using a ClusterRoleBinding. It is essential to recognize that although the ServiceAccount is a namespaced resource, this does not constrain the permissions conferred by a ClusterRole via a ClusterRoleBinding, as these are applicable across all namespaces, since they themselves are cluster-wide resources not associated with any particular namespace. We specify the namespace for the ServiceAccount solely to indicate its location; nonetheless, it has been granted cluster-wide access via the ClusterRoleBinding.

Figure 4.11 illustrates an outline of the scenario.

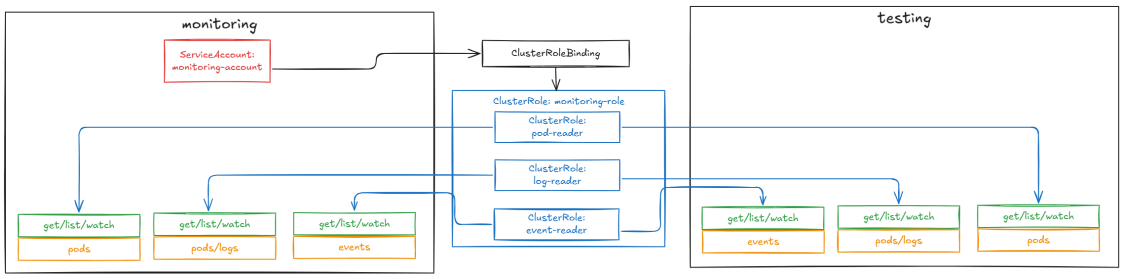

Figure 4.11. RBAC scenario 10

We can test our setup by running the following commands:

```
kubectl auth can-i get pods \
--as=system:serviceaccount:monitoring:monitoring-account #returns yes
kubectl auth can-i get pods/log \
--as=system:serviceaccount:monitoring:monitoring-account #returns yes
kubectl auth can-i get events \
--as=system:serviceaccount:monitoring:monitoring-account #returns yes
kubectl auth can-i get pods -n testing --\
as=system:serviceaccount:monitoring:monitoring-account #returns yes
kubectl auth can-i create pods \
--as=system:serviceaccount:monitoring:monitoring-account #returns no
```

This methodology of constructing aggregating ClusterRoles yields multiple advantages. It offers a better understanding of modularity by deconstructing permissions into logical and reusable components. It improves maintainability, as updates are required solely for the base roles without requiring alterations to the overall structure. Supports scalability by allowing the addition of new permissions through the creation of new base roles with corresponding labels. Finally, it maintains consistency by universally applying these common permission settings. This method is highly beneficial for establishing uniform role descriptions that can grow over time without requiring updates to all bindings, hence preserving the overall structure. It is important to highlight that this can only be accomplished using RBAC among all native Kubernetes authorization mechanisms.

Scenario 11

In this scenario, we are going to explore how the privilege escalation prevention works in RBAC. So, we have two ClusterRoles: `team-leader-role-manager` and `super-team-leader-role-manager`, which vary in their privileges. They possess permissions to `get`, `list`, `watch`, `create`, `update`, and `delete` pods, services, and deployments. However, regarding the creation or modification of roles and role bindings, the `team-leader-role-manager` is restricted to creating or updating roles and role bindings solely within the scope of their existing permissions and cannot escalate privileges. The `super-team-leader-role-manager` possesses additional privileges to create Role, ClusterRole, RoleBinding, and ClusterRoleBinding, with extended permissions due to the explicit inclusion of the verbs `escalate` and `bind`.

Subsequently, we assign these two ClusterRoles to two already authenticated users: `team-leader-user` and `super-team-leader-user`, respectively.

Figure 4.12 illustrates an outline of the scenario.

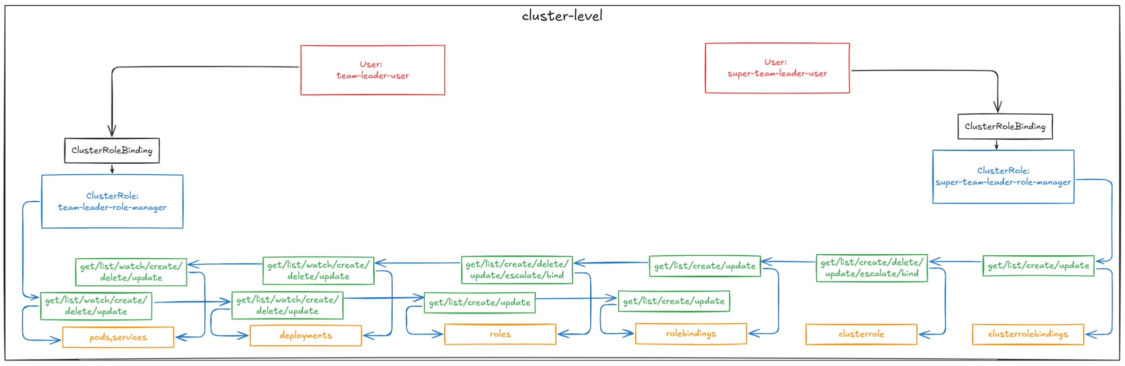
cluster-level
User: team-leader-user
User: super-team-leader-user
ClusterRoleBinding
ClusterRole: team-leader-role-manager
ClusterRoleBinding
ClusterRole: super-team-leader-role-manager
get/list/watch/create/ delete/update
get/list/watch/create/ delete/update
get/list/watch/create/ delete/update
get/list/watch/create/ delete/update
get/list/create/delete/ update/escalate/bind
get/list/create/update
get/list/create/update
get/list/create/update
get/list/create/delete/ update/escalate/bind
get/list/create/update
pods,services
deployments
roles
rolebindings
clusterrole
clusterrolebindings

Figure 4.12. RBAC scenario 11

We may test our configuration using the following commands:

```
kubectl create role developer --verb=get,list,watch \
--resource=pods,deployments --as=team-leader-user \
-n testing #successfully executed and role created

kubectl create rolebinding dev-binding --role=developer \
--user=dev-user --as=team-leader-user -n testing
#successfully executed and rolebinding created

kubectl create role admin-role --verb=get,list,update,delete \
--resource=nodes --as=team-leader-user -n testing
#gives an error: (Error from server (Forbidden):
#roles.rbac.authorization.k8s.io "admin-role" is forbidden:
#user "team-leader-user" (groups=["system:authenticated"]) is attempting
#to grant RBAC permissions not currently held:{APIGroups:[""],
#Resources:["nodes"], Verbs:["get" "list" "update" "delete"]})

kubectl create rolebinding super-binding \
--clusterrole=cluster-admin --user=special-user \
--as=team-leader-user -n testing
#gives an error: failed to create rolebinding:
#rolebindings.rbac.authorization.k8s.io "super-binding" is forbidden:
#user "team-leader-user" (groups=["system:authenticated"]) is attempting
#to grant RBAC permissions not currently held:{APIGroups:[""],
#Resources:[""], Verbs:[""]}{NonResourceURLs:[""], Verbs:[""]}

kubectl create role super-role --verb=get,list,update,delete \
--resource=nodes --as=super-team-leader-user -n testing
#successfully executed and role created

kubectl create rolebinding super-binding \
--clusterrole=cluster-admin --user=special-user \
--as=super-team-leader-user -n testing
#successfully executed and rolebinding created
```

In the end, this protection is essential for numerous reasons. Firstly, it helps in preserving security boundaries, ensuring that the delegation of work does not jeopardize security. It enforces the principle of least privilege, as even role administrators cannot surpass their authorized access level. It also prevents backdoor privileges, stopping users from creating roles with augmented permissions to obtain malicious advantages. Finally, it mitigates the danger of accidental or intentional misconfiguration, ensuring that administrators cannot create risky roles regardless of their intentions. Likewise, we must remain alert and cautious when granting explicit authorization for escalation or binding, since this could present significant risks and contradict the previously mentioned considerations. Last, it is important to highlight that this can only be accomplished with RBAC among all native Kubernetes authorization mechanisms.

Scenario 12

This scenario will examine the creation of non-resource-related RBAC rules. It is essential to recognize that non-resource URLs are generally endpoints such as `/metrics`, `/livez`, `/readyz`, etc., mostly utilized for system-level pathways and utility endpoints. They do not signify Kubernetes objects/resources such as pods, services, secrets, etc. In our case, we are creating two ServiceAccounts with varying access levels to Kubernetes metrics endpoints. First, `metrics-reader`, and then `basic-health-checker`. We then create two ClusterRoles: one for `metrics-viewer`, which can access non-resources such as `/metrics` and `/api`, and another for `health-checker`, which can access non-resources like `/livez` and `/readyz`. They are bound to `metrics-reader` and `basic-health-checker` by ClusterRoleBindings, respectively.

A brief overview of the selected endpoints:

- `/metrics` endpoint delivers Prometheus-formatted metrics for Kubernetes components. These metrics comprise performance data of API server, etcd interactions information, and resource use statistics. This endpoint is essential for assessing the performance and health of your Kubernetes cluster.
- `/livez` endpoint serves as a Kubernetes API server interface that signals the

present operational status of the API server, indicating whether the component is "alive."

- `/readyz` endpoint is another endpoint of the API server that indicates the current status of the API server, similar to the livez endpoint, which signals whether the component is prepared to handle requests.

Figure 4.13 illustrates an outline of the scenario.

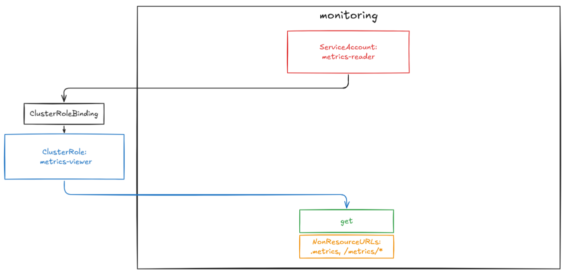

Figure 4.13. RBAC scenario 12

We can test our setup with the following commands:

```
kubectl auth can-i get /metrics \
--as=system:serviceaccount:monitoring:metrics-reader #returns yes
kubectl auth can-i get /api \
--as=system:serviceaccount:monitoring:metrics-reader #returns yes
kubectl auth can-i get /livez
--as=system:serviceaccount:monitoring:basic-health-checker #returns yes
kubectl auth can-i get /metrics \
--as=system:serviceaccount:monitoring:basic-health-checker #returns no
```

In that scenario, we demonstrated the simplicity of managing access to non-resources in Kubernetes.

This scenario concludes our practical analysis of RBAC in Kubernetes.

4.1.3 Strengths and limitations

RBAC provides significant advantages focused mainly on manageability and operational efficacy. The model's fundamental strength lies in its conceptual and practical simplicity, making it easy to comprehend, implement, and manage. Administrators experience minimal initial configuration overhead for the cluster, as the native Kubernetes RBAC mode typically requires no complex setup at startup. Authorization objects, including Roles, ClusterRoles, RoleBindings, and ClusterRoleBindings, are dynamically controlled via the API by any client with the appropriate permissions, allowing efficient management. Moreover, the use of default roles and role bindings accelerates the initial bootstrapping and offers clear examples for customization. The capacity to aggregate rules from various ClusterRoles into a singular role increases flexibility and reduces redundancy in the formulation of complex permission sets for users or groups. RBAC inherently integrates essential security measures to prevent privilege escalation, whereby a user might alter roles to get elevated permissions above their original rights, and contains mechanisms for the secure bootstrapping of initial administrative permissions. Ultimately, authorization decisions within the RBAC paradigm are generally faster due to the uncomplicated nature of the permission evaluations required through native Kubernetes objects.

Nonetheless, its simplicity and manageability have inherent trade-offs, mainly regarding granularity and scalability in complex settings. A notable constraint is the relatively coarse granularity of access control that can be attained. Permissions are strictly linked to predefined roles, often very box-shaped, limiting the implementation of fine-grained, context-aware authorization rules. For example, establishing rules that dynamically adjust according to certain resource attributes, user context, or environmental factors is beyond the scope of the RBAC paradigm. This inflexibility directly leads to another significant disadvantage, the risk of role explosion. As organizational complexity and unique permission requirements grow, administrators are forced to set up a constantly increasing number of highly specialized roles to meet complex access needs. The growth of roles affects management, auditing, and the overall understanding of the access control layout, presenting considerable scalability issues. Moreover, RBAC does not provide explicit definitions of ”deny” rules, which can occasionally offer a clearer and more concise method for articulating certain exclusionary policies than constructing overly liberal ”allow” roles. This Kubernetes mechanism’s shortcomings also involve listing and filtering operations, since it lacks the built-in ability for end-users to efficiently filter results according to their individual permissions, which may sometimes unintentionally reveal more information than necessary.

To summarize the usefulness of RBAC, it is especially well-suited for environments that prioritize administrative simplicity, fast deployment, and clear hierarchical structures. Its advantages are evident in contexts characterized by relatively static, well-defined roles and predictable permission needs. The mechanism excels in leveraging default roles, preventing privilege escalation, and enabling fast authorization decisions, which are key operational needs. Consequently, RBAC is particularly useful in organizations or systems of modest complexity, where the minimal administrative load and ease of management surpass the necessity for highly detailed, context-sensitive access regulations, or as only the first step in a chain of several authorization mechanisms, covering the basis before the more context-aware and complex authorization modes take over.

4.2 Attribute-Based Access Control (ABAC)

Attribute-based access control establishes a paradigm of access control that grants access rights to users based on attributes combined into policies. These attributes vary through a wide range, from user to environment, resources, object, etc.

The Kubernetes official documentation does not provide a detailed explanation of the ABAC paradigm itself, so we turned our focus to different sources. According to [22] description of the ABAC paradigm, attributes denote the properties or values of a component associated with an access event. ABAC evaluates the components’ **attributes** in relation to predefined **rules**. These rules specify the **permissible attribute combinations** necessary for the subject to effectively execute an action on an object. This way, the ABAC mechanism can assess attributes based on their interactions within an environment and apply rules and relationships accordingly. **Policies**, then consider attributes to delineate permissible and forbidden access situations.

Consequently, the primary advantage of ABAC is flexibility. The constraints of policy-making are fundamentally determined by the attributes that must be considered and the circumstances that the computational language may articulate. ABAC facilitates the widest range of subjects to access the maximum resources without requiring administrators to define relationships between each subject and object. These attributes and access control regulations may be further adjusted to align with the requirements of the organization. Moreover, accommodating new subjects becomes very easy in ABAC, provided that new subjects have been fitted with the required attributes for the access control regulations, making it unnecessary to alter current rules or object properties.

Up until this point, it seems that the advantages of the ABAC paradigm significantly surpass the disadvantages. However, one disadvantage to be considered before adopting attribute-based access control is the complexity of implementation. Administrators must manually specify characteristics, allocate them to each component, and establish a centralized policy engine that dictates permissible actions for these attributes based on certain criteria. The model's focus on attributes complicates the assessment of rights granted to individual users prior to the establishment of all attributes and regulations. Nonetheless, this yields significant rewards since we can duplicate and repurpose attributes for similar components and user roles, and the flexibility of ABAC ensures that managing policies for new users and access scenarios is a largely automated process.

Now that we have a clear view of the paradigm itself, we can go on to consider the official guide [23] on how ABAC is implemented in Kubernetes. ABAC policies are defined in a **single policy file** upon cluster startup, facilitating centralized configuration and management. The file format consists of one JSON object per line. Where each line represents a **policy object**, wherein each object is a map containing the following properties: `versioning` properties and `spec` properties. The `versioning` properties are two: `apiVersion` and `kind`. They both are useful to help with versioning and conversion of the format of the policies. Kubernetes supports `abac.authorization.kubernetes.io/v1beta1` for `apiVersion` and `Policy` as `kind`. It is worth mentioning here that the ABAC API is in beta version, and native ABAC mode in Kubernetes is slowly becoming obsolete, giving more priority to the RBAC one since it is better supported. Let's move on to `spec` properties, which can be divided into `subject-matching` properties, `resource-matching` properties, `non-resource-matching` properties, and `readonly` property that works as a verb-matching property. As the name suggests, each of these sets of properties defines the properties for each group they are named after. So, in `subject-matching` properties, we can distinguish `user` and `group`. `user` corresponds to the username from the token file for authentication, which we will see in detail later. Instead, if we specify `group`, it then corresponds to one of the groups associated with the authenticated user, like `system:authenticated` corresponds to all requests that are authenticated. `system:unauthenticated` corresponds to all requests that lack authentication. Moving on to `resource-matching properties`, we can distinguish `apiGroup`, `namespace`, and `resource`. The names are pretty explanatory at this point; the only thing worth mentioning here is that when used, the wildcard symbol (*) matches all the resource types it is being used on. Then we have `non-resource-matching` properties that have only `nonResourcePath` listed, and this one is very self-explanatory. Here, the wildcard symbol can be interpreted in two ways: either matching all non-resource requests when used alone or matching all subpaths of the path it is associated with. Last, we said the `readonly` property is the verb-matching property because when set to `true`, it indicates that the `resource-matching` policy is applicable solely to `get`, `list`, and `watch` verbs, whereas the `non-resource-matching` policy is restricted to the `get` operation only. Instead, if `false`, it applies all the verbs with no discrimination whatsoever. It is important to note that an unset property is equivalent to a property assigned the zero value corresponding to its type (e.g., an empty string, 0, or false).

To put it in a visualized manner, Figure 4.14 shows how policy is constructed in ABAC.

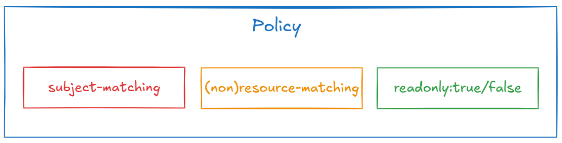

Figure 4.14. ABAC policy

Now that we know how a policy is constructed, we can explain the workings of the authorization algorithm. It starts with a request arriving at the API server from any client. This request contains attributes that align with the properties of the already explained policy object. These attributes are identified as soon as the request is received. Undefined attributes are assigned the default value according to their type, and also the wildcards will correspond to any value of the relevant attribute. The attribute tuple is verified for compatibility with each policy in the policy file in order. If at least one line corresponds to the request attributes, the request is approved. Otherwise, it is rejected.

