Kubeflow Pipeline Multi-User Design

**Authors:** Chen Sun (Google), Yannis Zarkadas (Arrikto), Yuan Gong (Google)

**Contributor**: Yang Pan (Google), Elias Katsakioris (Arrikto)

**Status:** **Open for comments**

**Last Modified**: 11/14/2019

# Overview

As of today, Kubeflow Pipeline only supports single user scenarios where users share all pipelines, runs, experiments, etc. Different teams in the same organization, who commonly share the same cluster, are able to access the resources from one another.

Access control over the pipeline related resources is a highly demanded feature. This design focuses on offering multi-user support in the Kubeflow Pipeline without running multiple deployments.

Note that the isolation support in Kubeflow Pipeline doesn’t provide any hard security guarantees against malicious attempts by users to infiltrate other user’s profiles.

# TL;DR

* To enforce access control, we need some kind of isolation primitive. We use namespaces to isolate resources.
* Authentication:
  + User: http header from the Istio Gateway
  + Other Kubeflow Service: ServiceAccountToken with KFPipelines service as audience
* Authorization:
  + Map Pipelines API Server endpoints to RBAC rules
  + Define Roles, RoleBindings using K8s RBAC
  + Use SubjectAccessReview to check authorization for a user

# Design

## Authentication

At high level, there are four paths a user can access a KFP instance

* **KFP UI**. KFP UI provides an interface to create/get non-K8s resources, such as experiments, runs, etc, as well as K8s resources such as tensorboards, viewers.
* **KFP SDK**. Similar to KFP UI, users can access the same set of resources through SDK.
* **kubectl**. It provides a backdoor for users to inspect into the cluster for the K8s native resources, such as pods, services, or K8s customer resources like argo workflow.
* **In cluster access**. For example, calling Kubeflow Pipeline API through a in-cluster Jupyter notebook.

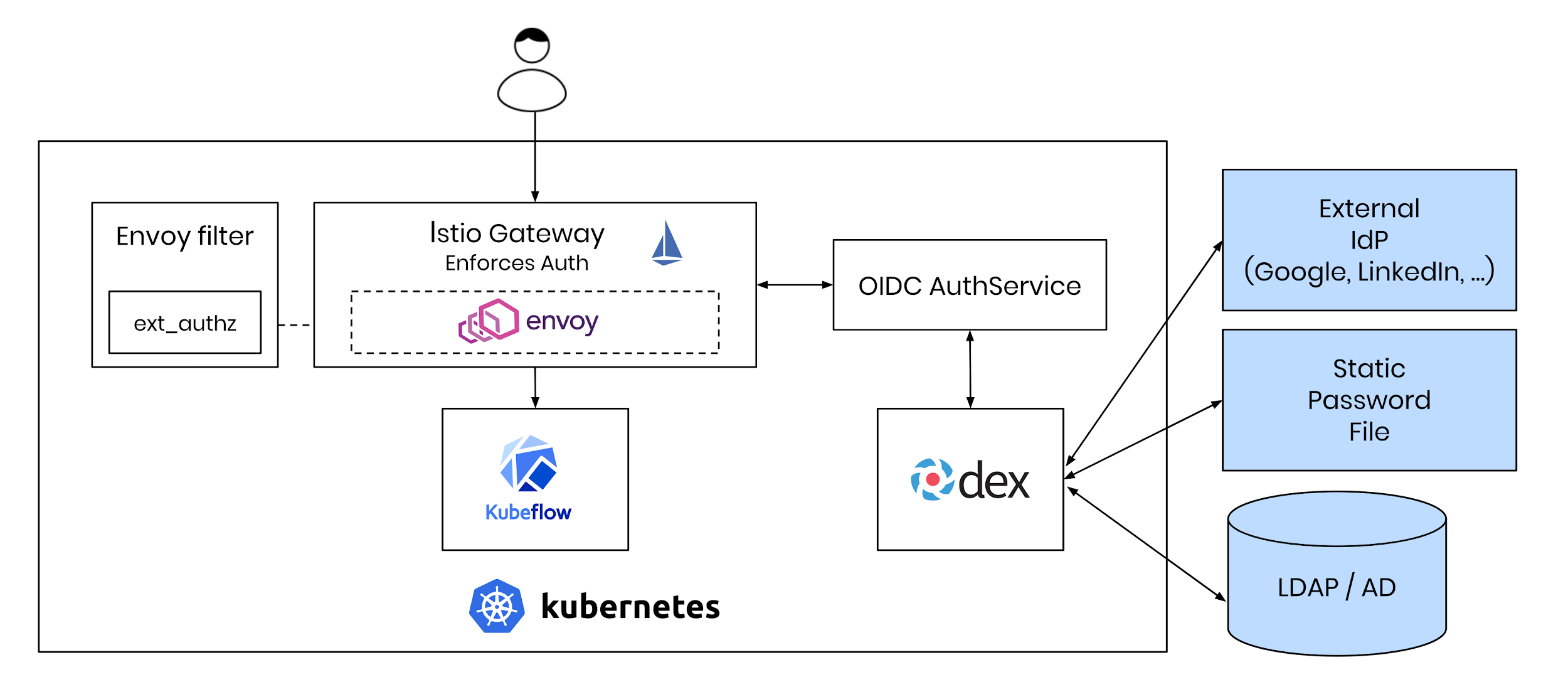
**Case #1 - User / North-South traffic**

In the Kubeflow authentication architecture, users come into the cluster from the Istio Gateway.

Authentication is done either at the Gateway (on-prem with Dex) or by a proxy (Google IAP).

After authentication is done, special headers are added which specify the user for this request (eg X-Goog-Authenticated-User-Email, kubeflow-userid).

This is similar to the [Kubernetes authenticating proxy method](https://kubernetes.io/docs/reference/access-authn-authz/authentication/#authenticating-proxy).



**Case #2 - Kubeflow Service / East-West traffic**

Data scientists work a lot with Jupyter Notebooks.

That’s why the Jupyter Notebooks in Kubeflow have a special ServiceAccount (kubeflow-edit) with permissions to create TFJobs, KatibJobs, etc.

Similarly, we would like for the user to be able to create a Pipeline or Run from their Notebook, using the Notebook’s ServiceAccount.

Reasons for using a ServiceAccount include:

* The Notebook may be shared amongst a team/project and user credentials are private.
* ServiceAccounts used by Notebooks are scoped to the namespace while user credentials gives access to all of the user’s namespaces.

**Option 1: TokenRequest and TokenReview API for scoped, secure ServiceAccountTokens**

An initial thought would be to send the Notebook’s ServiceAccount token to the Pipelines API-Server, where it can be verified with a [TokenReview](https://jpweber.io/blog/a-look-at-tokenrequest-api/) call to the K8s API-Server.

However, in the words of Joe Beda, [“This is a horrible horrible idea. You should never hand a k8s service account token to anything except the API server.”](https://twitter.com/jbeda/status/1010510544942424064?lang=en)

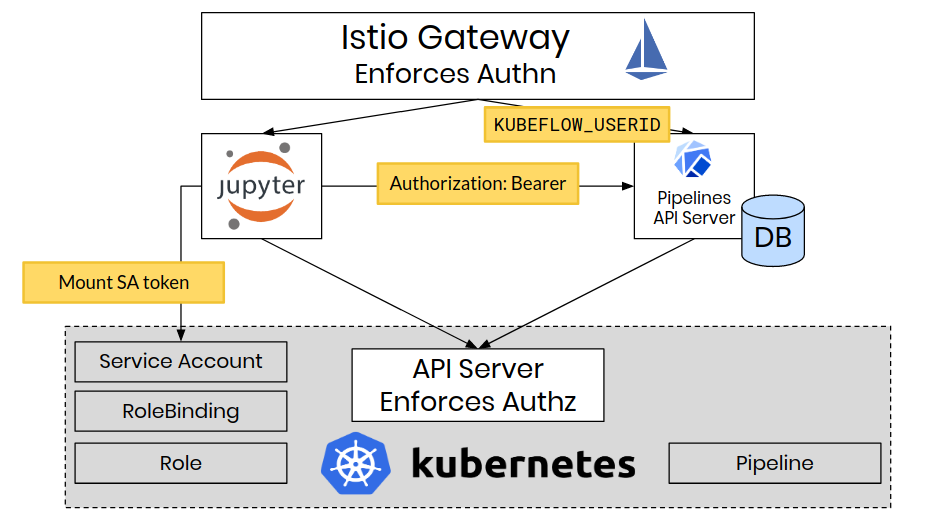
We need a way to authenticate to the Pipelines API Server that doesn’t involve trusting it with our credentials. Luckily, this is possible with the [TokenRequest API](https://github.com/mikedanese/community/blob/6e209490c441d8df84b6b5d8e352c0e2491a41bd/contributors/design-proposals/auth/bound-service-account-tokens.md).