In conclusion, this high-level overview of ABAC in Kubernetes demonstrates that it is an intuitive authorization paradigm while also providing significant flexibility in its use. However, this implies an increased complexity in the correct design of your policy. We will also see that the configuration and maintainability show an increased complexity. Similar to our approach with RBAC, before starting our scenario-based analysis to explore the practical implementation of ABAC, we will further examine the specifics of its configuration for use within the cluster.

4.2.1 Configuration

As anticipated, the configuration of the ABAC authorization mechanism in Kubernetes comes with an increased complexity compared to the RBAC one. This starts from the fact that **ABAC mode** does not come as a default authorization mode when you set up a normal cluster. You need to accommodate further configurations in order to be able to use it. So to begin with, to activate ABAC mode, it is necessary to add ABAC to the `--authorization-mode= ABAC,Node,RBAC`, and to provide `--authorization-policy-file=SOME_FILENAME` at cluster startup. When setting up our cluster with the kubeadm tool, and after numerous trials and mistakes, we have determined that it is more effective to set it up by providing a YAML configuration file with the `--config` option in the `kubeadm init` command like below:

```
sudo kubeadm init --config=kubeadm-config.yaml
```

At this point, after having created the cluster successfully, we can follow the same configuration for `kubectl`, CNI, and joining worker nodes as in the RBAC configuration.

Going back to the reason why we chose to set up our cluster like this. Although the previously mentioned flags are accessible via the command line, and it is commonly recommended to alter your API server manifest, we discovered that these techniques frequently result in errors and lead to the failure of the API server pod.

The configuration file looks like the following:

Listing 4.7. Kubeadm configuration file for ABAC mode

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
networking:
  podSubnet: "10.245.0.0/16"
  serviceSubnet: "10.255.0.0/16"
apiServer:
  extraArgs:
    authorization-mode: "ABAC,Node,RBAC"
    authorization-policy-file:
       "/etc/kubernetes/abac/policies.jsonl" #path to policy file
    token-auth-file: "/etc/kubernetes/token.csv" #path to token
       file
  extraVolumes:
    - name: abac-policies
      hostPath: "/etc/kubernetes/abac"
      mountPath: "/etc/kubernetes/abac"
      readOnly: true
      pathType: DirectoryOrCreate
    - name: token-auth
      hostPath: "/etc/kubernetes/token.csv"
      mountPath: "/etc/kubernetes/token.csv"
      readOnly: true
      pathType: File
```

It is important to note that we are using apiVersion: `kubeadm.k8s.io/v1beta3` rather than `v1beta4`, as the latter results in an error during cluster initialization. The ordering of authorization modes is crucial when many authorization modules are configured, as they are evaluated in order; if any authorizer authorizes or refuses a request, that decision is promptly returned without considering additional authorizers. To accurately implement a system utilizing ABAC, we are positioning it as the primary authorization mode in the hierarchy, succeeded by Node and RBAC. The removal or omission of these two default entries leads to the failure of the API server pod. In the configuration file, we see the declaration of `Volumes` for our policy file, with appropriate mounting specified by `mountPath`, `hostPath`, `name`, and `type`. This is crucial, since omitting this will prevent our cluster from accurately recognizing the policy file.

The configuration file also indicates that we are declaring `token-auth-file` as an extra argument and appropriately mounting it as an extra volume, as it is essential for user authentication and accurate policy parsing. We previously authenticated our users using X.509 certificates for RBAC scenarios; however, through experimentation, we have identified issues with this approach in ABAC scenarios. Instead, it requires a **token-based** authentication, where a `user` in `subject-matching` properties corresponds to a username linked to an authenticated user via the `user name` from the `--token-auth-file`. It is crucial to note that these files and directories must already exist prior to the initialization of our cluster, and we can thereafter alter their contents to suit our demands. This provides a brief description of file formats, with further details to be elaborated in the following scenarios.

As explained in the introductory section for ABAC, its policy file format must be in `jsonl`, with one JSON object per line, where each line describes a policy object. This is an example of what a policy file would resemble:

Listing 4.8. ABAC policy file sample

```
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1",
    "kind": "Policy", "spec": {"user":"user-1","namespace":
    "namespace-1", "resource": "*","apiGroup": "*","readonly":
    true }}
{"apiVersion": "abac.authorization.kubernetes.io/v1beta1",
    "kind": "Policy", "spec": {"user":"user-2","namespace":
    "namespace-2", "resource": "pods","apiGroup": "*","readonly":
    false}}
```

The authentication token CSV file comprises a minimum of three columns: `token`, `user name`, and `user UID`, with optional `group names` following. If there are several group names, the column must be enclosed in double quotes. Tokens may endure indefinitely. This is an example of what an authentication tokens file would resemble:

Listing 4.9. Authentication token file sample

```
user-1-token,user-1,1000,group-1
user-2-token,user-1,2000,"group-1,group-2"
```

At this point, we have set up our cluster, activated ABAC correctly, and become familiar with the file formats for both policy and token. We can now go over the technicalities of how we can apply changes in our cluster since we are dealing with two static files.

Initially, we must ensure that the tokens and policies are accurately configured. Both files are supposedly read-only, preventing direct modifications at the path where they reside. Therefore, when we want to do alterations, we must create a new file and then move or copy it to the appropriate directory, which might require sudo privileges. Additionally, it is essential to ensure that the file permissions are appropriately configured, so we might need to modify the access level of the file. Once having successfully modified the needed files, it is necessary to restart the API server for the modifications to take effect. To execute this correctly, we must temporarily remove the API server manifest (`kube-apiserver.yaml` file in our case, located in `/etc/kubernetes/manifests`) and afterwards restore it. This will force the API server to restart. We could alternatively remove the pod instance, as it is a static pod in kubeadm; however, it occasionally fails to perform a complete restart, resulting in the changes in files not being applied. The subsequent instructions will do a complete restart of the API server:

```
sudo mv /etc/kuberentes/manifests/kube-apiserver.yaml /tmp/
sleep 10
sudo mv /tmp/kube-apiserver.yaml /etc/kubernetes/manifests/
```

Confirmation of a successful restart of the API server instance is indicated by a new `Restart` entry (along with the elapsed time since the restart), a `Status` of `Running`, and a `Ready state` of `1/1` when you run:

```
kubectl get pods -n kube-system
```

In the cases where it happened that we could not access the cluster at all and needed to quickly restore functionality, the following commands were handy:

```
sudo systemctl stop kubelet
sudo systemctl start kubelet
```

As for testing our policies, we can do this by using the `kubectl auth can-i` commands, where we can impersonate whatever user we want. They look like the following:

```
#kubectl auth can-i verb resource -n namespace --as user
#returns yes or no depending if the established policies allow it or not
kubectl auth can-i get secrets -n namespace-a --as user-in-A
```

After having covered also how to handle making changes in the cluster, it brings us to the realization of one of the biggest downsides of ABAC in Kubernetes. Since we are declaring our policy and authentication token files statically, to implement any changes to policies and tokens, we are forced to access the control plane directly, edit the static files on disk, and, moreover, restart the API server to apply the changes. This is very different from what we encountered with RBAC, where we were able to manage all our access control dynamically through the Kubernetes API objects like Roles, ClusterRoles, RoleBindings, and ClusterRoleBindings via any client holding the appropriate credentials. Instead, here you need direct access to the control plane node, which, logically, in many environments (like those in cloud provider-powered clusters), is a node restricted to access due to security reasons. Therefore, limiting the use of ABAC to only self-managed Kubernetes clusters. Moreover, it obviously causes service disruptions, since you need to restart the API server each time you want to modify your policy. Which, because of the fact that this policy file exists as a static file on disk, the version control is challenging. This also leads us to the issues of scalability and maintainability, as the growth of the cluster renders the policy file increasingly complex and its management more burdensome.

The challenges associated with establishing and maintaining native ABAC in Kubernetes clusters are the cause for its declining popularity, despite its potentially more expressive policy capabilities. ABAC has been mainly restricted to earlier Kubernetes versions, and transitioning to RBAC is recommended.

At this point, we have carried out all the configurations needed, so we can move on to implementing our scenario-based analysis.

4.2.2 Scenario-based analysis

In this section, we will examine various scenarios demonstrating the implementation of ABAC. We will observe close to real-life applications of ABAC while highlighting the strengths and limitations of its mechanism and the paradigm itself.

Scenario 1

For the first scenario, we will follow the example presented in [24] to familiarize ourselves with a basic configuration. We have a single namespace: `projectcaribou`, and two users: `alice` and `bob`, who have varying levels of access permissions; `alice` has full permissions, whilst `bob` has only read-only ones. We create the namespace `projectcaribou` and deploy a pod within it for the sake of completeness.

Figure 4.15 illustrates an outline of the scenario.

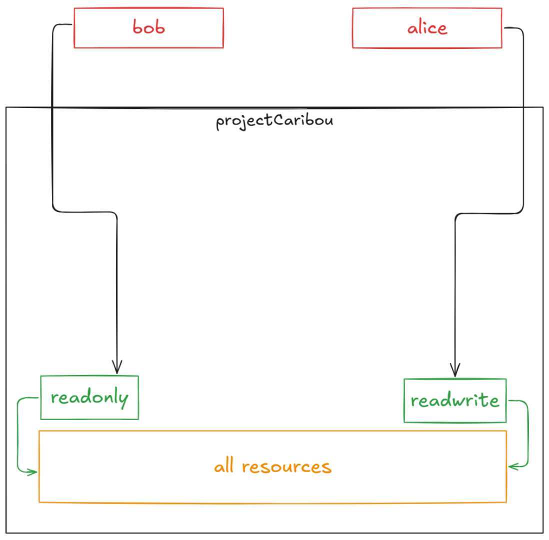

Figure 4.15. ABAC scenario 1

After these steps, we are ready to test our scenario with the following commands:

```
kubectl auth can-i get pod -n projectcaribou --as=alice #returns yes
kubectl auth can-i get pod -n projectcaribou --as=bob #returns yes
kubectl auth can-i delete pod -n projectcaribou --as=alice #returns yes
kubectl auth can-i delete pod -n projectcaribou --as=bob #returns no
kubectl auth can-i get pod -n kube-system --as=alice #returns no
```

Insights gained during the implementation of this straightforward scenario indicate that ABAC complexity can rapidly escalate. For instance, if we included a policy with `"group":"system:authenticated"` granting comprehensive access to the namespace `projectcaribou` before our policies for `alice` and `bob`, it would intercept all requests and authorize them without consideration for the following policies. It suggests that the order of the policies is crucial, and if necessary, we should position these extensive, inclusive policies towards the end of the policy file to ensure they get processed after. Additionally, clashing policies may result in complications within our system. This may be easily identified and fixed in a small-scale context, but in a real-world situation, it could become quite challenging to handle.

Scenario 2

In this scenario, we will set up a more realistic situation than the previous one and attempt to provide access based on environments. We presume the existence of three distinct environments: development, testing, and production. There are two development namespaces, one designated for each team: `team1-dev` and `team2-dev`. There exists a single namespace in the testing environment, designated for the exclusive use of `team2`: `team2-testing`. Finally, there exists a single namespace designated for production, referred to as `production`. There are three users: one `intern` assigned to `team2`, who has full permissions only in the testing environment but instead has read-only access in the development environments to assist with their tasks. A `senior` employee coordinates activities between the two teams and possesses full access to both development and testing environments. However, neither of these two users possesses permission in the production environment. The `admin` holds complete access in the production environment but lacks permissions in the other environments.

We may test our implementation using the following commands:

```
kubectl auth can-i get pod -n team1-dev --as=intern #returns yes
kubectl auth can-i get pod -n team2-dev --as=intern #returns yes
kubectl auth can-i delete pod -n team1-dev --as=intern #returns no
kubectl auth can-i delete pod -n team1-test --as=intern #returns yes
kubectl auth can-i get pod -n team1-dev --as=senior #returns yes
kubectl auth can-i delete pod -n team1-dev --as=senior #returns yes
kubectl auth can-i get pod -n production --as=intern #returns no
kubectl auth can-i get pod -n production --as=admin #returns yes
kubectl auth can-i get pod -n team1-dev --as=admin #returns no
```

Figure 4.16 illustrates an outline of the scenario.

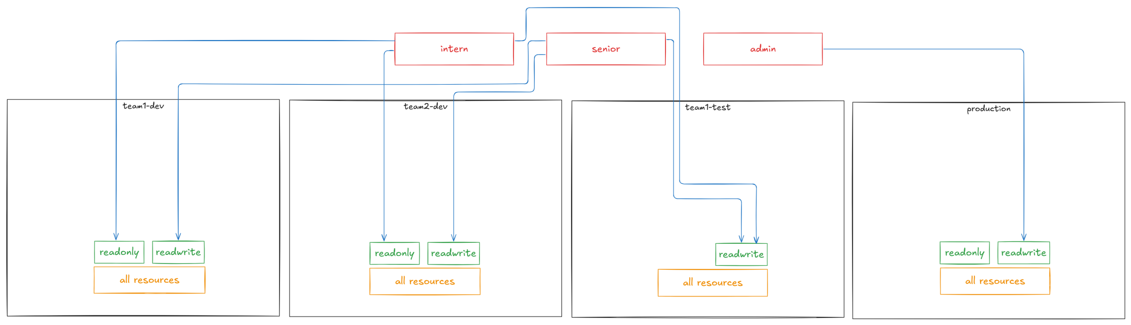

Figure 4.16. ABAC scenario 2

To execute this seemingly simple scenario, it is necessary to define distinct policy entries for each namespace to which the user has access, resulting in a linear increase in the size of the policy file corresponding to the number of namespaces. Pattern matching on namespaces, such as `"namespace": "*-dev"`, cannot be utilized, which would otherwise serve as an efficient tool. It is essential to remember that the wildcard `*` is applicable alone in the following manner: `"namespace": "*"`, exclusively matching all potential namespaces. This presents a drawback regarding the maintainability of ABAC policies, particularly given that the order of policies is crucial and a minor misconfiguration might result in conflicts between policies, hence disrupting access control. In RBAC, this issue would be addressed by establishing a singular Role and using RoleBindings to implement it across various namespaces, no matter the order. But it is crucial to note that with RBAC, there is a potential for role explosion, whereas ABAC mitigates this risk. Assuming that policies are well-crafted, ABAC achieves a greater level of granularity with fewer words. Additionally, the disruption of service caused by API server restarts will repeatedly provide a challenge in our ABAC situations.

Scenario 3

In this scenario, we will implement access control based on resource attributes across various environments. We will reutilize the Scenario 2 framework, including its settings, namespaces, and two of the users: `intern` and `senior`, while adjusting the permissions to better align with our new configuration. Assume we authorize the intern to access all pods across all namespaces, excluding `production`. `senior` can access all pods across all namespaces. Defining the policy for `senior` is straightforward; we simply assign `"namespace":"*"`. However, for interns, it is necessary to create a policy entry for each namespace, deliberately excluding `production` from any of them.

Figure 4.17 illustrates an outline of the scenario.

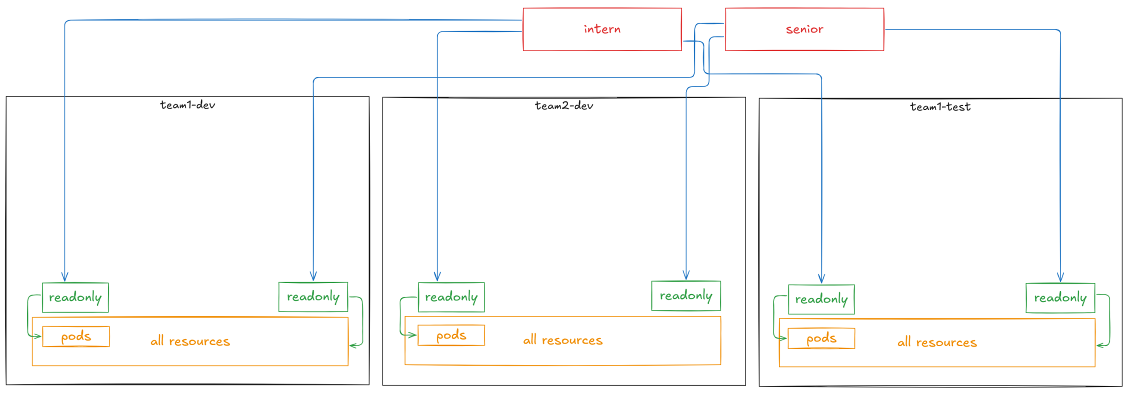

Figure 4.17. ABAC scenario 3

We can test our implementation with the following commands:

```
kubectl auth can-i get pod -n team1-dev --as=intern #returns yes
kubectl auth can-i get pod -n production --as=intern #returns no
kubectl auth can-i get pod -n production --as=senior #returns yes
kubectl auth can-i delete pod -n team1-dev --as=intern #returns no
kubectl auth can-i get service -n team1-dev --as=senior #returns no
```

With this scenario, we are being presented with a challenge compared to RBAC, where one must establish a Role and bind it across namespaces using RoleBinding, however, here it is done in a single line. In RBAC, it is not possible to create a ClusterRole with a ClusterRoleBinding that excludes a specific namespace through labelSelector, which would be a highly effective and rational approach. To restrict the resource, we can use `"resource":"pod"` in both instances, ensuring that the restriction exclusively applies to pods. However, permitting access to `senior` on resources such as pods, services, and deployments, while excluding secrets, would require the specification of policy rules for each resource, hence exponentially increasing the total number of policy entries required. In RBAC, one may effortlessly describe the list of resources directly within the Role or ClusterRole, eliminating the necessity to create several entries for each resource. But then, if we had to establish several levels of access, the box-like roles of RBAC would suffer the same. Instead, with ABAC, however, the additional complexity in policy creation also presents the opportunity for fine-tuned access control, enabling more flexible combinations of rights across users, namespaces, and resources.

Scenario 4

This scenario will examine the functionality of ABAC using service accounts rather than user accounts, as well as the grant of access based on non-resource attributes.

We assume the existence of a ServiceAccount named `metrics-collector-sa` within the `kube-system` namespace that requires rights to access `nonResources` such as `/metrics`. We implement our ServiceAccount and declare in our policy `"user": system:serviceaccount:kube-system:metrics-collector-sa`. The official Kubernetes documentation states that each service account is assigned a corresponding ABAC username generated according to a specific naming pattern

`system:serviceaccount:<namespace>:<serviceaccountname>`, without the need to authenticate it any further. Be aware that creating a new namespace results in the creation of a new service account in the specified format:
`system:serviceaccount:<namespace>:default`.

We may test our implementation using the following commands:

```
kubectl auth can-i get /metrics \
--as=system:serviceaccount:kube-system:metrics-collector-sa #returns yes
kubectl auth can-i get pods \
--as=system:serviceaccount:kube-system:metrics-collector-sa #returns no
```

Figure 4.18 illustrates an outline of the scenario.

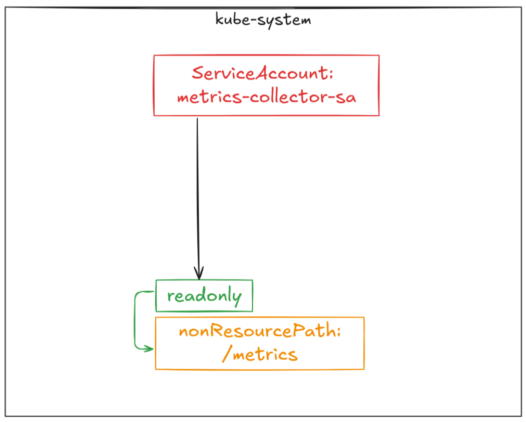

Figure 4.18. ABAC scenario 4

In this scenario, the formulation of rules in ABAC for non-resource URLs is comparatively more straightforward and concise than in RBAC, where it is necessary to create a ClusterRole and ClusterRoleBindings. Although there is no substantial difference, ABAC syntax is somewhat more plain and intuitive.

Scenario 5

This scenario will examine group-based access involving several resource types and the implications of integrating user and group policies.

Assume there are two environments: `production` and `testing`, each containing a distinct namespace. There are two groups of users: `developers` and `testers`. In `developers`, we have `intern`, `dev`, and `senior-dev`. In `testers`, we have a `tester`. We are assigning group-based rights to `developers`, providing them full access to deployments while restricting them to read-only access for pods in `production`. We are providing `testers` with complete access to all resources in the `testing` environment. We are attempting to restrict access permissions for the `intern`, stipulating that while being part of the `developers` group, they possess only read-only access to deployments. Our policies define group-based policy entries for `developers` and `testers`, along with a user-based policy entry for `intern`.

Figure 4.19 illustrates an outline of the scenario.

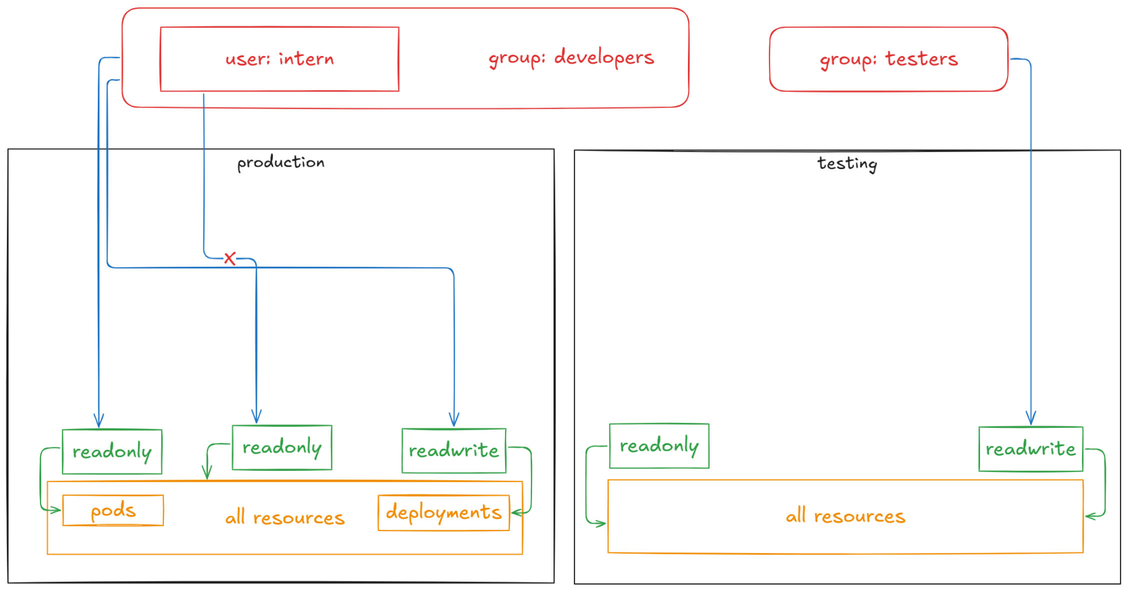

Figure 4.19. ABAC scenario 5

We try to test our implementation with the following commands:

```
kubectl auth can-i get pods -n production --as=dev --as-group=developers
#returns yes
kubectl auth can-i get deployments -n production --as=senior-dev \
--as-group=developers #returns yes
kubectl auth can-i delete deployments -n production --as=senior-dev \
--as-group=developers #returns yes
kubectl auth can-i get deployments -n testing --as=tester \
--as-group=testers #returns yes
kubectl auth can-i delete pods -n testing --as=tester \
--as-group=testers #returns yes
kubectl auth can-i get pods -n production --as=tester \
--as-group=testers #returns no
kubectl auth can-i get pods -n testing --as=dev\
--as-group=developers #returns no
kubectl auth can-i get deployments -n production --as=intern \
--as-group=developers #returns yes
kubectl auth can-i delete deployments  -n prodcution --as=intern \
--as-group=developers #expected to return no but returns yes
```

Although we believed we had restricted its access permissions inherited from the group-based policy for `developers` to read-only, this does not actually work. This is due to a significant characteristic of Kubernetes ABAC authorization. ABAC employs a permissive approach for permissions when multiple policies align, with the more permissive policy prevailing. In native Kubernetes ABAC, there is no notion of "denying policy"; hence, attempting to override a more liberal policy with a more restricted one, as we aimed to do, is ineffective. Thus, the difficulty in articulating "deny" rules in ABAC is a contributing factor to the switch to RBAC in Kubernetes. Although the additive approach to permissions is applicable to RBAC as well, managing restrictions is far more straightforward when addressing the issue with a different architectural perspective. In ABAC, the considerable flexibility is accompanied by significant complexity in such scenarios.

This scenario concludes our practical analysis of ABAC in Kubernetes.

4.2.3 Strengths and limitations

ABAC offers a significant boost in granularity and contextual awareness relative to more simplistic models such as RBAC. Its principal advantage resides in its ability to define complex authorization rules that assess multiple attributes (including user characteristics, resource traits, environmental factors, or action types) to render more dynamic access control choices. This enables highly detailed access control, exceeding the capabilities of role-based assignments, and allows rules to adjust to complex, real-world situations. Furthermore, policies in ABAC are defined within a single, centralized file, providing a unified perspective on the authorization rules that govern the cluster. Authorization decisions in ABAC are typically fast since they still take place within the API server, but loading and parsing each policy object in the policy file results in a slower execution relative to RBAC.

However, this power implies significant operational complexity and management challenges. A notable drawback is the necessity for additional configuration during cluster startup, as the ABAC policy file must be explicitly supplied to the API server. The syntax for constructing sophisticated, attribute-driven rules is inherently complicated, affecting policy definition, validation, and maintenance, which may result in errors and present scalability challenges for large or constantly evolving environments. ABAC policies are statically defined at the cluster startup, so any modification to these policies requires a restart of the API server, resulting in unavoidable service disruptions and excluding dynamic, version-controlled policy updates with associated audit trails, but also limiting ABAC implementation to self-managed Kubernetes clusters. Similar to RBAC, ABAC falls short in establishing explicit deny rules and encounters difficulties in effectively listing and filtering based on user permissions.

To summarize the usefulness of ABAC, it shines in settings requiring highly detailed, context-aware authorization choices that must dynamically adjust based on a multitude of conditions, such as tightly regulated industries or complex multi-tenant systems. Its strengths are especially advantageous in situations requiring fine-grained control over sensitive resources or operations based on combinations of user attributes, resource properties, and environmental context. Thus, ABAC is ideally suited for security-critical applications where the operational burden of policy management and cluster restarts is an acceptable compromise for attaining the necessary level of dynamic, attribute-based access control accuracy, particularly in cloud-native environments abundant in resource metadata.

4.3 Webhook

A WebHook is an HTTP callback, specifically an HTTP POST triggered by an event; it serves as a straightforward event notification through HTTP POST. A web application utilizing Webhooks will send a POST request to a URL upon the occurrence of specific events. When designated, the Webhook mode prompts Kubernetes to consult an external REST service to determine user privileges.

As a consequence, the authorization mechanism of a Webhook can be explained in more detail on how the requests and responses are constructed, the format they follow, and the flexibility offered. Upon encountering an authorization decision, the API server submits a JSON-serialized `authorization.k8s.io/v1beta1` SubjectAccessReview object detailing the action being taken. This object comprises fields that identify the user initiating the request, together with either specifics regarding the resource being accessed or the attributes of the request. We have to keep in mind that Webhook API objects obey the same versioning compatibility regulations as other Kubernetes API objects. So when implementing an authorization Webhook, we should recognize the less strict compatibility guarantees for beta objects and verify the `apiVersion` field of the request to guarantee accurate deserialization. A simple request would look like this:

Listing 4.10. Request to Webhook via SubjectAccessReview object

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "namespace-x",
      "verb": "get",
      "group": "",
      "resource": "pods"
    },
    "user": "user-x",
    "group": [
      "group-x",
      "group-y"
    ]
  }
}
```

In the case of non-resource paths, it would have `nonResourceAttributes` instead of `resourceAttributes`, defining `path` and `verb`.

The remote service is supposed to provide the `status` field of the request and respond with either permission or denial of access. The `spec` field in the response body is disregarded and can be excluded. A positive response, considered permissive, would look like the following:

Listing 4.11. Permissive response from Webhook via SubjectAccessReview object

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": true
    }
  }
```

Instead, there are two methods for denying access. The primary option is favored in most instances, indicating that the authorization Webhook neither permits nor expresses a stance on the request; however, if additional authorizers are enabled, they are allowed the opportunity to approve the request. If there are no additional authorizers or if none provide permission, the request is prohibited. An example of how this denying response would look is as follows.

Listing 4.12. Denying response from Webhook via SubjectAccessReview object

```
  {
    "apiVersion": "authorization.k8s.io/v1beta1",
    "kind": "SubjectAccessReview",
    "status": {
      "allowed": false,
      "reason": "reason why user does not have access"
    }
  }
```

But this option leaves room for edge cases where, due to a misconfiguration, a request that we want to completely deny may only be partially denied by the Webhook authorizer, perhaps leading to approval by another authorizer following that. So it is advised to make a thorough decision in case we have the knowledge that the request should be fully denied. The second option instantly denies, so it doesn't even give a chance to the other configured authorizers that stand in order after the Webhook one. This should be utilized exclusively by Webhooks that possess full knowledge of the complete authorizer settings of the cluster and an example of how it would look is as follows.

Listing 4.13. Absolute denying response from Webhook via SubjectAccessReview object

```
  {
    "apiVersion": "authorization.k8s.io/v1beta1",
    "kind": "SubjectAccessReview",
    "status": {
      "allowed": false,
      "denied": true,
      "reason": "reason why user does not have access"
    }
  }
```

When the `AuthorizeWithSelectors` functionality is activated, the request's `field` and `label`
`selectors` are transmitted to the authorization Webhook. The Webhook may make permission decisions based on the defined `field` and `label selectors`, if desired. The SubjectAccessReview documentation [25] provides instructions on

the interpretation and management of these fields by authorization Webhooks, emphasizing the use of parsed requirements over raw selector strings and detailing the safe handling of unrecognized operators since several CVEs have been identified.

Last, we must also consider that further measures may be necessary to secure our authorization Webhooks. Given the fact that the authorization Webhook functions as an external service, it indicates a transition from the Kubernetes API server directly executing authorization decisions (like when using RBAC and ABAC) to a cluster service taking on this responsibility. Then it is only logical that we must implement additional safeguards to maintain the integrity of this service. To mention some of them: properly establishing TLS for all Webhook traffic and permitting only authenticated traffic to access the Webhook, implementing additional network policies to limit service access to external networks, etc. This is a concern that should not be taken lightly, and it should be addressed in accordance with the specific use case.

We explained the working mechanisms for the Webhook. The key point to highlight from this thorough description is the flexibility presented by this method. By conforming to the protocols of request deserialization and response serialization, we can construct intricate authorization rules, enabling fine-grained access control. Two novel elements have emerged that were previously absent in the native authorization methods of Kubernetes. First, we can now express deny rules. Secondly, the ability to discriminate based on selectors, these being field or label selectors.

However, this higher level of flexibility results in an elevated level of complexity. The effectiveness of our authorization mechanism is directly dependent upon the capabilities of the Webhook we design. This is why constructing custom Webhooks from scratch is uncommon, leading us to favor existing solutions that provide Webhook-like authorization.

4.3.1 Configuration

Prerequisite configurations must be handled to utilize a Webhook as an authorization mechanism. These configurations relate to setting up the cluster to accept the Webhook as a valid authorization mode or setting up the Webhook server itself. The official Kubernetes documentation [26] states that the **Webhook mode** requires a file for HTTP configuration, following the kubeconfig file format, specified using the `--authorization-webhook-config-file=SOME_FILENAME` flag. Instead, there are few existing guides to setting up an authorization Webhook mechanism from scratch, and there is a lack of comprehensive explanations for each step involved. The sole guide we will be examining to some extent [27] tries to describe both how to set up the cluster and how to create a Webhook server. However, the setup of the authorization Webhook has been a process of trial and error until we successfully accomplished its proper functionality.

We begin by generating the **TLS certificates** for the CA (Certification Authority), which in this case will be a self-signed CA, and the Webhook server itself. The CA certificate establishes a basis for trust for TLS connections. The server certificate is provided by your Webhook upon client connection. The server's private key is utilized to establish a secure connection. They fulfill entirely distinct functions and are utilized in various segments of the TLS handshake. Since the certificates were one of the steps that were most confusing in setting up the Webhook, we will provide the following commands to carry out this step correctly.

```
#create a private key for the CA
openssl genrsa -out ca.key 2048
#create a self-signed CA certificate
openssl req -x509 -new -nodes -key ca.key -days 365 -out ca.crt -subj \
"/CN=webhook-ca"
#create a private key for the server
openssl genrsa -out server.key 2048
#create a CSR for the server
cat > server.conf << EOF
[req]
req_extensions = v3_req
distinguished_name = req_distinguished_name

[req_distinguished_name]

[v3_req]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names

[alt_names]
DNS.1 = webhook-server.default.svc.cluster.local
EOF
openssl req -new -key server.key -out server.csr -subj \
"/CN=webhook-server.default.svc.cluster.local" -config server.conf
#sign the server certificate with the CA
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key \
-CAcreateserial -out server.crt -days 365 -extensions v3_req -extfile \
server.conf
#rename the ca.crt since we are might confuse it later in our cluster
mv ca.crt webhook-ca.crt
```

After we are done with the certificates, we can move on to the Webhook configuration file. As mentioned before, it follows the kubeconfig file format and looks like the following:

Listing 4.14. Webhook configuration file

```
#Kubernetes API version
apiVersion: v1
#kind of the API object
kind: Config
#clusters refer to the Webhook service
clusters:
- name: webhook-server
  cluster:
    #CA for verifying the remote service.
    certificate-authority-data: #webhook-ca.crt base64 encoded
       here
    #URL of remote service to query. Must use 'https'. Also, we
       need to provide the service's full name for correctness
    server:
    https://webhook-server.default.svc.cluster.local:443/authorize

# users refers to the API server's Webhook configuration
users:
- name: kube-apiserver
  user:
    token: test-token
    #can also include cert for the Webhook plugin to use
    #and key matching the cert
# kubeconfig needs contexts
current-context: webhook
contexts:
- context:
    cluster: webhook-server
    user: kube-apiserver
  name: webhook
```

Then we can start up our cluster normally with the `kubeadm init` command and follow the same configuration for `kubectl`, CNI, and joining worker nodes as in the RBAC configuration.

However, further steps are required to enable Webhook as an authorization mechanism in our cluster. Prior to changing our API server, we must ensure that the Webhook configuration file and its CA certificate are present on the control node of our cluster. Subsequently, we modify the API server, first to add the Webhook mode as the authorization mode, since the default options are limited to Node and RBAC, and then add the Webhook configuration file and its CA certificate, like below:

Listing 4.15. API server modification for Webhook authorization mode

```
#adding Webhook mode
- --authorization-mode=Node,RBAC,Webhook
#adding the Webhook configuration file
- --authorization-webhook-config-file=
   /etc/kubernetes/webhook-config.yaml
#mounting the Webhook configuration file and its CA certificate
volumeMounts:
    - mountPath: /etc/kubernetes/webhook-config.yaml
      name: webhook-config
      readOnly: true
    - mountPath: /etc/kubernetes/pki/webhook-ca.crt
      name: webhook-ca
      readOnly: true
volumes:
  - hostPath:
      path: /etc/kubernetes/webhook-config.yaml
      type: File
    name: webhook-config
  - hostPath:
      path: /etc/kubernetes/pki/webhook-ca.crt
      type: File
    name: webhook-ca
```

Keep in mind that this way of setting up indicates that the authorization modes will be activated sequentially, with Webhook as the last one. If no applicable RBAC rule exists, the system will proceed to evaluate the Webhook; if no rules are specified there either, the request will be immediately rejected. In the event of a match in the Webhook, the request will be processed with respect to the corresponding rule, either permitting or denying it. Furthermore, be aware that if a rule exists in RBAC, it will be enforced immediately without consulting the Webhook.

Once we are done with the previous step, it is necessary to restart the API server for the modifications to take effect. The procedure is identical to that outlined in the ABAC configuration, either by temporarily removing the API server manifest or by stopping and restarting the `kubelet`.

The cluster has been set up completely, and the only remaining task is to configure the **Webhook server** and deploy it within our cluster.

We chose to develop our Webhook server in Go due to its increasing popularity in cloud computing, intending to adopt an updated approach in the development process. The script for the Webhook server exhibits remarkable flexibility in the formulation of our policies and rules. We can fully leverage the information obtained from the SubjectAccessReview object to construct intricate rules as desired. But not to forget that it comes at a cost of elevated overall complexity. We will give more details on the Go scripts themselves when we discuss scenarios, in order not to overcomplicate our flow here.

Following that, we can easily construct a **Docker image**, simplifying the deployment of our Webhook server as a remote service within the cluster. The Dockerfile shown below is able to construct a lightweight image through a multi-stage build, incorporating the Webhook server certificate and its private key within the Docker image to ensure full completeness. It is essential to ensure that the mentioned files are located in the directory where we are constructing the Docker image. An example of Dockerfile for our Webhook server image is as follows.

Listing 4.16. Dockerfile example for the Webhook server image