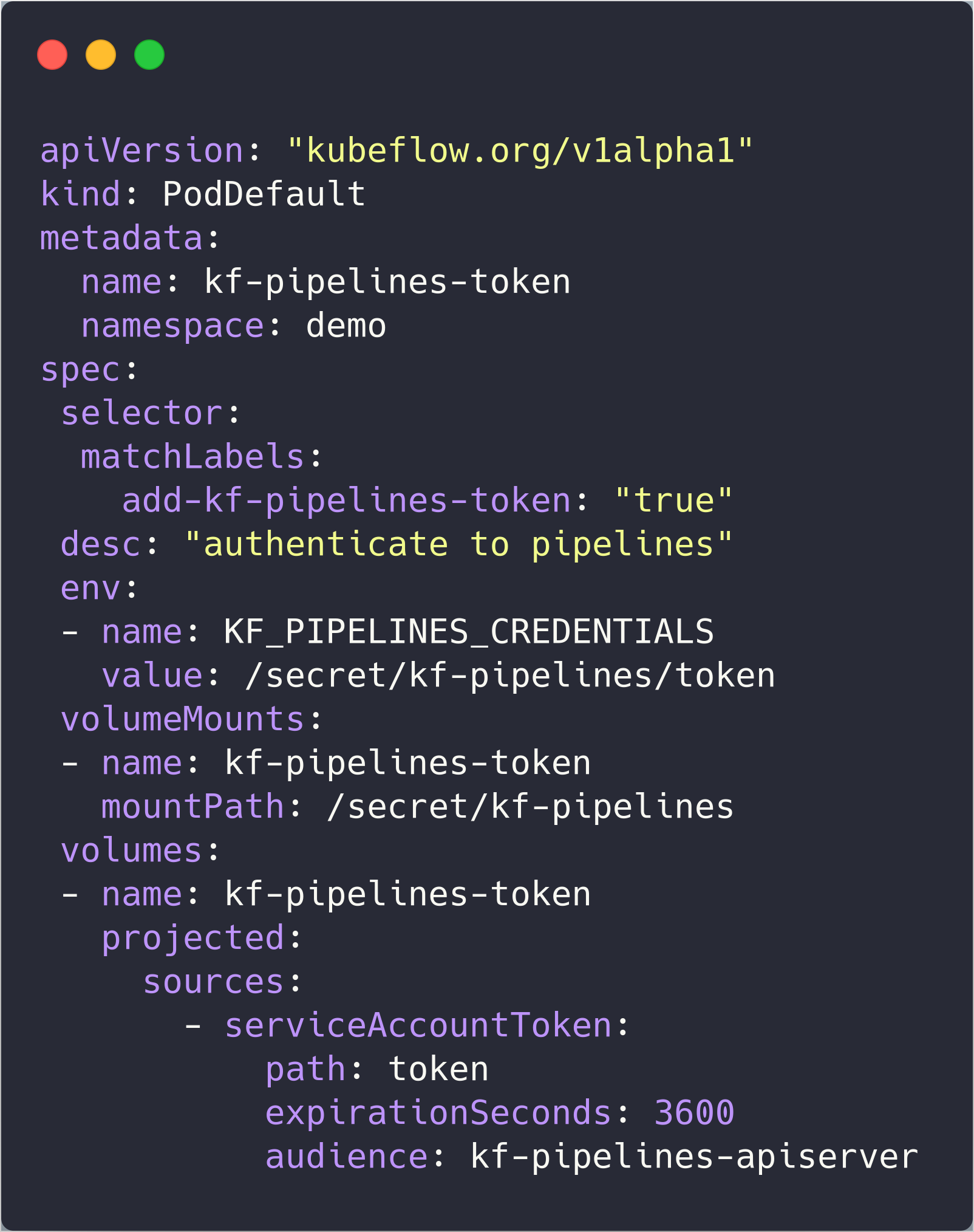
Using the TokenRequest API, we can ask K8s to issue a token that proves who we are (the ServiceAccount) but has a custom audience (the Pipelines API Server).

Thus, it is only accepted by Servers who identify as that audience and not by the K8s API Server.

TokenRequest is integrated as a projected volumesource, so issuing such a token is as simple as adding an extra volume to the PodSpec.

This can be modeled as a PodDefault and added to notebooks with the click of a button:





**Option 2: Istio Mutual TLS**

### If there is an Istio sidecar running alongside the KFP API Server, then we can make use of Istio Mutual TLS to authenticate the other party (see source.principal <https://istio.io/docs/reference/config/policy-and-telemetry/attribute-vocabulary/>)

A Pod running inside the cluster that has an Istio proxy should be able to authenticate to the KFP API Server in that way.  
The disadvantage of this method is that it won’t work for Pods that run as part of a Job/Pipeline, because of sidecar injection limitations.

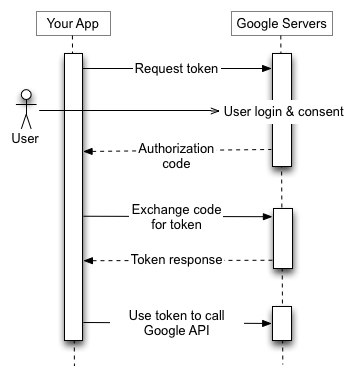
The next sections go into more detail about the implementations of the various ways to authenticate.

### Access through KFP UI/SDK - Authenticate with IAP

In the GCP environment, requests from the users are authenticated and identity tokens are generated by GCP IAM and carried through the requests as a JWT Token in the request headers. Users setup up [IAP](https://cloud.google.com/iap/docs/) for the authentication. When users use the KFP SDK to connect to the pipeline service, the SDK fetches the credentials; When users access the KFP UI, they need to login their google accounts for credentials;

In the on-prem environment, users can set up the [authentication with Istio and Dex](https://journal.arrikto.com/kubeflow-authentication-with-istio-dex-5eafdfac4782). Similar to the GCP environment, a signed JWT token needs to be returned from the Auth Server.

Requests are forwarded to Google Authorization Server to obtain the authorization code and the access token before passing the IAP gateway. After the IAP verifies the JWT token in the request header and signs it with its own private key, the IAP passes it to the Istio gateway. Finally, the istio gateway verifies the JWT token using the IAP’s public key.



#### Alternative - Authenticate with Inverse Proxy

[Inverse Proxy](https://github.com/google/inverting-proxy) does not support the OAuth consent flow, which leads to privacy issues. The multi-user support relies on the fact that the user identities are relayed on from the inverse proxy. Without the OAuth support, it is a privacy violation for the inverse proxy to pass on the PII data(user identity) to any downstream third-party applications without the user consent.

##### Option A: anonymizing the user identity in the Inverse Proxy.

A deterministic and irreversible algorithm can be applied to convert the plain user identity to an anonymized ID. Then the inverse proxy can forward the anonymized ID to the downstream third-party applications. The upside is that it involves little engineering work. The downside is two-fold: the administrator will not be able to make sense of the user permissions. However, they can still check the permissions for users. Second, there will be an implicit agreement between the inverse proxy and the KFP system about the anonymization algorithm used. Potential breaks can happen if one party changes the algorithm without having the other party aware.