```
FROM golang:1.24.1 AS builder
WORKDIR /app
# Copy the Go code
COPY go.* ./
RUN go mod download
COPY . .
```

```
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o
    webhook-server .
# Use a minimal image for the final container
FROM alpine:latest
WORKDIR /app
RUN apk --no-cache add ca-certificates
# Copy the binary from the builder stage
COPY --from=builder /app/webhook-server /app/
COPY server.crt /app/
COPY server.key /app/
# Create a non-root user and switch to it
RUN adduser -D -H -u 10001 webhook && \
    chown -R webhook:webhook /app
USER webhook
# Expose the port respecting the one set in Go script
EXPOSE 8443
# Set the entrypoint
ENTRYPOINT ["/app/webhook-server"]
```

After creating the Docker image, we can deploy it to our cluster using two different approaches. We can either upload it locally to our cluster, but this requires ensuring its presence on every node, which is tedious and limits scalability. Alternatively, we may upload it to a Container Registry and select the image upon deploying our server within the cluster. We opted for the second alternative and utilized the GitHub Container Registry because of its simplicity and effective version control management.

What is left as the final step is to deploy the Webhook server in the cluster. And we can do this simply by creating a Deployment based on the image we build and exposing the container port correctly (in our case, is `8443` as declared in the Dockerfile and, moreover, in the Go script of our Webhook server) and then a Service related to that Deployment with the exposed port corresponding to the one from our Webhook configuration file (in our case, `443`) and target port, or the port where the Service directs traffic to on the container of the Pod (in our case, `8443`).

At this point, the cluster and the Webhook server should be up and running, and we can utilize our Webhook authorization.

Before we conclude this section, we have some important remarks to make. First, sometimes we had problems with DNS resolution when we declared our Webhook server as an internal Kubernetes service URL. As mentioned also in [28], this might be because the API server itself starts before most of the other components of a cluster, also including DNS services (like `CoreDNS`). So, logically, we cannot depend on those to resolve hostnames that are part of the authorizer Webhook configuration. But there are some tricks offered on how to still achieve this and run our Webhook inside the cluster. We actually tried configuring the Webhook service to run with a static `clusterIP` and then have our API server directed to that IP instead of a hostname, but as warned in the guide, it gave a problem regarding the certificates we were using, since they were issued on hostnames and not IP addresses. So we tried modifying the API server with its own hosts entry, resolving manually with `hostAliases` the `clusterIP` of the Webhook service to the hostname declared in our Webhook configuration file and the certificates, and finally it resulted in success.

Second, having to manually restart the API server static pods when we are configuring our cluster to use the Webhook authorization mechanism poses a big drawback in its usage, and that is why we see the Webhook more used within the scope of admission controllers. As stated in [28] again, this act of restarting often demands access to certain components of the underlying system hosting Kubernetes, an access level that is hardly granted in managed Kubernetes settings. So, it leaves us with the sole possibility of utilizing it in self-managed Kubernetes clusters, excluding a significant part of the Kubernetes community. Therefore, even though the flexibility that comes with building your Webhook server from scratch is quite high and beneficial, it is improbable that we will witness similar thriving ecosystems related to custom Webhook authorizer modules as observed with admission control, where the flexibility remains but is accompanied by easy deployment as well.

At this point, we have carried out all the configurations needed, so we can move on to implementing our scenario-based analysis.

4.3.2 Scenario-based analysis

In this section, we will examine various scenarios demonstrating the implementation of Webhook. We will observe close to real-life applications of Webhook while highlighting the strengths and limitations of its mechanism and the paradigm itself.

Scenario 1

In this first scenario, we will demonstrate a basic version of an authorization Webhook. Evaluating the flexibility offered by an authorization Webhook will be beneficial, although it will also highlight the increased complexity associated with developing it from scratch.

We are attempting to provide a certain level of access to a ServiceAccount named `test-user` located in the `default` namespace. The service account will have the capability to `get` and `list` every type of resource and non-resource, but it will lack the right to `delete` any of them.

Figure 4.20 illustrates an outline of the scenario.

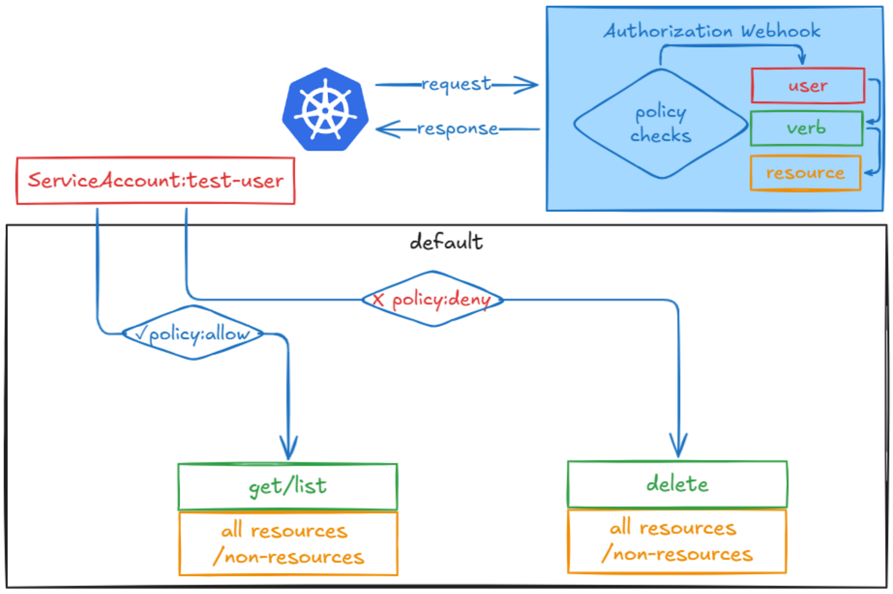

Figure 4.20. Webhook scenario 1

The mechanism works as follows: when this ServiceAccount attempts to carry out an action within the cluster (such as getting pods in the `default` namespace), it will prompt the API server to take an authorization decision. The API server will thereafter POST a JSON-serialized `authorization.k8s.io/v1beta1` SubjectAccessReview object detailing the action. We expect the request body to look like the following.

Listing 4.17. SubjectAccessReview object in Webhook 1st scenario

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "default",
      "verb": "get",
      "group": "",
      "resource": "pods"
    },
    "user": "system:serviceaccount:default:test-user"
  }
}
```

Authorization Webhook receives the SubjectAccessReview via POST and is able to parse any field of interest in a set order to determine whether to permit or deny the request. The following snippet is from the script of the Webhook server implemented in our case.

Listing 4.18. Script snippet of Webhook server in Webhook 1st scenario

```
if req.Spec.User == "system:serviceaccount:default:test-user" {
        if req.Spec.ResourceAttributes != nil {
            if req.Spec.ResourceAttributes.Verb == "get" ||
                req.Spec.ResourceAttributes.Verb == "list" {
                    req.Status.Allowed = true
            }

            if req.Spec.ResourceAttributes.Verb == "delete" {
                    req.Status.Allowed = false
            }
        }

        if req.Spec.NonResourceAttributes != nil {
            if req.Spec.NonResourceAttributes.Verb == "get" ||
                req.Spec.NonResourceAttributes.Verb == "list" {
                    req.Status.Allowed = true
            }

            if req.Spec.NonResourceAttributes.Verb == "delete" {
                    req.Status.Allowed = false
            }
        }
    }
    return ctx.JSON(req)
```

We refer to our SubjectAccessReview object as `req`, and we can parse `req.Spec.User`, checking if it matches the string `system:serviceaccount:default:test-user`. If not, we need to deny access, therefore returning the `status` field in the SubjectAccessReview object as `false`. We accomplish this by setting `req.Status.Allowed` as `false`, which later will be returned back to the API server in JSON format. Once we are sure that the user is the ServiceAccount, we can parse the object further for resources with `req.Spec.ResourceAttributes` are not null and non-resources with `req.Spec.NonResourceAttributes` are not null, and then move on to checking the verbs the access is requested on with `req.Spec.ResourceAttributes.Verb` and `req.Spec.NonResourceAttributes.Verb`. If these last ones are `get` or `list`, we set the `req.Status.Allowed` to `true` and return it back to the API server in JSON format. Instead, if the verb is `delete`, the `req.Status.Allowed` is set to `false`. Once the API server receives the response, it is able to interpret it to allow or deny the initial request for action from the ServiceAccount.

We can test our scenario very simply with the following commands:

```
kubectl auth can-i get pods \
--as=system:serviceaccount:default:test-user -n default #returns yes
kubectl auth can-i delete pods \
--as=system:serviceaccount:default:test-user -n default #returns no
```

As previously said, we can determine the flexibility, customizability, and consequently the power of an authorization Webhook. We hold full control over the fields to consider and the methods of comparison, enabling us to determine the granularity level of access control. We can also leverage the option of integrating external application logic, such as user-defined functions, into the evaluation of the SubjectAccessReview object, which may allow for more complex decision-making. However, it is important to note that substantial flexibility also introduces an additional level of complexity. We must address every detail, every use case, and every edge case, which may be manageable on a small scale but would undoubtedly become considerably more challenging to manage as we scale up with additional users, resources, and rules within the cluster. Ultimately, the decision on whether to use the Webhook authorization mode depends on the specific circumstances and the trade-offs we are willing to make.

Scenario 2

In the present scenario, we will leverage the authorization Webhook's capability to make decisions based on label selectors, a functionality that was previously unavailable with RBAC and ABAC.

An authenticated user named `intern` has been established with X.509 certificates. We aim to allow this user to list solely the pods labeled with `env=test`, excluding those labeled with `env=prod`. When an `intern` attempts to `list` pods with a specific label, the API server must make an authorization decision and will POST a JSON-serialized `authorization.k8s.io/v1beta1` SubjectAccessReview object outlining the action. In the scope of this request, we anticipate the SubjectAccessReview object to look like the following:

Listing 4.19. labelSelector in the SubjectAccessReview object

```
"labelSelector": {
        "requirements": [
          {"key":"label-name", "operator":"In",
             "values":["label-value"]}
        ]
      }
```

Authorization Webhook receives the SubjectAccessReview via POST and is able to parse any field of interest in a set order to determine whether to permit or deny the request. In our case, the authorization Webhook is more interested in parsing the `labelSelector` field once it has established that the request is coming from the user `intern` trying to `list` pods. In our Webhook server script, it looks as follows:

Listing 4.20. Script snippet of Webhook server in Webhook 1st scenario

```
if req.Spec.User == "intern" {
    if req.Spec.ResourceAttributes != nil {
        if req.Spec.ResourceAttributes.Verb == "list" ||
            req.Spec.ResourceAttributes.Resource == "pods" {
            allowedSelector := metav1.LabelSelector{
                MatchExpressions:
                    []metav1.LabelSelectorRequirement{
                        {
                                Key: "env",
                                    Operator: metav1.LabelSelectorOpIn,
                                    Values: []string{"test"},
                               },
                          },
                     }
                     if req.Spec.ResourceAttributes.LabelSelector != nil &&
reflect.DeepEqual(*req.Spec.ResourceAttributes.LabelSelector,
                                    allowedSelector) {
                          req.Status.Allowed = true
                          } else {
                            req.Status.Allowed = false
                          }
                     }
                }

}
return ctx.JSON(req)
```

We refer to our SubjectAccessReview object as `req`, and we can parse `req.Spec.User`, checking if it matches the string `intern`. Then we check the requested verbs and resources with `req.Spec.ResourceAttributes.Verb` and `req.Spec.ResourceAttributes.Resource`. Then we construct the `allowedSelector` and then compare it with the `labelSelector` in our SubjectAccessReview, here referred to as `req.Spec.ResourceAttributes.LabelSelector`. If we find a match, we set the `req.Status.Allowed` to `true` and return it back to the API server in JSON format. If not, the `req.Status.Allowed` is set to `false`. Once the API server receives the response, it is able to interpret it into allowing or denying the initial request for action from the user intern.

Figure 4.21 illustrates an outline of the scenario.

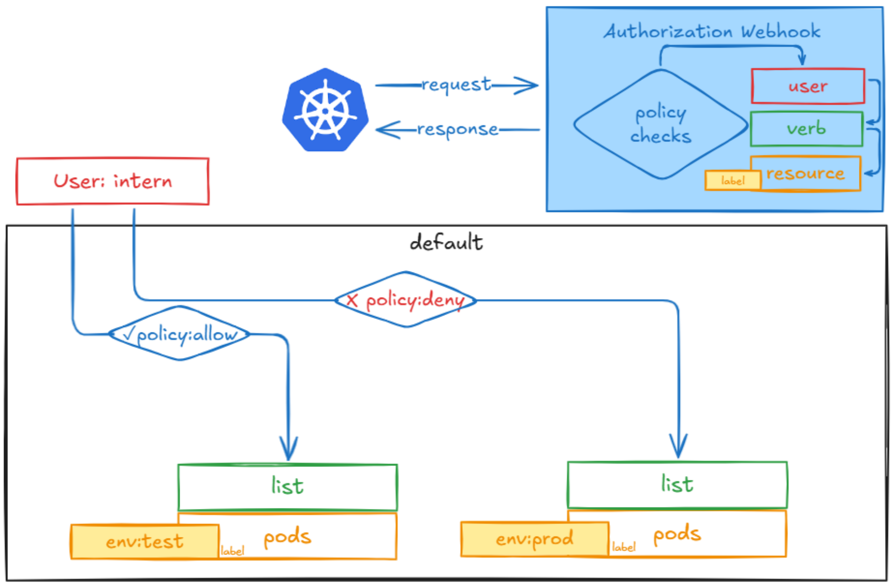

Figure 4.21. Webhook scenario 2

We can test our scenario very simply with the following commands:

```
kubectl auth can-i list pods -l env=test --as=intern #returns yes
kubectl auth can-i list pods -l env=prod --as=intern #returns no
```

In this case, we observed the appearance of a new, powerful feature in access control through the Webhook authorization method. This further demonstrates the efficacy of the authorization Webhook while still highlighting the complexity of developing one from scratch.

This scenario concludes our practical analysis of Webhook in Kubernetes.

4.3.3 Strengths and limitations

Webhook authorization provides exceptional flexibility and customization by delegating access control decisions to an external service. The primary advantage lies in its ability to execute highly detailed and dynamic authorization policies tailored to precise requirements, potentially integrating complex business logic, real-time context, or external application logic on comparative tasks, surpassing the functionalities of the other native mechanisms of RBAC or ABAC. This paradigm naturally enables centralized policy management, allowing for unified governance across various tenants in a cluster or even multiple Kubernetes clusters from a singular authoritative source. Importantly, policy modifications are administered dynamically within the external service, where updates are implemented instantaneously without necessitating cluster restarts or inducing service disruptions except for those at cluster startup, thus enabling dynamic security management. Moreover, in contrast to RBAC and ABAC, the Webhook architecture clearly accommodates the construction of "deny" rules, hence enhancing its expressive capability for specific security policies.

However, such power and flexibility lead to considerable operational complexity and interdependencies. A primary drawback is the need for additional configuration during cluster startup, which involves precise setup of the Webhook endpoint and related certificate authorities for secure connection, but also service disruptions due to API server restarts, limiting its implementation to self-managed Kubernetes clusters. The mechanism's core dependency on an external service causes significant risks. Like that of the external service becoming a single point of failure, in that its unavailability impacts all authorization decisions within the cluster. It also introduces an innate latency overhead for each API request as the cluster awaits the external service's response. Establishing, securing, and sustaining a resilient, highly available external authorization service is complex, requiring considerable skill and resources. Similar to previous mechanisms, Webhooks also lack inherent solutions for efficiently listing and filtering based on user permissions. Moreover, securing the communication between the API server and the authorization Webhook, usually over mutual TLS, and hardening the external service against attacks imposes significant operational burdens.

Webhook authorization is ideally deployed in environments requiring extensive customization and flexibility, where previous RBAC or ABAC mechanisms are inadequate. Its advantages are essential for organizations requiring centralized, high-level policy enforcement across complex environments. This mechanism is particularly effective in high-security or heavily regulated environments that require customized authorization logic, which integrates dynamic, real-time data feeds, like threat intelligence and business context, or extensive integration with external systems, assuming that the operational complexity of managing and securing the external authorization service is an acceptable compromise.

Chapter 5

Open source authorization solutions

5.1 Limitation of native mechanisms and evolution paths

A thorough analysis of native authorization mechanisms in Kubernetes reveals significant limitations that restrict their efficacy in complex, dynamic, and large-scale environments. Although each method targets particular aspects of access control, their combined shortcomings highlight a significant weakness: the lack of native support for advanced authorization paradigms required in modern cloud-native ecosystems. This naturally leads us to explore alternative solutions that can enhance fine-grained access control to meet the demands of today's cloud computing world.

In this light, we regard RBAC and ABAC as well-explored authorization paradigms with not much to be extended on. On the other hand, Webhooks hold substantial potential that, unfortunately, is not fully leveraged in their native authorization mechanism form within Kubernetes. It is even somehow overlooked, as Mutating and Validating Webhooks in the Admission Control phase have gained more popularity and are better realized in Kubernetes. Thus, we concentrated on seeking open source solutions that extend the authorization Webhook mechanism with new and avant-garde underlying paradigms that are able to provide what is lacking from the native mechanisms considered.

We are going to analyze two of them in this chapter: the **Policy-Based Access Control (PBAC)** paradigm realized through **Open Policy Agent (OPA)** and the **Relationship-Based Access Control (ReBAC)** paradigm realized through **SpiceDB**. Without further ado, let us move on, not just exploring the general workings from a high-level view, but also delivering a scenario-based analysis and results drawn from it.

5.2 Policy-Based Access Control (PBAC)

We will first provide an overview of the Policy-Based Access Control (PBAC) paradigm itself and subsequently discuss its implementation via a specific open source solution. Even though we will see several converging points between the paradigm notion and the architecture of OPA as the chosen open source, we think it is beneficial to have the full picture.

Policy-Based Access Control (PBAC) is an authorization mechanism that regulates access to resources according to policies aligned with an organization's business objectives [29] [30]. The PBAC paradigm employs both **attributes** and **roles** to formulate **policies**, in contrast to conventional access control paradigms that utilize just roles or attributes to determine access rights. This feature renders PBAC a genuinely potent and dynamic method for parameter control. In contrast to old methods, policies are now enforced automatically, are scalable, and include a structured framework for implementation. The PBAC paradigm implements policies in a totally different manner, leveraging a methodology known as **Policy-as-Code** (PaC), indicating that policy can be directly coded. It eliminates the manual labor of compliance teams and integrates into the code, thus automating its procedures concurrently. All processes, from version control to review and validation, are automated entirely.

According to [30], there are numerous advantages to automating policy administration with Policy-as-Code. It demonstrates significant improvements in dynamic flexibility, security strength, and compliance adherence, establishing it as a leading modern authorization framework.

The primary advantage is its dynamic and flexible policy enforcement, allowing for detailed access decisions via real-time assessment of several parameters. In contrast to static models, PBAC integrates attributes of user, resource, action, and environment. This ”if-then” decision framework enables firms to establish context-sensitive regulations, such as prohibiting access systems from non-corporate devices on weekends, ensuring accurate, adaptive governance in accordance with changing operational demands. The model's intrinsic flexibility enables effortless expansion across various organizational sizes, while centralized management allows for rapid policy distribution throughout all interconnected systems.

In terms of security and reliability, PBAC reduces human error and systemic weaknesses via automation and privilege separation. Automating policy evaluation and updating removes manual oversight shortcomings characteristic of manual systems. Architecturally, PBAC implements rigorous privilege separation, guaranteeing that breaches in one network segment do not spread laterally. This confinement method, along with proactive policy enforcement, mitigates significant security vulnerabilities inherent in reactive RBAC models. The framework's uniform application of policies across the system mitigates configuration drift and improves security-in-depth strategies against insider threats and data theft.

To guarantee compliance and consistency, PBAC offers auditable and methodical governance crucial for regulatory adherence. The unified policy engine organically connects access restrictions with legal demands, like GDPR, by codifying data privacy and security needs into enforceable regulations. Organizations can consistently limit access to sensitive data across hybrid environments, producing immutable decision logs for auditing purposes. This consistency enables compliance demonstrations, speeds up policy alignment after mergers or expansions, and guarantees that access control advances in sync with regulatory landscapes. Thus, PBAC surpasses conventional access frameworks by integrating compliance, security, and operational agility into a unified, policy-driven architecture.

Having gained a clearer understanding of the paradigm, we may proceed by focusing on the practical application of it through OPA as an open source solution.

5.2.1 Open Policy Agent (OPA)

Open Policy Agent (OPA) is an open source, general-purpose **policy engine** intended to separate policy decision-making from application logic. Created by **Styra** and now under incubation at the Cloud Native Computing Foundation, it is now the go-to choice for a policy engine unifying policy enforcement across the stack. It facilitates cohesive, context-sensitive policy enforcement across several domains, including microservices, Kubernetes, CI/CD pipelines, and infrastructure-as-code technologies. Utilizing a high-level declarative language known as **Rego**, OPA enables the precise articulation of policies as code, thereby simplifying dynamic oversight of authorization, resource allocation, compliance, and configuration management without altering foundational services [31].

Before we focus on the working details of OPA, let us start by defining what a policy is [32]. A **policy** is a collection of rules that regulates the software service's behavior. The policy is versatile in what it can define, from rate limitations to identification of trusted servers, deployment clusters for an application, authorized network routes, or accounts from which a user may withdraw funds. Authorization is a specific type of policy that often determines which individuals or computers are permitted to execute particular actions on designated resources. Authorization is often confused with Authentication: the process by which individuals or systems verify their identity. Authorization and policy in general frequently employ the outcomes of authentication, such as username, user attributes, groups, and claims; however, they base judgments on a broader array of information beyond mere user identity. Abstracting from authorization back to policy clarifies the distinction further, as certain policy decisions are unrelated to users, like in the cases where policy only defines invariants that must be maintained within a software system, say, all binaries must originate from a trustworthy source. Policies can be implemented manually based on documented regulations or implicit conventions that encompass an organization's culture. Policies may be enforced by application logic or statically set upon deployment.

Currently, policy is frequently an integral, fixed component of the software service it regulates. Open Policy Agent enables the separation of policy from the software service, allowing those responsible for policy to read, write, analyze, version, distribute, and manage policy independently from the service. OPA provides a unified toolbox to separate policy from any software service and to formulate context-aware policies utilizing any desired context. In simpler words, OPA enables the decoupling of policies from any context across any software system.

This brings us to a very important feature offered by OPA, that of **policy decoupling**. When software services enable the declarative specification of policies, they permit updates at any moment without the need for recompilation or redeployment, ensuring automatic enforcement, particularly when choices require rapid processing beyond human capability. In today's setting, the decoupling policy enables the development of scalable software services, enhances adaptability to evolving business needs, improves the detection of violations and conflicts, increases consistency in policy adherence, and reduces the likelihood of human error. The regulations we construct can more readily adjust to the external environment, like factors that the developer may not have anticipated during the software service's design phase.

In the absence of OPA, we would have to develop policy management for our product independently. The essential components, including the policy language (syntax and semantics) and the evaluation engine, must be meticulously planned, built, tested, documented, and updated to guarantee accurate functionality and an optimal user experience for clients. Additionally, one must meticulously evaluate security, tools, management, and other factors. That would result in an enormous amount of effort.

This naturally brings us to the question of how OPA actually works [31] [32]. Architecturally, OPA functions as a lightweight binary or container deployed alongside host applications. Services assign policy decisions to OPA using RESTful API calls, supplying structured JSON data that outlines the request context, like user attributes and resource metadata. Then OPA assesses this information according to established Rego policies, articulated in a specialized, high-level language designed for querying hierarchical document architectures to generate **policy decisions**. This data is typically referred to as a **document**, a set of attributes, a piece of context, or simply "JSON". Most importantly, OPA policies may produce decisions taking into consideration arbitrary structured data, since it is not associated with any specific domain model. Likewise, OPA policies can encapsulate decisions as arbitrary structured data (e.g., booleans, strings, maps, lists of maps, etc.). These decisions are provided as JSON outputs. OPA distinctly differentiates **policy formulation** from **enforcement**, facilitating centralized policy administration and dynamic updating without the need for service redeployment.

To clarify a bit on the document model of OPA, data can be imported into OPA from external sources through push or pull interfaces that function synchronously or asynchronously to policy assessment. All data imported into OPA from external sources is referred to as **base documents**. These primary sources consistently inform the policy decision-making process. Yet, your policies may also influence one another's decisions. Policies typically comprise several **rules** that reference further rules, maybe created by various parties. In OPA, the values produced by rules, also known as **decisions**, are referred to as **virtual documents**. The term "virtual" in this context signifies that the document is generated by the policy, meaning it is not externally sourced into OPA.

Base and virtual documents can denote identical types of information, such as integers, strings, lists, maps, etc. Additionally, Rego allows for the referencing of both base and virtual documents utilizing the identical dot/bracket-style reference syntax. The uniformity in the representation of values and their referencing allows policy authors to master a single method for modeling and referencing information that feeds policy decision-making. Furthermore, due to the absence of a conceptual distinction between the types of values or their references in base and virtual documents, Rego permits the referencing of both base and virtual documents.

And the last thing to emphasize in this introduction of OPA is its primary policy language, **Rego**. It enhances declarative programming paradigms to accommodate intricate rule definitions. It utilizes pattern recognition, logical synthesis, and internal functions to examine hierarchical data. The following shows a snippet of what Rego looks like [33].

Listing 5.1. Rego language in OPA

```
import rego.v1

default allow := false #deny requests by default

#more of RBAC approach
allow if user_is_admin #allow admins to do anything
user_is_admin if "admin" in data.user_roles[input.user] #how you
   actually check is admin

#allow the action if the user has permission to perform that
   action
allow if {
some grant in user_is_granted #find grants for the user
input.action == grant.action #check if the grant permits the
   action
input.type == grant.type #check if the grant permits the action
}

#more of ABAC approach
#allowing the action if the attribute value is valid
allow if {
attribute_is_valid
}
attribute_is_valid if
   data.data_attributes[input.attribute].value == input.value
```

Policies can reference many data sources to implement context-sensitive rules such as RBAC, ABAC, ReBAC [33], or tailored evaluations like ABAC with a broader range of attributes or RBAC with blacklisting [34]. Rules are fundamentally **if-then** logical statements that allow for the encapsulation and reutilization of reasoning using Rego. Rules are categorized as either complete or partial. We define them as complete when the if-then statements assign a singular value to a variable, and we define them as partial if the conditional statements produce instead a set of values and assign all to a variable. If OPA is unable to identify variable assignments that match the rule body, we declare the rule as undefined. Rego's testability and modularity facilitate version control, validation, and simulation of policies in development environments, hence improving dependability and auditability. Essential characteristics comprise **dynamic policy bundling**, wherein OPA routinely retrieves updated policies and external data from distant servers, alongside decision **logging** for **audit trails** and guaranteeing of **idempotent enforcement**. At the end, this allows OPA to naturally connect with cloud-native ecosystems through sidecar installations, such as a Kubernetes admission controller, HTTP middleware, or SDKs. The stateless architecture facilitates horizontal scaling, while tools such as the Rego Playground [35] enhance policy creation and debugging.

5.2.2 Configuration

We start by discussing what the recommended deployment patterns for OPA on Kubernetes are, and then move on to how we configured our clusters to perform our scenario-based analysis.

Deploying OPA as a **Policy Decision Point** (PDP) on Kubernetes is a prevalent method for implementing fine-grained policy management within containerized workloads, and then enforcing policy decisions in real-time with **Policy Enforcement Points** (PEPs).

There are several recommended deployment patterns for OPA on Kubernetes, encompassing both cluster- and application-level deployments [36].

- **Sidecar (Application)** When a Policy Enforcement Point (PEP) application requires low-latency policy decisions, OPA can be implemented as a sidecar within the same pod as the application containers. This design facilitates per-application OPA configuration, exhibits network fault tolerance, and scales with the application. But we need to keep in mind that loading large volumes of data into OPA can be resource-intensive. So, for substantial data loads into OPA, it would be better to go with a centralized OPA deployment to minimize cluster memory consumption or utilize Enterprise OPA for enhanced memory efficiency.
- **The Sidecar (Service Mesh)** Integrates Enterprise OPA and the `opa-envoy-plugin`, both of which enhance OPA's native API functionalities to accommodate Envoy's External Authorization API. This enables OPA to apply L7 policy on requests prior to their landing at the application. In this scenario, the PEP is the Envoy proxy; nevertheless, the OPA sidecar continues to provide additional APIs for local policy determinations as previously mentioned.
- **Cluster Service** At times, operating a cluster service is the optimal method for executing OPA. This deployment approach not only addresses Kubernetes Admission Review requests but also facilitates PEPs apps utilizing a highly available OPA PDP service. But take into consideration that this may result in increased latencies and provide a single point of failure if misconfigured.
- **External Service** Analogous to a cluster service, deploying a cluster OPA PDP service behind a load balancer can effectively facilitate the sharing of OPA with external PEPs. This may be advantageous for apps operating on alternative products or on-premises. The main issues to consider prior to going with this way of deployment are latency and network fault tolerance.

- **Cluster Daemonset** Generally, executing OPA as a Daemonset is not suggested. This pattern may present challenges in scalability for applications and necessitate meticulous resource allocation. Nevertheless, if you are operating with a limited number of pods per node and cannot implement a sidecar, this may serve as an appropriate alternative.

Let us proceed with explaining the route we followed for configuring OPA in our clusters. Initially, we attempted to deploy OPA in the same manner as an authorization Webhook by establishing the Webhook mode in the API server (`kube-apiserver`) along with its configuration files and mounted volumes, setting up the Webhook, certificates, scripts, and manifests, and deploying it within the cluster as a service. The only distinction from the native authorization Webhook lies in the format of the policy declaration. We would encounter the same challenges with DNS resolution, and most importantly, this deployment pattern would still require access to the control-plane node to perform the necessary configuration for the Webhook mode, thus restricting this approach to self-hosted Kubernetes clusters only.

We additionally explored the cluster DaemonSet approach as outlined in the [37] repository. This repository, like most others documenting OPA implementation in Kubernetes, is significantly outdated and mostly nonfunctional. This specific one has incompatibilities in the kubeadm version, image version, Rego language version, and other related aspects.

This led us to reconsider the practical significance of OPA in Kubernetes. The power and benefits of OPA as a unified policy agent throughout the stack are undeniable; yet, in the context of Kubernetes deployment, we observed that OPA's popularity in this area is mostly attributed to its admission controller functionalities. As described in [34], **dynamic Admission Control** is constrained by the fact that Webhooks are invoked exclusively for `create`, `update`, and `delete` actions involving Kubernetes resources. It is, for instance, not possible to refuse `get` requests. However, they possess advantages over the authorization Webhook module as they can reject requests based on the content of a Kubernetes resource. Instead, the authorization Webhook module lacks access to this information. Instead, it takes decisions based on SubjectAccessReviews, while the ValidatingWebhook and MutatingWebhook make decisions based on **AdmissionReviews**. We have incorporated OPA through the authorization module and the MutatingWebhook in our implementation. So for our requests to be authorized, they need to follow the route as shown in Figure 5.1.

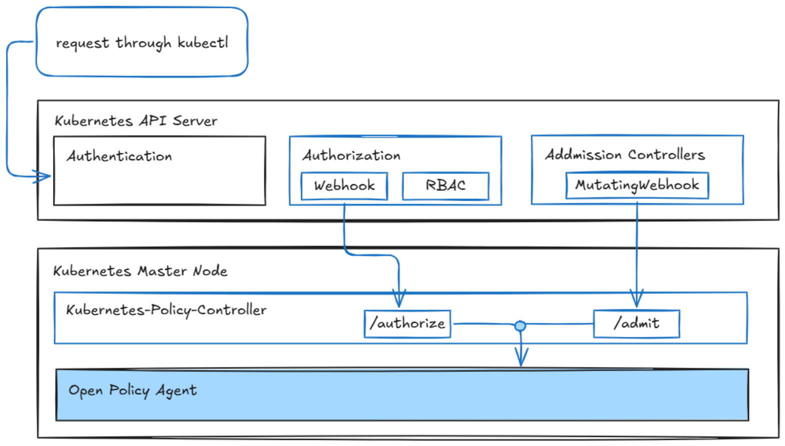

Figure 5.1. OPA deployment pattern

The request must be authenticated beforehand. After that, go through the authorization Webhook, which may either reject the request or notify the next authorization module, specifically RBAC in this instance. Subsequently, RBAC goes into effect, and if it fails to authorize the request, it is immediately rejected. If our request involves modifications to resources, such as creation, updating, or deletion, then Admission Controllers are invoked. For the sake of simplicity, since we have already explained the downsides of deployment of OPA as a webhook, with a cluster service, we are going to skip that part and have only RBAC and OPA through means of the Admission Controller. The module known as `Kubernetes-Policy-Controller` is, in fact, the **Gatekeeper** [38], a policy controller for Kubernetes that enables the conversion of Kubernetes SubjectAccessReviews and AdmissionReviews objects into OPA queries, given that OPA lacks a REST interface implementation with Kubernetes. It provides improved functionalities, all contained within a single, optimized package. Moreover, Gatekeeper supports native Kubernetes Custom Resource Definitions (CRDs) for the instantiation and enhancement of the policy library.

Although Gatekeeper is mostly recognized as an Admission Controller tool, it may effectively enable fine-grained authorization and validation when used wisely in carefully constructed scenarios. Consequently, we deem it appropriate to provide this here, not only in the context of authorization mechanisms but also to expand our perspective by examining real-life scenarios where many authorization paradigms are interconnected and the admission control layer is established as well, reviewing in full the access control framework.

With all this being said, the only preconfiguration that we need to carry out before starting our scenario-based analysis is to install Gatekeeper in our already set-up clusters. This time we opted for Kind clusters, since they are lightweight and easy to use without any extra configuration (like CNI in kubeadm, for example). So with the following commands, we are good to go and implement our scenarios.

```
kind create cluster --name opa-scenario #create Kind cluster
kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/\
gatekeeper/master/deploy/gatekeeper.yaml #install gatekeeper
kubectl -n gatekeeper-system get pods
```

```
#check if all gatekeeper pods are running fine
#meaning it was installed correctly
```

And to conclude, we expect to have two important manifests in our scenarios. One is the **policy template** that defines the Rego policy logic, and the other is the **policy constraint** that enforces the policy on specified resources. Then OPA evaluates the policies using the Rego language so that valid requests pass through and invalid ones get rejected. But we will see this last part in more detail in the coming scenarios.

At this point, we have explained our deployment pattern and carried out all the configurations needed, so we can move on to implementing our scenario-based analysis.

5.2.3 Scenario-based analysis

In this section, we will examine one scenario demonstrating the implementation of PBAC with OPA while being close to real-life application of the paradigm and mechanism and highlighting the strengths and limitations of its mechanism and the paradigm itself.