##### Option B: drop the inverse proxy support.

This will make the multi-user support not available in the MKP deployment, which currently only supports inverse proxy, not the IAP.

##### Option C: add OAuth support to the inverse proxy.

This enables the inverse proxy to pass on the user identity down to the third-party applications with the user's consent.

### Access through kubectl - Authenticate with Auth Server

Before users access the KFP system using the kubectl, they run [gcloud command to obtain the credentials](https://cloud.google.com/kubernetes-engine/docs/how-to/cluster-access-for-kubectl#generate_kubeconfig_entry), which basically stores the kubeconfig entry locally. Then, users use kubectl to access the kubernetes clusters by passing the kubeconfig credentials. This is achieved by proxying Kubernetes Services through the Kubernetes master. (more info: <https://kubernetes.io/docs/tasks/administer-cluster/access-cluster-services/#manually-constructing-apiserver-proxy-urls>)



### In-cluster authentication

To secure the in-cluster communication, [Isito traffic management](https://istio.io/docs/concepts/traffic-management/) is adopted. Particularly, [Istio sidecar](https://istio.io/docs/reference/config/networking/v1alpha3/sidecar/) is enabled to mediate inbound and outbound communication. The detailed requirements are as follows:

* Allow traffic among all KFP components and from KFP components to KFAM and Argo(the dependent KF components)
* Deny traffic from non-KFP sources to KFP internal[[1]](#footnote-0) components including persistent agent, mysql, etc.
* Allow traffic from all sources to KFP external components[[2]](#footnote-1) including the UI and API.



Istio sidecar are injected to all KFP components except for Minio service such that the traffic among all KFP components, from KFP components to KFAM and Argo, from all sources to KFP UI/API are whitelisted. At the same time, Traffic from non-KFP sources to KFP internal components including mysql, etc. are denied based on the istio default traffic policy.

The Minio service is not injected with sidecar since the argo controller injects wait containers that connect to the Minio service and the istio auto sidecar injection is disabled to argo workflows due to the fact that argo workflows with istio sidecar injected cannot complete gracefully. It is not possible to whitelist the traffic from a pod without the istio sidecar to the service with the istio sidecar. In the future when argo workflows can gracefully complete with the sidecar injected, the minio service can also be injected with the istio sidecar for better traffic management.

Similarly, workflow cannot access KFP API until argo workflows support istio sidecars.

**Note:** The artifact traffic (Minio) doesn’t pass through the Pipelines API Server.

This means that authentication and authorization is left up to the object server.

In the current design, all users can access all artifacts stored in Minio.

To mitigate this, we have to either:

1. Make the artifact traffic pass through the Pipelines API Server, where we can impose access control on buckets/objects.
2. Use the access control model of the Object Server. Minio seems to be lacking on that front. With cloud offerings (eg S3, Google Storage), this should be more feasible.

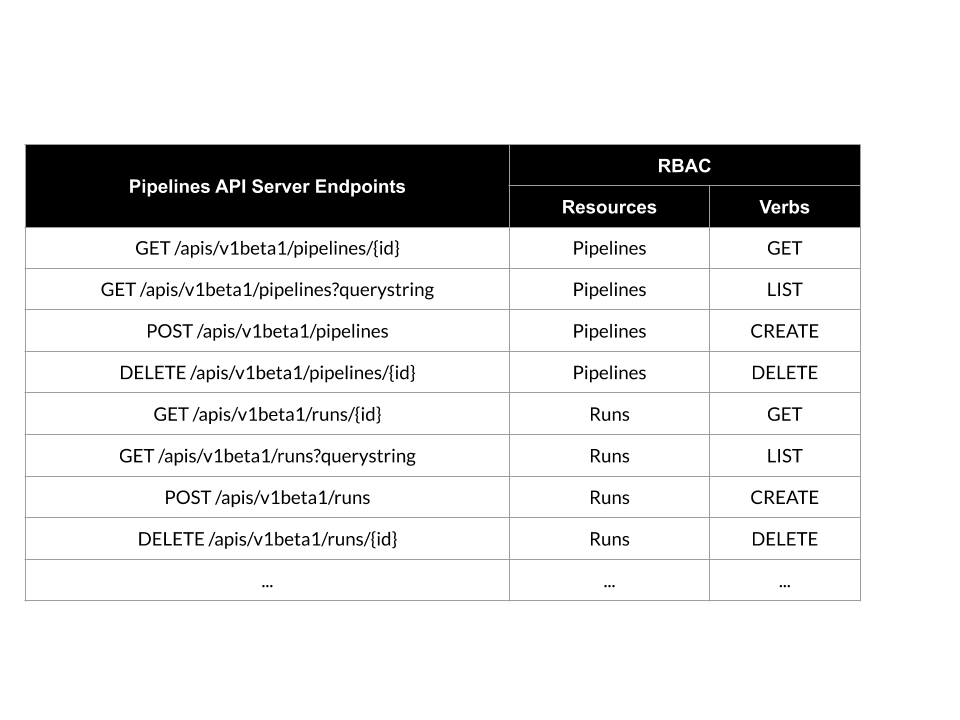
## Authorization

How does the Pipelines API Server know if we can take an action?

Currently, Kubeflow is relying on Kubernetes RBAC authorization for API Server resources.

In addition, the Kubeflow Pipeline resources are a good fit for the model.

We make a mapping of the Pipelines API Server to RBAC verbs. One or more endpoints are mapped to an RBAC verb. If needed, custom verbs can also be used.:



When a request arrives at the Pipelines API Server, it maps the request to a Resource, Verb and namespace.

After that, the Pipelines API Server makes a SubjectAccessReview request to the K8s API Server and asks:

“Is *USER*(found from authentication step) allowed to *VERB* *RESOURCE* (eg create Pipelines) in *NAMESPACE*?”

Advantages of using K8s RBAC to express permissions:

* Uniform representation of permissions across Kubeflow. All permissions are expressed in Roles and RoleBindings.
* Well-known model that we trust for everything else in Kubeflow.
* Kubernetes is the source of truth of authorization. Better visibility and governance.

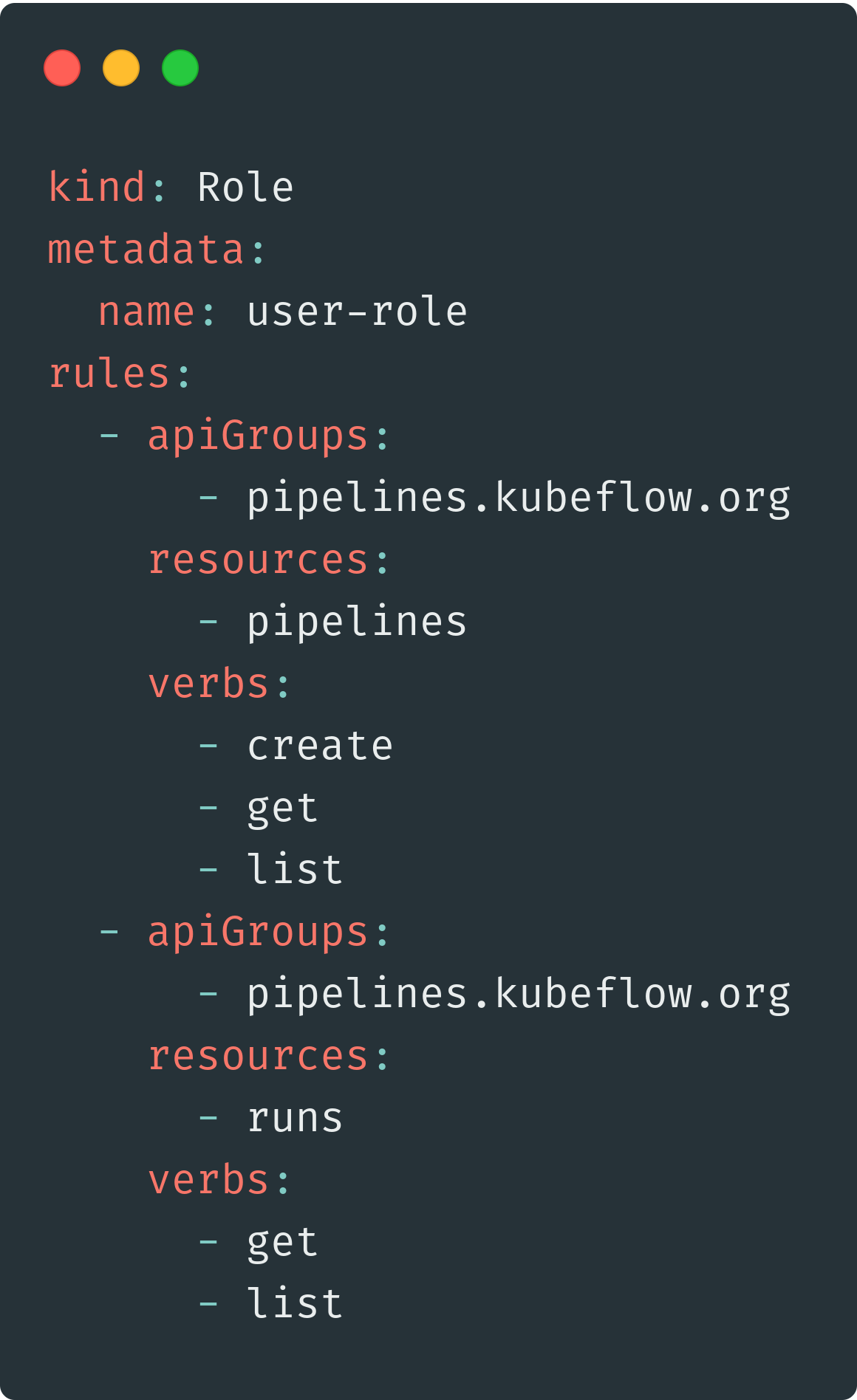
No CRDs need to be created for this to work! The Kubernetes authorizer works just on string matching, it doesn’t care if the resource/verb exists or not.