Scenario 1

In this scenario, we are going to implement an advanced Policy-based access control using OPA. We aim to utilize two key features of OPA: first, the capacity to incorporate time-based attributes for context-aware policies, and second, the ability to implement "blacklisting", allowing for the formulation of deny rules. These are two features that are absent in native Kubernetes authorization mechanisms. It may be feasible with native authorization Webhooks, but it would involve significant complexity and challenges in scalability. OPA manages these two aspects efficiently. We will utilize Gatekeeper, even though it is mainly seen as an admission controller; when properly set, it serves as a powerful tool for fine-grained access control. Alongside Gatekeeper, we will implement RBAC to establish the basis of our access control. This demonstrates how we can integrate these two mechanisms and distinct access-control paradigms.

The scenario involves a user named `dev` who has access throughout the week across three designated namespaces: `dev`, `test`, and `prod`. He is permitted to read and write in pods, services, and deployments, except on Fridays, when he loses the ability to `create`, `update`, `patch`, or `delete` resources in the `prod` namespace. We will implement the Role and Rolebindings to grant the user `dev` access to all resources across the three namespaces. And then on top of RBAC, there will be the Gatekeeper that, in case the day is Friday, will actually prevent user `dev` write access in the `prod` namespace. Figure 5.2 illustrates an outline of the scenario.

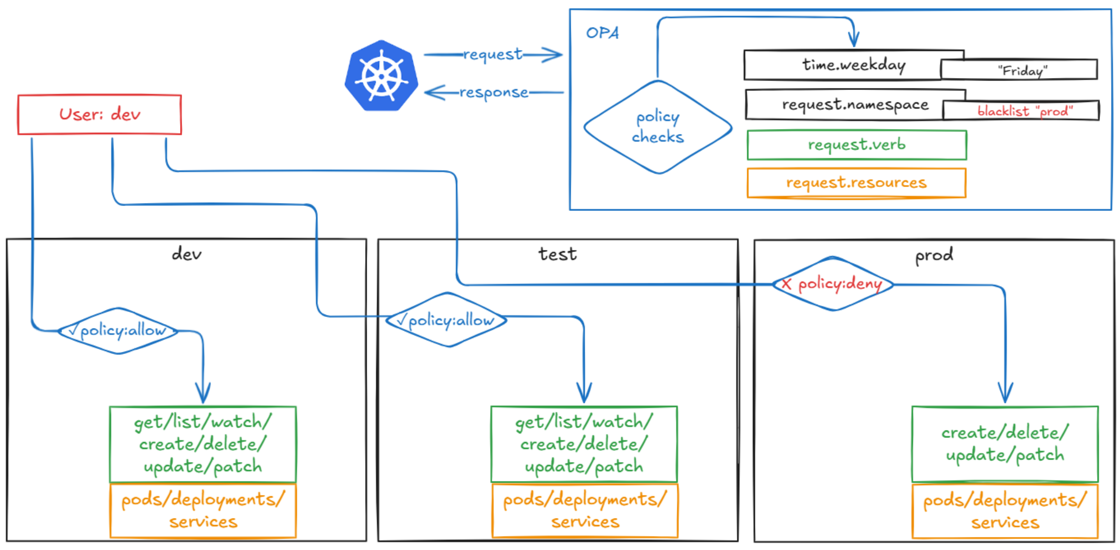
OPA
request
response
policy checks
time.weekday
"Friday"
request.namespace
blacklist "prod"
request.verb
request.resources
User: dev
dev
test
prod
✓policy:allow
✓policy:allow
X policy:deny
get/list/watch/ create/delete/ update/patch
pods/deployments/ services
get/list/watch/ create/delete/ update/patch
pods/deployments/ services
create/delete/ update/patch
pods/deployments/ services

Figure 5.2. PBAC with OPA scenario 1

As outlined in the configuration section, we begin by creating a basic Kind cluster and installing Gatekeeper within it. We take the appropriate precautions to authenticate a user using X.509 certificates, referred to as `dev`, as we have repeatedly done before. We ensure the proper creation of all three namespaces: `dev`, `test`, and `prod`. Subsequently, we set up RBAC for the user `dev`. This will maintain the workflow for all other weekdays, excluding Friday, across all three namespaces, for every applicable verb concerning pods, services, and deployments.

At this point, we must develop a policy template that defines the Rego policy logic. In this case, for the time-based policy. It will function as an additional layer that can obstruct access even when RBAC would typically permit it. Utilizing `time.now_ns()`, we can figure out the current weekday on which the request is being made, with Friday represented as the integer `5`. Subsequently, we verify the user and the corresponding namespace. Upon having a match involving a restricted user, within a restricted namespace, concerning a restricted resource, on a restricted day, we trigger a violation message in Rego. A snippet of the policy template is as follows.

Listing 5.2. OPA policy template snippet for time-based policy

```
rego: |
    package k8stimerestriction
    # This rule triggers a violation when all conditions are met
    violation[{"msg": msg}]{
        #get current day
        now_ns := time.now_ns()
        current_day := time.weekday(now_ns)
        #is current day restricted
        input.parameters.restrictedDays[current_day]
        #is requested namespace restricted
        input.review.object.metadata.namespace==
           input.parameters.restrictedNamespaces
        #is user doing the request up for restriction
        user_annotation :=
           input.review.object.metadata.annotations["requested-by"]
        user_annotation==input.parameters.targetUsers
        #construct the violation message to be displayed
        msg := sprintf("%s Current day: %d, Namespace: %s, User:
           %s", [
            input.parameters.message,
            current_day,
            input.review.object.metadata.namespace,
            user_annotation
        ])
    }
```

We can verify it is created with the following command.

```
kubectl get constrainttemplates
```

Then we need to define the meaning of a restricted user within a restricted namespace concerning a restricted resource on a restricted day. We accomplish this by creating a constraint that, as the name suggests, constrains the policy on specified cases. This acts as a policy enforcement that specifies the restricted user as `dev`, the restricted namespace as `prod`, the restricted resources as `pod`, `service`, and `deployment`, and the restricted day as `Friday`. It also serves the purpose of policy decoupling that we have presented before. A snippet of the policy constraint is as follows.

Listing 5.3. OPA policy constraint for time-based policy

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sTimeRestriction
metadata:
  name: friday-prod-block
spec:
  enforcementAction: dryrun #only for testing. 'deny' in prod
  match:
    kinds:
      # Target pods
      - apiGroups: [""]
        kinds: ["Pod"]
      # Target services
      - apiGroups: [""]
        kinds: ["Service"]
      # Target deployments
      - apiGroups: ["apps"]
        kinds: ["Deployment"]
  parameters:
    restrictedDays: [5] #Friday only (int 5 = Friday)
    restrictedNamespaces: ["prod"] #only restrict prod namespace
    targetUsers: ["dev"] #only apply to dev user
    message: "BLOCKED: Dev user cannot access prod namespace on
       Fridays!"
```

We can verify everything is as it should be with the following command.

```
kubectl get k8stimerestriction friday-prod-block
kubectl describe k8stimerestriction friday-prod-block
```

In order to test out our scenario, we first need to check what day Gatekeeper thinks it is by creating a template that, as a violation message, will output the current day perceived by Gatekeeper. It will also need a constraint to supply the restricted values.

Small note here: if we have the template and the constraint in one file, we might get an error when applying it, saying the CRD does not exist. We should know it is just because it needs a little bit of time for the CRD to be applied, and if the constraint is immediately after in the manifest, it might be that when it is being executed, the CRD is still not fully applied in the cluster. We can just rerun it until everything is created.

At this point we can create a test pod in our cluster. And since we have set the `enforcementAction: dryrun` in the constraint for debug, the pod will be created, but we can see the debug information in the constraint status with the following command.

```
kubectl describe k8stimedebug time-debug
```

We will see here the message with the current day. Let's say that we run this scenario on a Wednesday. So if we try to access the pod in `prod` namespace impersonating user `dev`, we would succeed. We can test for this with the following command.

```
kubectl auth can-i get pods --namespace=prod --as=dev #returns yes
```

But if we were to run this scenario on a Friday, or even create a simulation with a template that has the current day hardcoded as integer `5`, representing Friday, attempting to create a pod in `prod` namespace impersonating user `dev` we would fail. We can test for this with the following command.

```
kubectl auth can-i create pods --namespace=prod --as=dev #returns no
```

This scenario holds significance as it illustrates the complicated details of OPA and its integration with native authorization mechanisms that handle base cases for simplicity. It is worthy to acknowledge that while the fine-grained authorization provided by this implementation is notable, it implies a steep learning curve for those who need to get familiar with OPA and Rego in particular.

Different from the native authorization mechanisms, the scenarios on the open source solutions are able to cover much more ground with less effort. This explains the reason we are presenting only one scenario and concluding here our practical analysis of PBAC and OPA in Kubernetes.

5.2.4 Strengths and limitations

PBAC, implemented through OPA, presents a paradigm shift to unified, context-aware governance across various systems. Its main advantage lies in offering a single, cross-platform framework for authorization and access control, facilitating uniform policy enforcement across Kubernetes, and not only that, but also APIs, microservices, CI/CD pipelines, and cloud infrastructure. Policies are articulated declaratively in Rego, the specialized language of OPA, facilitating highly complex, context-aware decisions that assess intricate links among people, resources, activities, and external data sources. This includes innate support for explicit blacklisting (deny rules), addressing a significant constraint of native RBAC and ABAC in Kubernetes. Moreover, policy modifications are executed dynamically, meaning updates are implemented instantaneously without requiring cluster restarts or causing service disruptions, thus allowing agile security management and ongoing compliance.

However, achieving these capabilities involves considerable complexity and operational burden. A principal drawback is the significant learning curve linked to mastering Rego syntax, comprehending OPA's deployment patterns, and properly integrating it with Kubernetes. Connecting OPA to the Kubernetes API server requires an extra component, such as the Kubernetes Policy Controller (or Gatekeeper), which increases deployment and maintenance complexity. This architectural dependency, along with the necessity to evaluate potentially complex policies for each relevant API request, results in latency overhead as compared to simpler native mechanisms such as RBAC. Meticulous performance optimization and policy formulation are crucial for mitigating this side effect. Again, there is difficulty with efficiently listing and filtering based on user permissions.

PBAC and OPA shine in situations requiring unified, auditable policy enforcement across hybrid or multi-platform infrastructures, such as Kubernetes clusters, cloud services, and on-premises systems. Its advantages are essential for enterprises requiring extremely detailed, context-aware authorization governed by intricate business logic, regulatory compliance, or evolving security demands that are beyond the functionalities of RBAC or ABAC. Thus, OPA is perfectly suited for security-sensitive, large-scale implementations where the commitment to mastering Rego and managing Gatekeeper is justified by the necessity for dynamic, auditable, and uniformly enforced policies that can integrate real-time data and explicit denial rules throughout the entire stack.

5.3 Relationship-Based Access Control (ReBAC)

Relationship-based Access Control (ReBAC) is a promising authorization paradigm. As described in [39], the fundamental concept of ReBAC is that the presence of a sequence of relationships between a subject and a resource defines access. This abstraction can replicate all existing authorization paradigms, including the widely used RBAC and ABAC models. The idea was initially outlined by Carrie Gates in a 2006 paper titled Access Control Requirements for Web 2.0 Security and Privacy [40], with Facebook recognized as a pioneer adopter of this paradigm. Nonetheless, it was not until the release of the Zanzibar paper [41] in 2019 that ReBAC gained popularity beyond applications that were already using graph abstractions for their data. With Broken Access Control currently leading the OWASP Top 10 [2], ReBAC is now regarded as the recommended approach for developing reliable authorization systems [42]. To give a little bit more detail on the paradigm itself, let's start by defining what a relationship is [43]. **Relationships** bind two entities or objects, these being either a **Subject** or a **Resource**, through a **Relation**. The principle of ReBAC authorization systems is that access may be determined by tracing a **chain of relationships**. The core of authorization logic can be simplified to a single question: "Is this user allowed to perform this action on this resource?" We can break this down into its fundamental components: the actor being the subject object, the resource being, as the name suggests, the resource object, and the action being the permission or relation. The true power of ReBAC lies in rephrasing this basic question into a more complex one: "Is there a chain of relationships beginning from this resource via this relation that eventually leads to this subject?" This question is related to graph reachability. General-purpose graph databases optimize for various graph traversals, typically depth-first search and breadth-first search. In contrast, ReBAC systems are specifically designed to efficiently and scalably compute reachability, using all relevant assumptions associated with the domain of authorization. So the final word on relationships is that they have significant power due to their dual nature, serving as both the question and, when considered together, the answer.

Let us briefly describe the **Google Zanzibar** paper that contributed to the superiority of ReBAC [44]. Google Zanzibar is a globally distributed authorization system created internally at Google to solve the significant challenge of handling fine-grained permissions on a global scale. Emerging from necessity as Google's product ecosystem broadened, it addressed a critical fragmentation issue, that of each service having established its own ad hoc authorization protocols, resulting in inconsistencies, redundant engineering efforts, and operational vulnerabilities. Zanzibar consolidated these independent systems into a single platform, facilitating centralized administration of access control policies across numerous services while managing trillions of permission evaluations daily. The development was motivated by the necessity for a system that could uniformly enforce security invariants, minimize developer overhead, and adapt to Google's exponential growth. Zanzibar pioneered an innovative paradigm focused on relationship-based access control (ReBAC). In contrast to RBAC and ABAC, Zanzibar conceptualizes permissions as explicit relationships among entities organized inside a graph-like structure. Authorization decisions depend on navigating these relationships, or "walking the graph". It utilizes distributed technologies such as globally consistent storage (via Spanner), concurrent graph traversal, and cache layers to effectively resolve queries in milliseconds, even over billions of edges. Its architecture guarantees robust external consistency while preserving low latency, which is essential for user-facing services.

So we can say that Zanzibar's approach provides major advantages by fundamentally redefining authorization on a large scale. The centralized logic guarantees consistency and reliability, resolving policy conflicts while enforcing uniform security compliance across all services. The scalability of its distributed architecture efficiently manages millions of requests per second, accommodating Google's worldwide user base without performance decline. Developer efficiency is significantly improved by a straightforward API that delegates complex permission logic, hence accelerating product development cycles. Expressive flexibility arises from its relationship-graph architecture, which intuitively depicts real-world hierarchies, such as organizational structures or nested resources, facilitating complex regulations like inherited permissions. Low latency, attained via optimized caching and concurrent evaluation, ensures sub-10 ms response times for the majority of queries, facilitating uninterrupted user experiences even under significant loads. Collectively, these features create a paradigm shift in secure, scalable access control.

To sum up, even though Zanzibar is closely related to ReBAC, it is not identical. Zanzibar is Google's authorization system, whereas ReBAC is an authorization paradigm that emphasizes relationships between objects to determine access rights. Zanzibar employs ReBAC as its foundational authorization framework, so it can be classified as a ReBAC system; yet, it encompasses additional complexities. It also includes the infrastructure, algorithms, and optimizations that enable its operation at Google's vast scale [39]. But as we will see in the following section, Zanzibar will be the source of inspiration for SpiceDB, the final authorization mechanism we are going to analyze in this thesis.

5.3.1 SpiceDB

SpiceDB is **AuthZed**'s open source, Google Zanzibar-inspired database system for real-time, security-critical application permissions [45]. It is engineered to deliver comparable capabilities to Zanzibar but focuses on accessibility and adaptability for a broader spectrum of use cases. It aligns with Zanzibar's fundamental concepts, comprising relationship-based access control (ReBAC), scalability, performance, and robust consistency.

Let us begin by examining the operational mechanisms of SpiceDB. In SpiceDB, a functional **Permissions System** consists of **Schema**, which outlines the data structure, and **Relationships**, which represent the data itself. A SpiceDB schema establishes the types of objects present, the relations between them, and the permissions derivable from those relations [46]. The top level of a Schema consists of zero or more **Object Type** definitions and **Caveats**.

An Object Type definition serves to define a new type of object. Objects serve as the representation of instances of resources or subjects in SpiceDB. Object Type definitions can be likened to class definitions in object-oriented programming or table definitions in an SQL database.

Caveats are a feature in SpiceDB that enables the conditional definition of relationships, meaning that the relationship is deemed present only if the caveat expression evaluates to true during query execution. This will be crucial later to mix ABAC with ReBAC.

A relation defines how two objects (or an object and a subject) can be related. For example, a reader of a document or a member of a team. Relations are consistently identified with a name (preferably a noun) and one or more allowed object types that may serve as the subjects of that relation. Relations can also "contain" references to other relations/permissions. A worthy note here on wildcards. Relations may utilize wildcards to denote that a grant applies to the entire resource type rather than to a specific resource. This permits public access to a specific subject type.

A permission defines a calculated set of subjects possessing a specific type of permission about the parent object; for example, a user is part of the group authorized to modify a document. Permissions are consistently identified by the name and an expression that establishes the computation of the permissible set of subjects for that permission. Since permissions define a collection of objects capable of performing an action or possessing a particular attribute, they need to be named as verbs or nouns. Permissions support four kinds of operations: union (+), intersection (&), exclusion (-), and arrow (`->`). The first three are quite self-explanatory, while the arrow allows the traversal of the hierarchy of relations and permissions established for an object's subject, referencing a permission or relation relevant to the resultant subject's object.

Listing 5.4. SpiceDB Schema example