# User Journey

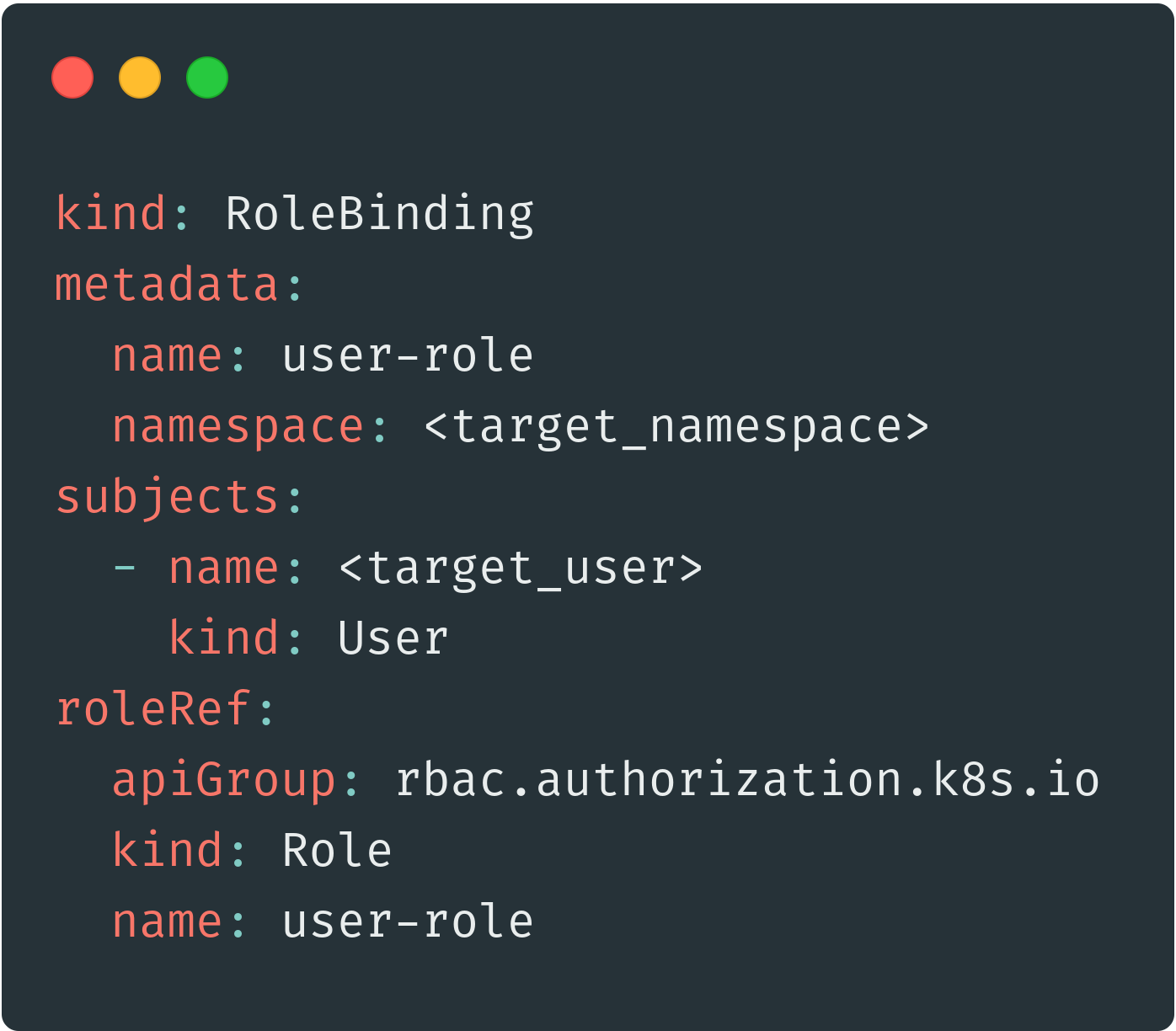
In this section, we follow the user journey of a Kubeflow / Pipelines admin and user using the proposed multi-user features.

**Scenario 1:** Admin wants to give a user permissions to read and create Pipelines and read Runs in a target namespace.

* Admin creates a Kubernetes Role, specifying the desired permissions:

[](https://carbon.now.sh/?bg=rgba(74%2C144%2C226%2C0)&t=material&wt=none&l=yaml&ds=false&dsyoff=20px&dsblur=68px&wc=true&wa=true&pv=0px&ph=0px&ln=false&fl=1&fm=Fira%20Code&fs=14px&lh=152%25&si=false&es=4x&wm=false&code=kind%253A%2520Role%250Ametadata%253A%250A%2520%2520name%253A%2520user-role%250Arules%253A%250A%2520%2520-%2520apiGroups%253A%250A%2520%2520%2520%2520%2520%2520-%2520pipelines.kubeflow.org%250A%2520%2520%2520%2520resources%253A%250A%2520%2520%2520%2520%2520%2520-%2520pipelines%250A%2520%2520%2520%2520verbs%253A%250A%2520%2520%2520%2520%2520%2520-%2520create%250A%2520%2520%2520%2520%2520%2520-%2520get%250A%2520%2520%2520%2520%2520%2520-%2520list%250A%2520%2520-%2520apiGroups%253A%250A%2520%2520%2520%2520%2520%2520-%2520pipelines.kubeflow.org%250A%2520%2520%2520%2520resources%253A%250A%2520%2520%2520%2520%2520%2520-%2520runs%250A%2520%2520%2520%2520verbs%253A%250A%2520%2520%2520%2520%2520%2520-%2520get%250A%2520%2520%2520%2520%2520%2520-%2520list)

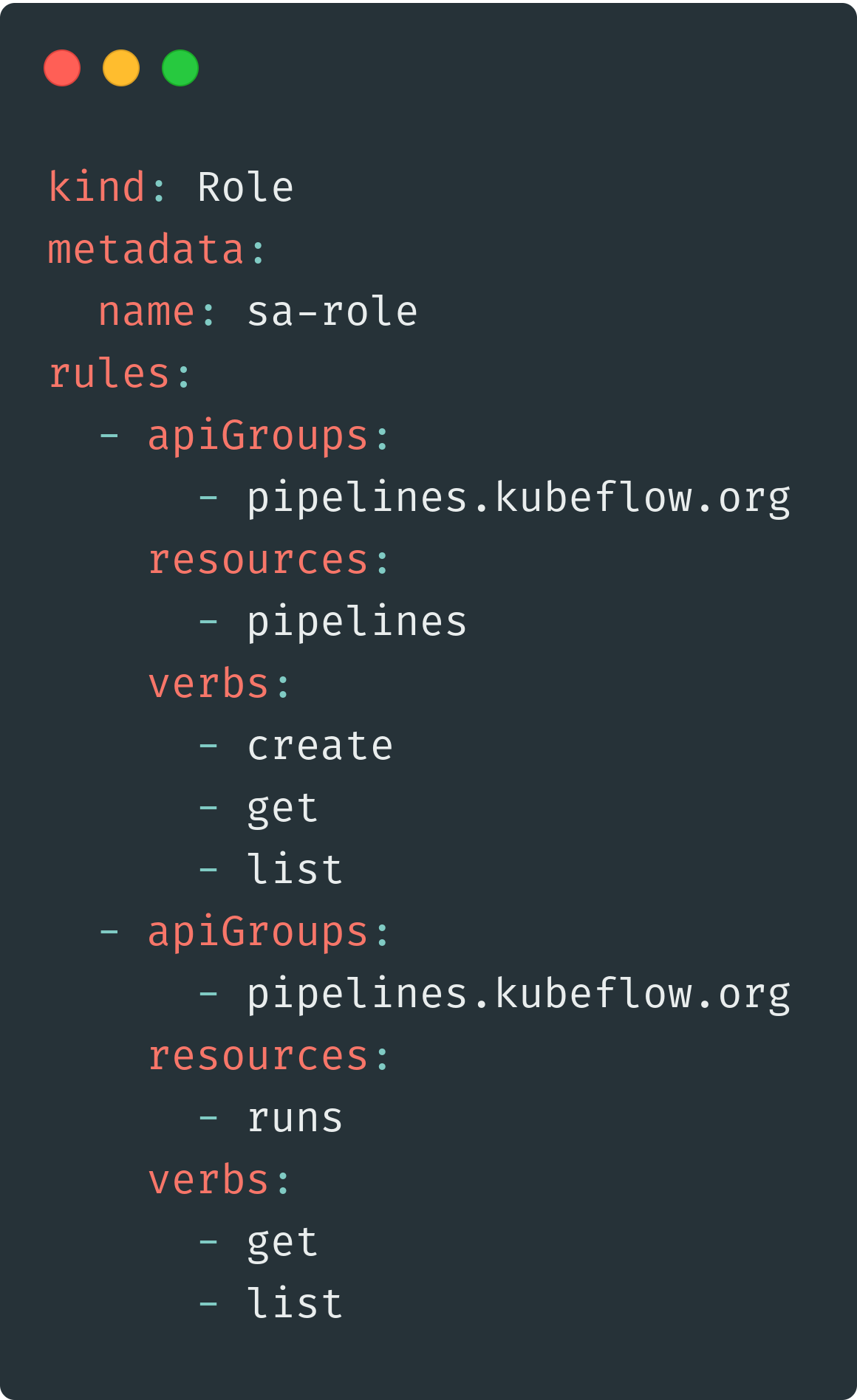
* Admin binds the Role to the desired user with a RoleBinding:

[](https://carbon.now.sh/?bg=rgba(74%2C144%2C226%2C0)&t=material&wt=none&l=yaml&ds=false&dsyoff=20px&dsblur=68px&wc=true&wa=true&pv=0px&ph=0px&ln=false&fl=1&fm=Fira%20Code&fs=14px&lh=152%25&si=false&es=4x&wm=false&code=kind%253A%2520RoleBinding%250Ametadata%253A%250A%2520%2520name%253A%2520user-role%250A%2520%2520namespace%253A%2520%253Ctarget_namespace%253E%250Asubjects%253A%250A%2520%2520-%2520name%253A%2520%253Ctarget_user%253E%250A%2520%2520%2520%2520kind%253A%2520User%250AroleRef%253A%250A%2520%2520apiGroup%253A%2520rbac.authorization.k8s.io%250A%2520%2520kind%253A%2520Role%250A%2520%2520name%253A%2520user-role)