```
schema: |-
definition user {}

definition group {
    relation member: user | group#member
}

definition folder{
   relation reader: user
   permission read = reader
}

definition document {
    relation parent_folder: folder
    relation writer: user | group#member
    relation reader: user | group#member

    permission write = writer
    permission read = reader + write + parent_folder->read
}
```

Unlike Zanzibar, which makes no distinction between **relations** and **permissions**, SpiceDB introduces distinct terminology and syntax to categorize them as two different concepts, as we have seen previously. Permissions are best regarded as the "public API" utilized by applications to verify access. Permissions are defined by set semantics; instead, relations are entirely abstract relationships between objects kept in SpiceDB. The API can query them, but it is strongly advised to exclusively call Permissions using the API, as Permissions may be modified to provide backward compatibility in access computation.

One last cool feature of SpiceDB is **Reverse Indices**. As described by Lea Kissner, Zanzibar coauthor, "Reverse-index expand answers the question 'What does this employee have access to?', which most organizations validate as part of meeting those compliance obligations. But, even more critically, organizations use this information to debug access issues and as baseline data to ensure careful data handling." [47]. Zanzibar and SpiceDB both have a Reverse Index Expand API. This API returns a tree structure that might be troublesome for applications to process, particularly when it is preferred to separate permissions logic from application code. Consequently, SpiceDB offers supplementary APIs, such as `LookUpResources` and `LookUpSubjects`, to facilitate the utilization of Reverse Indices without a defined structure. They are intended to address the following questions, respectively: "What are all of the resources this subject can access?" and "What are all of the subjects with access to this resource?" [47].

In conclusion of our introduction to SpiceDB and the ReBAC paradigm it embodies, let us examine the distinguishing features that set SpiceDB apart from other systems [45]. It actually differentiates itself through a suite of integrated capabilities tailored for modern authorization challenges. It offers expressive gRPC and HTTP/JSON APIs that provide granular authorization checks, extensive access listings, and developer tool integrations, optimizing both implementation and debugging processes. SpiceDB fundamentally operates as a distributed parallel graph engine that rigorously follows the architectural concepts established in Google's Zanzibar paper, guaranteeing proven scalability and reliability. The foundation is improved by a configurable consistency model per request, incorporating essential safeguards against edge cases. SpiceDB provides an expressive schema language for policy administration, together with an interactive playground [48] and comprehensive CI/CD interfaces, facilitating the effortless validation and testing of complex authorization frameworks prior to deployment. The pluggable storage layer accommodates many production-grade databases, including in-memory, Spanner, CockroachDB, PostgreSQL, and MySQL, hence offering deployment flexibility across different infrastructure configurations. Ultimately, thorough observability features incorporate Prometheus metrics, pprof profiling, structured logging, and OpenTelemetry tracing, providing exceptional insight into authorization performance and system behavior. Together, these features position SpiceDB as a unified platform for relationship-oriented authorization that balances expressiveness, resilience, and operational efficiency.

5.3.2 Configuration

The most straightforward method for deploying SpiceDB in Kubernetes environments leverages the open source `spicedb-kubeapi-proxy` [49]. This proxy intermediates between Kubernetes clients and the API server, performing authorization by delegating access control decisions to an embedded or remote SpiceDB instance while concurrently filtering API responses. In simpler words, it allows for making any Kubernetes API call (through any command-line tool, like `kubectl`) to the proxy, which will then use SpiceDB to answer access control questions, and it eventually forwards the authorized requests onto the Kubernetes API server that will perform the actions outlined in the authorized request and respond back to the client again through the proxy. Figure 5.3 illustrates an outline of the architecture.

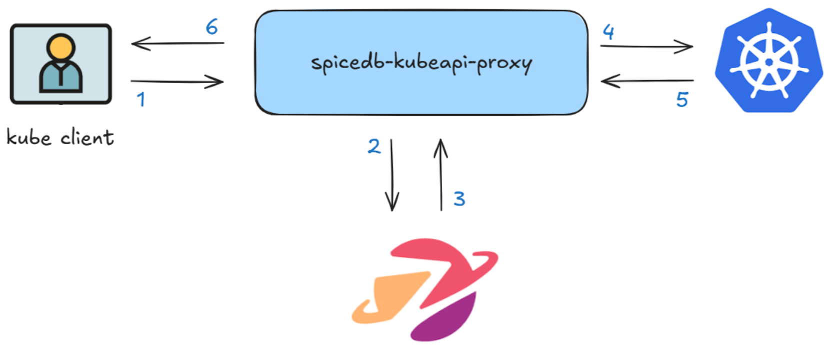

Figure 5.3. Communication schema between spicedb-kubeapi-proxy and Kubernetes

Most importantly, this architecture introduces no hard dependencies; the proxy operates transparently and can be bypassed during SpiceDB outages or debugging without disrupting cluster operations, which is a significant advantage over Webhook-based extensions that deeply integrate with the API server's authorization chain.

Configuration occurs through declarative rules defined as Kubernetes custom resources. Each rule specifies the precise permission to evaluate within SpiceDB's relationship graph and the relationships to update when users modify governed resources. The proxy dynamically loads these rules at runtime, executing queries against SpiceDB without interpreting request semantics or cluster state. This abstraction ensures the proxy remains stateless and efficient, solely translating Kubernetes operations into SpiceDB queries and responses.

A critical innovation is its support for atomic transactions: the proxy orchestrates synchronized writes to both Kubernetes and SpiceDB, guaranteeing consistency between cluster state and relationships. Consequently, standard `kubectl` operations automatically propagate real-time updates to SpiceDB's graph, acting the same way as direct schema manipulations via SpiceDB's native `zed` CLI. This capability enables seamless adoption of relationship-based access control without altering existing Kubernetes workflows.

Deployment in Kubernetes is extremely simple. As described in the README of the repo [49], we can set up a development kind cluster with the proxy running in it with an embedded SpiceDB with the following commands.

```
#the project uses mage to offer several development-related commands
#so run this to get all available commands
brew install mage
mage
#set up the development environment
mage dev:up
#configure your client to talk to the proxy
kubectl --kubeconfig $(pwd)/dev.kubeconfig --context proxy get namespace
```

With this set of commands, we are all set to further explore SpiceDB within our Kubernetes cluster. At the end, we utilized this deployment pattern due to its simplicity and adequate fit for our scenario-based analysis.

Just for the sake of completeness, we are going to describe briefly another deployment pattern available for SpiceDB in Kubernetes. The `SpiceDB Operator` [50] is a Kubernetes Operator that oversees the installation and lifecycle management of SpiceDB clusters. Considering the fact that SpiceDB is engineered with both cloud-native and Kubernetes-native principles, it makes `SpiceDB Operator` the optimal method for deploying SpiceDB in a production environment. It is worth noting here that all managed AuthZed products essentially leverage the `SpiceDB Operator`.

Upon installation of the `SpiceDB Operator`, Kubernetes clusters acquire a new resource named
`SpiceDBCluster`. These clusters, created with the `SpiceDBCluster` resource, possess certain features, like centralized administration for cluster configurations, automated updates of SpiceDB, and automated datastore migrations with zero downtime during SpiceDB upgrades. For further details, we recommend checking out the official GitHub page of AuthZed [51].

As a final note, we want to emphasize how well documented SpiceDB is. There are up-to-date guides, examples, and video tutorials. And also a cool feature of Playground [48] where you can learn and test out how to build Schemas and learn your way around SpiceDB, all supported with tutorials.

At this point, we have explained the deployment pattern and carried out all the configurations needed, so we can move on to implementing our scenario-based analysis.

5.3.3 Scenario-based analysis

In this section, we will examine one scenario demonstrating the implementation of ReBAC with SpiceDB while being close to a real-life application of the paradigm and mechanism, and highlighting the strengths and limitations of its mechanism and the paradigm itself.

Scenario 1

For this final scenario in our analysis, we are going for a more interactive approach in order to highlight the responsiveness of SpiceDB.

We start by executing the following command.

```
kubectl get namespaces
```

It shows that we have one active namespace named `spicedb-kubeapi-proxy`, and at the same time, we can check the SpiceDB instance logs, and we can see several `LookupResources` calls. They literally show that the proxy is asking SpiceDB which namespaces we, as the user, can view. We can check that the information is consistent on SpiceDB itself with the `zed` CLI.

Executing the following command:

```
zed relationship read namespace
```

will show all the relationships existing for the namespace object. At this moment, it will show `namespace:spicedb-kubeapi-proxy#viewer@user:rakis` since it is the only one existing in the bootstrap schema and relationships that come with the proxy.

In case we were to create a new namespace in our cluster, the proxy will automatically create two relationships, one with us, the user, as the creator of the namespace, looking like:
`namespace:name-of-new-namespace#creator@user:rakis`. And another one with a cluster object, useful if we want to grant some permissions at the cluster level later on. Meanwhile, we can check SpiceDB logs and see that several read/write operations were indeed executed on it, showing how the proxy is able to write to both SpiceDB and Kubernetes in one single transaction.

We can try and create a pod via `kubectl` as we have normally done throughout our scenarios. At the same time that it is executed, a new relationship will be added, connecting us, the user, as the creator of the pod. In this moment, if we try to delete ourselves as the creator of the pod with the following command:

```
zed relationship delete pod:namespace-of-pod/name#creator@user:rakis
```

It would immediately block us from getting the pod via `kubectl`, even if the pod still exists in our cluster. Even more interesting, if we were to run the `kubectl get pod` with the `-watch` tag before deleting the relationship, we would see exactly the pod disappearing from the watchlist the moment we run the delete command on SpiceDB.

We can also make changes to the schema in real time. We have the new schema like the following listing.

Listing 5.5. SpiceDB scenario 1 first schema

```
schema: |-
  definition cluster {}

  definition user {}

  definition namespace {
    relation cluster: cluster
    relation creator: user
    relation viewer: user
    permission view = viewer + creator
  }

  definition team {
    relation member: user
  }

  definition pod {
    relation namespace: namespace
    relation creator: user
    relation viewer: user | team#member
    permission edit = creator
    permission view = viewer + creator
    }
```

We are defining a team object consisting of a member related to a user. In the pod object, we are setting as the viewer of the pod, either the user or a member of a team that has permission to view the pod. Meaning the team has permission to view the pod, and as a result, the member of the team also inherits this permission. We can apply the new schema with the following command.

```
zed schema write name-of-schema.zed
```

We can continue by creating a relationship so that a team foo is a viewer of the pod with the following command.

```
zed relationship create pod:namespace-of-pod/name#viewer@team:foo#member
```

And then add ourselves as a member of that team with the next command.

```
zed relationship create team:foo#member@user:rakis
```

As soon as we have added ourselves to the team that has access to the said pod, we instantly regain access to the pod. Even though we are not a direct viewer, because of the fact of being a member of the team that is a viewer, by the transitive property of the graph, we inherit the permission as well.

We can establish the organization notion with SpiceDB, even though it is not a native notion to Kubernetes. We do this simply by declaring it as a separate object in our schema, with a member related to a user and a permission to view granted to a member. Let's say that we want any member of the organization to be able to view the pod. So we modify the view permission in the pod object like `permission view = viewer + creator + organization->member`. That basically forces the decision process to walk every organization through the view permissions. In order to try this out, we first remove ourselves from the team in order to not have access anymore to the pod. We do this with the following command.

```
zed relationship delete team:foo#member@user:rakis
```

We update the schema to the new one by applying a modified file as we did before, and then grant the organization access to the pod by creating a relationship and adding ourselves as a member to the organization with the following commands.

```
zed relationship create \
pod:namespace-of-pod/name#organization@organization:org
zed relationship create organization:org#member@user:rakis
```

As soon as we have added ourselves to the organization, we regain access to the pod.

Last, we will see how to integrate ABAC within ReBAC. We define a new schema like in the following listing.

Listing 5.6. SpiceDB scenario 1 second schema

```
schema: |-
    definition user {}
    caveat somepolicy (current_day string){
        current_day == "Friday"
    }
    definition pod {
        relation creator: user
        relation viewer: user with somepolicy
        permission edit = creator
        permission view = viewer + creator
    }
```

We are defining the viewer of the pod as a user with some policy, and then defining some caveat that processes this policy. If you then execute the view permission and walk to the viewer, and you have a relationship that exists in the system like `pod:namespace-of-pod/name#viewer@user:username(somepolicy)`. The trick is that this relationship will only be considered existing if the conditions are met. It might not be as powerful as a real policy, but you get a good tradeoff of the scalability and performance of ReBAC and the powerfulness of ABAC in such a simple implementation.

Figure 5.4 illustrates an outline of the scenario, and moreover, it serves as a visualization of the schema in SpiceDB.

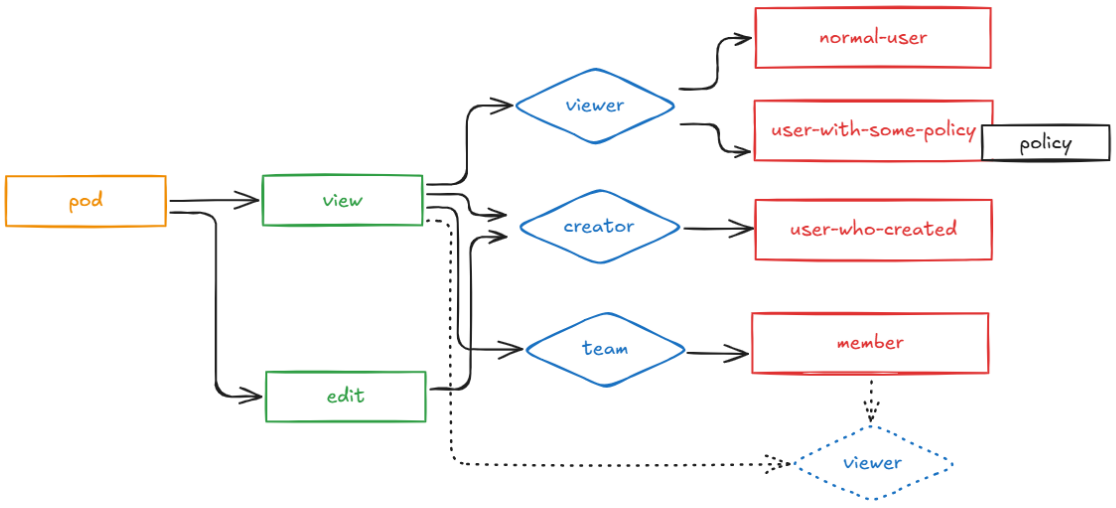

Figure 5.4. ReBAC with SpiceDB scenario 1

This scenario concludes our practical analysis of ReBAC and SpiceDB in Kubernetes.

5.3.4 Strengths and limitations

ReBAC, implemented through SpiceDB, presents a robust paradigm focused on modeling access via explicit relationships among entities, these being users, groups, resources, or more hierarchical organizations, inside a graph architecture. Its principal advantage resides in allowing exceptionally fine-grained permissions and complex authorization decisions that adjust dynamically according to the presence and particulars of relationships. SpiceDB serves as the core engine, offering outstanding scalability and superior performance, effectively navigating dense relationship graphs to produce low-latency authorization decisions in demanding environments. The authorization schema and relational data are dynamically managed by API requests from any authorized client, allowing for instantaneous modifications without service disruption. Importantly, ReBAC and SpiceDB inherently enable efficient listing and filtering actions, addressing a significant issue found in native RBAC, ABAC, Webhook mechanisms, and even basic OPA integrations.

However, the implementation and management of ReBAC and SpiceDB present specific challenges. Although SpiceDB runs through a proxy divided from Kubernetes, thus avoiding a hard technical dependency within the core Kubernetes control plane, it still introduces considerable operational complexity to the infrastructure, requiring the deployment, scaling, monitoring, and securing of this additional essential service. Moreover, mastering the notion of graph-based permissions, building an appropriate schema, and integrating SpiceDB requires a significant learning curve, demanding expertise beyond traditional role- or attribute-based models.

ReBAC, powered by SpiceDB, shines in contexts demanding complex, relationship-based permission models that reflect real-world organizational frameworks or resource hierarchies. Its qualities are essential for large-scale, performance-sensitive applications requiring ultra-fast permission checks across extensive, interconnected data schemas, where fine-grained control and efficient listing and filtering capabilities are vital. Thus, SpiceDB is optimally designed for entities developing access control frameworks focused on dynamic resource relationships and inheritance, especially when the operational demands of managing SpiceDB and the effort to master its schema design are justified by the need for exceptional granularity, performance, and dynamic manageability.

Chapter 6

Comparative Analysis

6.1 Evaluation matrix

In the preceding chapters, we have thoroughly examined the access control framework in Kubernetes and an extensive selection of authorization mechanisms, covering both the native alternatives within Kubernetes and those provided by open source solutions. We explored these mechanisms both theoretically and in terms of the potential of their paradigms, as well as conducted an in-depth analysis of their practical implementation through scenario-based assessments, providing a thorough understanding of their implementation, configuration, strengths, and limitations. The work completed thus far provides an excellent foundation for a final comparative review of all the alternatives we have been evaluating. To do this, we have opted to employ a four-dimensional evaluation matrix that spans complexity, granularity, scalability, and performance, enabling us to assess their operational viability, access control precision, growth potential, and efficiency in a more methodical way.

Complexity

Assesses the burdens associated with implementation and maintenance, including initial configuration challenges and policy lifecycle management with updating, version control, and auditing. It also assesses conceptual clarity, dependence on other components, and the need for specialized skills. Reduced complexity favors rapid adoption and minimizes operational cost, although it often trades off against advanced functionalities.

Granularity

Evaluates the precision of access control, determining whether mechanisms allow for only basic allow/deny rules or more complex context-aware policies that incorporate dynamic attributes, relationships, or external data. Increased granularity enables finer security boundaries and compliance adherence, although it consequently complicates policy management.

Scalability

Examines how each mechanism accommodates growth in users, resources, policies, or clusters without degradation. Key variables include architectural limitations such as role explosion and constraints on policy file size or structure, the influence of policy complexity on decision latency, and the overhead associated with propagating policy state.

Performance

Measures the speed of authorization decisions, considering the architectural perspective, possible network latency in case of external services involved, and the computational overhead of policy evaluation. Performance directly impacts user experience and system responsiveness, particularly for listing and filtering operations or high-traffic APIs.

This matrix is optimal for cross-comparison among all solutions, as it blends technical ability with operational sustainability. It exposes inherent trade-offs, such as cases where solutions that excel in granularity may experience drawbacks in complexity or performance, or highly scalable systems may sacrifice fine-grained control. By methodically assessing all the mechanisms across these interconnected dimensions, we identify an optimal alignment between organizational needs and technical solutions. With this being said, we can move on to conducting our final cross-comparison analysis.

6.2 Cross-comparison of all solutions

As anticipated in the previous section, we are going to cross-evaluate all the mechanisms and their respective paradigms through each dimension of the evaluation matrix. Without much further ado, we start with the first dimension being **complexity**. RBAC demonstrates the least complexity among all mechanisms. The role-binding model is conceptually simple, requires no external services or cluster restarts for updates, and leverages native Kubernetes objects controlled using regular kubectl commands or CI/CD pipelines. Moreover, Kubernetes-native event logs enable the auditing process. Instead, for ABAC, we notice a significantly elevated complexity because its static policy file demands API server restarts for modifications, limiting version control and dynamic policy administration. The JSON-based policy syntax, although easily understandable and flexible, is prone to errors and lacks robust testing tools. Moving on to Webhook, it imposes significant operational complexity by requiring a highly available external authorization service. Securing the Webhook with TLS, maintaining its lifecycle, and guaranteeing resilience against failures require specialized development, operational, and security-related expertise. Bridging to the open source solutions, OPA, representing the PBAC paradigm, imposes a steep learning curve mainly because of Rego, a specific declarative language that requires a certain level of expertise. Although policies are managed as version-controlled code, the deployment and synchronization of OPA, frequently through Gatekeeper, introduces additional infrastructure management challenges. Lastly, SpiceDB for the ReBAC paradigm presents the most initial complexity, demanding a solid understanding of graph-based permissions (Zanzibar model) and schema architecture. Instead, in terms of operating and scaling SpiceDB as a crucial external service, even though it requires substantial operational investment, its dynamic API-driven management facilitates post-deployment maintenance slightly better than OPA.

Moving on to the second dimension in our evaluation matrix, **granularity**. RBAC provides the lowest granularity, limited by its rigid role-permission bindings. It is unable to enforce deny rules, context-aware rules, or more detailed-driven rules or support complex settings. Even though it is worth noticing that it is able to bootstrap and aggregate roles and prevent privilege escalation. So we can confirm that RBAC lays a good basis for access control, even though of a low granularity; however, this can be enhanced when combined with other mechanisms in a chain of authorization. Moving on to ABAC, we notice an enhanced granularity by considering attributes, these being of user, resource, or environment, but it does not provide inherent support for logical relationships among entities, like hierarchy or interaction with other data. Last on the native mechanisms, Webhook provides high granularity by assigning decision-making to custom logic, supporting dynamic context and deny rules, but imposing complexity on the external service development. Moving on to the open source solutions, OPA for PBAC provides remarkable granularity, facilitating context-aware policies through the expressive capabilities of Rego. It can assess complex settings, include other external data, implement deny rules, and also modify requests, since it also acts as an admission controller. Lastly, SpiceDB for ReBAC achieves maximum granularity and contextual accuracy by representing permissions as explicit relationships inside a graph. It naturally resolves inherited permissions, enhances the ReBAC paradigm by also incorporating ABAC, and facilitates efficient list and filter operations not possible by the other mechanisms.

Now we consider our third dimension in our evaluation matrix, **scalability**. RBAC suffers from role explosion in large environments, where numerous roles and bindings degrade API server performance and complicate auditing processes. Scaling requires assertive role consolidation, possibly leading to over-privilege. Just as well, ABAC encounters limitations on policy file sizes and slow and static API server startup times with large policies. Scaling requires file partitioning or migration, which is not feasible for now. Instead, the scalability of a Webhook is wholly dependent upon the architecture of the external service. Poorly constructed services create bottlenecks, so achieving high availability and low latency requires significant expertise in distributed systems. Worthy to mention here for ABAC and Webhook, since they require carrying out configurations with accessibility in the control-plane node, their usage is limited to self-managed Kubernetes clusters and poses an issue if we want to scale out with cloud provider clusters instead. Moving on to OPA for PBAC, it has good horizontal scalability through sidecars or remote services. Policy bundles are distributed efficiently, while Rego's indexing capabilities manage complex rules. Nonetheless, large datasets imported into OPA may affect memory and performance. And last, SpiceDB for ReBAC is designed for extensive scalability, utilizing the proven architecture of Google Zanzibar. It effectively navigates billion-edge permission graphs with distributed caching, replication, and optimized storage solutions. Relationship updates propagate dynamically without demanding global recomputation, making it an excellent choice in scalability terms.

Last in our evaluation matrix is **performance**. RBAC provides native speed, as decisions are made through in-memory lookups within the API server, making it optimal for high-throughput clusters. Alongside, ABAC exhibits moderate speed due to its native nature, but suffers from startup disruptions when modifying the policy and lags when dealing with an extensive number of policies. Moving on to Webhook, it introduces considerable latency due to network round-trip time and external processing delay. The performance is reliant upon the speed and closeness of the Webhook service. Next, OPA for PBAC also introduces modest latency. Local deployments with sidecar mode reduce network hops; however, complex Rego rules or large data imports can hinder decision-making times. Alternatives to alleviate this issue are bundling and caching. Lastly, SpiceDB for ReBAC achieves superior performance at scale via improved graph traversal, enduring connection pools, and local caching mechanisms. Although network calls to SpiceDB may introduce latency compared to native RBAC, its architecture guarantees relatively fast replies for the majority of queries, greatly surpassing generic Webhooks and OPA. The filtering and listing operations are also greatly optimized, something that any of the other mechanisms fail to achieve.

6.3 Final takeaways

The comparative analysis of all the considered authorization mechanisms across the four-dimensional evaluation matrix reveals that no singular authorization mechanism excels universally in complexity, granularity, scalability, and performance. The ideal choice is context-dependent, requiring alignment with an organization's particular operational needs, security goals, and infrastructure maturity. Thus, strategic adoption depends on prioritizing dimensions that embody core organizational limitations and objectives.

In settings where operational simplicity and low-latency performance are critical, Role-Based Access Control (RBAC) is the optimal option. This mechanism provides exceptional efficiency for simple permission models, especially in smaller Kubernetes clusters or static operational environments with clearly defined team hierarchies. The minimal configuration overhead of RBAC, its direct integration with Kubernetes object management, and the lack of external dependencies render it optimal when rapid deployment and administrative efficiency take precedence over the need for extensive contextual controls. The in-memory decision engine guarantees minimal performance impact; nonetheless, users must recognize its limitations in dynamic or complex situations, where rigid role-permission bindings may lead to role explosion or overly permissive access.

Organizations seeking a compromise between policy flexibility and infrastructural manageability should consider either ABAC or Webhook authorization. ABAC is suitable for scenarios requiring basic attribute-based decisions, such as environment-specific resource access, without the need for external service dependencies. Its viability depends entirely on the acceptance of operational limitations, such as required API server restarts for policy modifications and the difficulties associated with auditing static policy files. On the other hand, Webhook authorization is advantageous when extensive customization is necessary. This methodology accommodates complicated logic via external services, rendering it appropriate for businesses with advanced DevOps teams adept at sustaining highly accessible, secure authorization endpoints. This flexibility requires considerable investment in the design, security, and monitoring of the Webhook service to reduce latency and eliminate single points of failure.

In scenarios where advanced governance, cross-platform consistency, and auditable compliance are essential, PBAC implemented via OPA becomes a completely reasonable option. This paradigm is particularly effective in regulated sectors, multi-cluster Kubernetes environments, or integrated CI/CD pipelines that require context-aware policies documented as version-controlled artifacts. OPA's Rego language allows for the formulation of expressive rules that include real-time resource attributes, external data sources, and explicit denial capabilities, features that were not achievable with native Kubernetes mechanisms. Despite the steep learning curve linked to Rego and the operational demands of implementing Gatekeeper or other deployment patterns, the investment produces considerable benefits in unified policy enforcement across hybrid settings and dynamic policy updates without service disruptions.

Finally, for systems requiring authorization logic to reflect complex resource relationships or organizational hierarchies at a vast scale, ReBAC, driven by SpiceDB, represents the state of the art. This design is exceptional for extensive multi-tenant SaaS platforms, collaborative document ecosystems, social networks, or any field where permissions originate from complex graph-based relationships. SpiceDB's Zanzibar-inspired design provides fast authorization decisions across billion-edge graphs while natively supporting efficient listing and filtering, a feature lacking in the other evaluated mechanisms. Organizations should adopt this paradigm when relationship-driven granularity, horizontal scalability, and filtering performance are critical requirements, regardless of the substantial initial investment in schema design and SpiceDB-powered cluster management.

To sum up, the authorization landscape presents intrinsic trade-offs: simplicity vs. expressiveness, performance vs. context-awareness, and native integration vs. cross-platform consistency. Native mechanisms effectively address conventional workloads, while advanced paradigms enable organizations to implement sophisticated security and compliance requirements inside dynamic cloud-native environments. The decision route thus reflects the shift from addressing urgent operational requirements to designing for large-scale strategic governance.

We hereby conclude our in-depth comparative analysis of Kubernetes authorization mechanisms for fine-grained access control.

Chapter 7

Conclusions

This thesis addresses the major challenge of fine-grained access control in Kubernetes, motivated by the security demands of cloud-native multi-tenant environments. Through a comprehensive analysis of native and open source authorization mechanisms, this work delivers a comparative framework to guide practitioners in balancing security, scalability, and operational efficiency.

Inherent trade-offs are revealed for native methods such as RBAC, ABAC, and Webhook. RBAC excels in simplicity and low-latency settings but lacks contextual granularity because of its box-shaped roles. ABAC provides attribute-based flexibility but is constrained by high operational costs resulting from static policy management. Webhook enables customized logic but introduces latency and infrastructure dependencies, thus limiting its applicability. Collectively, these solutions prove insufficient for dynamic settings requiring real-time adaptability.

Open source alternatives overcome these constraints through innovative paradigms. OPA for PBAC allows the formulation of expressive, context-sensitive policies through Rego, ensuring cross-platform uniformity and explicit deny rules, at the cost of a steeper learning curve. SpiceDB for ReBAC achieves exceptional granularity via graph-based permissions, effectively addressing complex access scenarios and optimizing listing and filtering operations; however, the initial schema design demands expertise. Both solutions demonstrate that improved functionality necessitates greater architectural investment.

The four-dimensional assessment, in complexity, granularity, scalability, and performance, confirms the absence of a universal solution. RBAC is ideal for latency-sensitive situations and basic access control requirements. ABAC and Webhook support moderate customization needs. OPA is efficient in hybrid infrastructure governance. SpiceDB excels in handling relationship-intensive workloads. The practitioner's guide consolidates these insights, enabling mechanism selection that aligns with organizational needs.

Finally, this thesis has contributed a taxonomy of access control paradigms, clarifying authorization trade-offs, empirical benchmarks for deployment decisions, and a field-tested practitioner's guide. As future research work, this analysis could be extended to integrate real-time threat response and automated validation tools. These advancements will further strengthen the security-agility balance in cloud-native ecosystems.

Bibliography

[1] “What is kubernetes?” https://cloud.google.com/learn/what-is-kubernetes?hl=en, accessed: 2025-06-15.

[2] “Owasp top 10:2021,” https://owasp.org/Top10/, accessed: 2025-02-05.

[3] “Owasp api security top 10,” https://owasp.org/API-Security/editions/2023/en/0x11-t10/, accessed: 2025-02-05.

[4] “Kubernetes documentation,” https://kubernetes.io/docs/home, accessed: 2025-02-01.

[5] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at google with borg,” in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.

[6] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, “Omega: flexible, scalable schedulers for large compute clusters,” in *SIGOPS European Conference on Computer Systems (EuroSys)*, 2013, pp. 351–364.

[7] “The history of kubernetes on a timeline,” https://blog.risingstack.com/the-history-of-kubernetes/, accessed: 2025-06-15.

[8] “Overview - historical context for kubernetes,” https://kubernetes.io/docs/concepts/overview/#going-back-in-time, accessed: 2025-06-15.

[9] “Overview - why you need kubernetes and what it can do,” https://kubernetes.io/docs/concepts/overview/#why-you-need-kubernetes-and-what-can-it-do, accessed: 2025-06-15.

[10] “Cluster architecture,” https://kubernetes.io/docs/concepts/architecture/, accessed: 2025-06-15.

[11] “Understanding etcd in kubernetes: A beginner’s guide,” https://blog.kubesimplify.com/understanding-etcd-in-kubernetes-a-beginners-guide, accessed: 2025-06-15.

[12] “Objects in kubernetes,” https://kubernetes.io/docs/concepts/overview/working-with-objects/, accessed: 2025-06-15.

[13] “Controlling access to the kubernetes api,” https://kubernetes.io/docs/concepts/security/controlling-access/, accessed: 2025-02-01.

[14] “Authenticating,” https://kubernetes.io/docs/reference/access-authn-authz/authentication/, accessed: 2025-04-03.

[15] “Authorization,” https://kubernetes.io/docs/reference/access-authn-authz/authorization/, accessed: 2025-04-03.

[16] “Admission control in kubernetes,” https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/, accessed: 2025-04-03.

[17] “Api access control,” https://kubernetes.io/docs/reference/access-authn-authz/, accessed: 2025-02-01.

[18] "Creating a cluster with kubeadm," https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/, accessed: 2025-02-03.

[19] "Kind official documentation," https://kind.sigs.k8s.io/, accessed: 2025-02-03.

[20] "Generate certificates manually," https://kubernetes.io/docs/tasks/administer-cluster/certificates/, accessed: 2025-02-05.

[21] "Service accounts," https://kubernetes.io/docs/concepts/security/service-accounts/, accessed: 2025-02-03.

[22] "What is attribute-based access control (abac)?" https://www.okta.com/blog/2020/09/attribute-based-access-control-abac/, accessed: 2025-02-15.

[23] "Using abac authorization," https://kubernetes.io/docs/reference/access-authn-authz/abac/, accessed: 2025-06-13.

[24] "Abac authorizer in kubernetes repo," https://github.com/kubernetes/kubernetes/tree/v1.33.0/pkg/auth/authorizer/abac, accessed: 2025-02-15.

[25] "Subjectaccessreview," https://kubernetes.io/docs/reference/kubernetes-api/authorization-resources/subject-access-review-v1/, accessed: 2025-02-25.

[26] "Webhook mode," https://kubernetes.io/docs/reference/access-authn-authz/webhook/, accessed: 2025-03-25.

[27] "Kubernetes beyond rbac - make your own authorization via webhook," https://mstryoda.medium.com/kubernetes-beyond-rbac-make-your-own-authorization-via-webhook-6b901196591b, accessed: 2025-03-25.

[28] "The kubernetes authorization webhook," https://www.styra.com/blog/kubernetes-authorization-webhook/, accessed: 2025-03-25.

[29] "What is policy based access control (pbac)?" https://www.nextlabs.com/products/cloudaz-policy-platform/what-is-policy-based-access-control-pbac/#:~:text=Policy%2Dbased%20access%20control%20(PBAC)%20also%20known%20as,permissions%2C%20roles%2C%20groups%2C%20or%20user%20identities%20alone.&text=This%20central%20management%20makes%20it%20easier%20to,policies%20and%20ensure%20consistency%20across%20an%20organization., accessed: 2025-06-03.

[30] "Policy-based access control (pbac): A comprehensive overview," https://permify.co/post/policy-based-access-control-pbac/#what-is-policy-based-access-control-pbac, accessed: 2025-06-01.

[31] "Open policy agent docs - introduction," https://www.openpolicyagent.org/docs, accessed: 2025-06-01.

[32] "Open policy agent docs - philosophy," https://www.openpolicyagent.org/docs/philosophy, accessed: 2025-06-03.

[33] "Implementing policies with opa — example use cases," https://medium.com/@chathuragunasekera/implementing-policies-with-opa-example-use-cases-6f8f850cdec4, accessed: 2025-06-01.

[34] "Kubernetes authorization via open policy agent," https://itnext.io/kubernetes-authorization-via-open-policy-agent-a9455d9d5ceb, accessed: 2025-06-01.

[35] "Opa rego playground," https://play.openpolicyagent.org/, accessed: 2025-06-03.

[36] "Styra docs - deploying opa on kubernetes," https://docs.styra.com/opa/deploy/k8s, accessed: 2025-06-03.

[37] "Kubernetes authorization webhook using opa," https://github.com/open-policy-agent/contrib/tree/main/k8s_authorization, accessed: 2025-06-03.

[38] "Gatekeeper," https://github.com/open-policy-agent/gatekeeper, accessed: 2025-06-05.

[39] "Spicedb documentation - google zanzibar," https://authzed.com/docs/spicedb/concepts/zanzibar, accessed: 2025-05-01.

[40] C. Gates, "Access control requirements for web 2.0 security and privacy," in *Workshop on Web 2.0 Security and Privacy*, 2007.

[41] R. Pang, R. Caceres, M. Burrows, Z. Chen, P. Dave, N. Germer, A. Golynski, K. Graney, N. Kang, L. Kissner, J. L. Korn, A. Parmar, C. D. Richards, and M. Wang, "Zanzibar: Google's consistent, global authorization system," in *2019 USENIX Annual Technical Conference (USENIX ATC '19)*, 2019.

[42] "Owasp cheat sheet series - authorization cheat sheet," https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html#prefer-attribute-and-relationship-based-access-control-over-rbac, accessed: 2025-06-15.

[43] "Spicedb documentation - relationships," https://authzed.com/docs/spicedb/concepts/relationships, accessed: 2025-05-01.

[44] "Understanding google zanzibar: A comprehensive overview," https://authzed.com/blog/what-is-google-zanzibar, accessed: 2025-05-01.

[45] "Spicedb documentation," https://authzed.com/docs/spicedb/getting-started/discovering-spicedb, accessed: 2025-05-01.

[46] "Spicedb documentation - schema language," https://authzed.com/docs/spicedb/concepts/schema, accessed: 2025-05-01.

[47] "Spicedb documentation - frequently asked questions," https://authzed.com/docs/spicedb/getting-started/faq, accessed: 2025-05-01.

[48] "Spicedb playground," https://play.authzed.com/schema, accessed: 2025-05-10.

[49] "spicedb-kubeapi-proxy repo," https://github.com/authzed/spicedb-kubeapi-proxy, accessed: 2025-05-05.

[50] "Spicedb operator repo," https://github.com/authzed/spicedb-operator, accessed: 2025-05-05.

[51] "Authzed official github page," https://github.com/authzed, accessed: 2025-05-10.