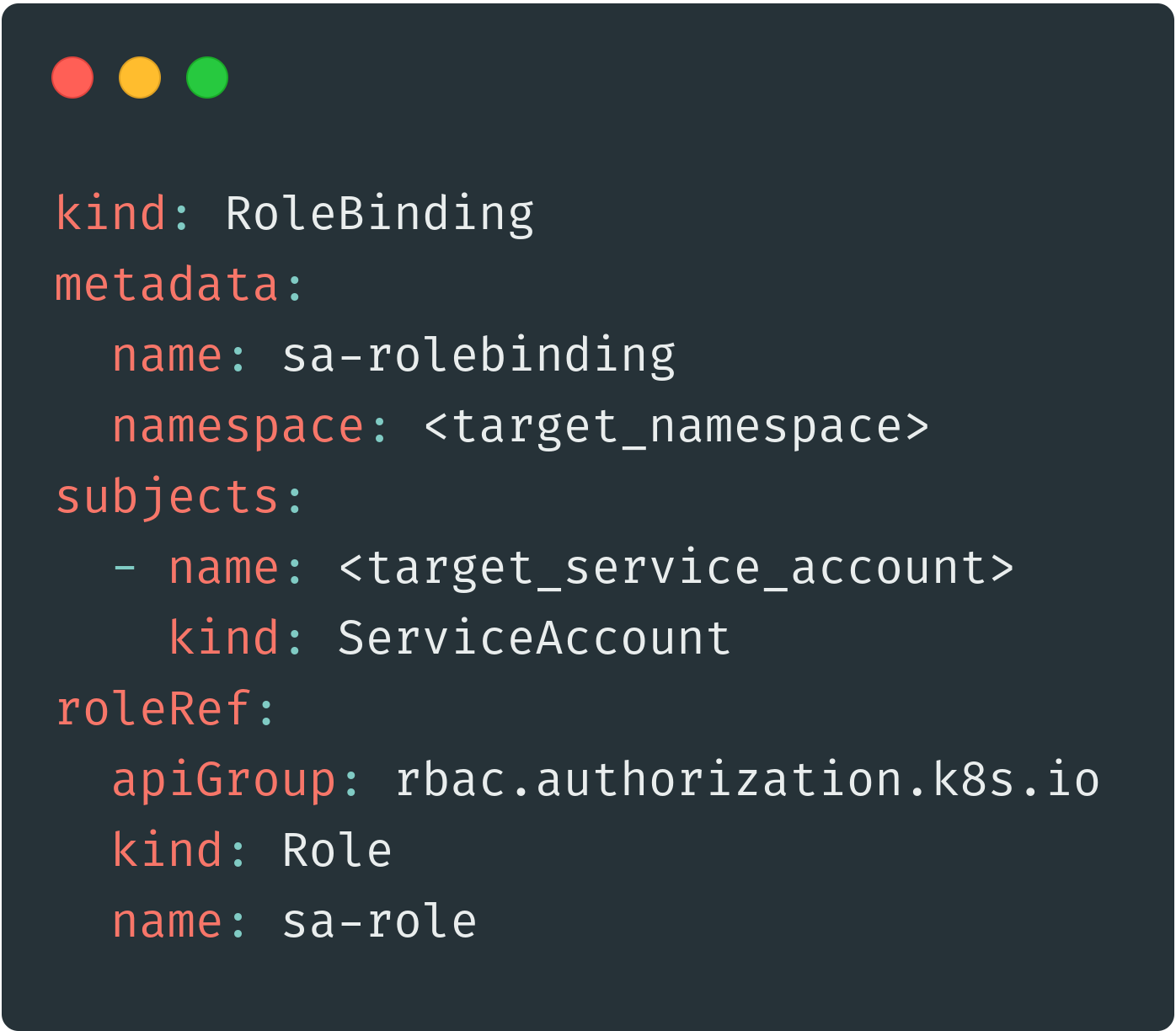
* The user can now read and create Pipelines and read (not create) Runs in the target namespace. The user’s actions are isolated inside the target namespace.

**Scenario 2:** Admin wants to deploy an automated Service which accesses the Pipelines API Server. Since this is a service, not a user, the Admin will use a ServiceAccount, as per the Kubernetes design.

* Admin creates a Kubernetes Role, specifying the desired permissions:

[](https://carbon.now.sh/?bg=rgba(74%2C144%2C226%2C0)&t=material&wt=none&l=yaml&ds=false&dsyoff=20px&dsblur=68px&wc=true&wa=true&pv=0px&ph=0px&ln=false&fl=1&fm=Fira%20Code&fs=14px&lh=152%25&si=false&es=4x&wm=false&code=kind%253A%2520Role%250Ametadata%253A%250A%2520%2520name%253A%2520sa-role%250Arules%253A%250A%2520%2520-%2520apiGroups%253A%250A%2520%2520%2520%2520%2520%2520-%2520pipelines.kubeflow.org%250A%2520%2520%2520%2520resources%253A%250A%2520%2520%2520%2520%2520%2520-%2520pipelines%250A%2520%2520%2520%2520verbs%253A%250A%2520%2520%2520%2520%2520%2520-%2520create%250A%2520%2520%2520%2520%2520%2520-%2520get%250A%2520%2520%2520%2520%2520%2520-%2520list%250A%2520%2520-%2520apiGroups%253A%250A%2520%2520%2520%2520%2520%2520-%2520pipelines.kubeflow.org%250A%2520%2520%2520%2520resources%253A%250A%2520%2520%2520%2520%2520%2520-%2520runs%250A%2520%2520%2520%2520verbs%253A%250A%2520%2520%2520%2520%2520%2520-%2520get%250A%2520%2520%2520%2520%2520%2520-%2520list)

* Admin binds the Role to the desired ServiceAccount with a RoleBinding:

[](https://carbon.now.sh/?bg=rgba(74%2C144%2C226%2C0)&t=material&wt=none&l=yaml&ds=false&dsyoff=20px&dsblur=68px&wc=true&wa=true&pv=0px&ph=0px&ln=false&fl=1&fm=Fira%20Code&fs=14px&lh=152%25&si=false&es=4x&wm=false&code=kind%253A%2520RoleBinding%250Ametadata%253A%250A%2520%2520name%253A%2520sa-rolebinding%250A%2520%2520namespace%253A%2520%253Ctarget_namespace%253E%250Asubjects%253A%250A%2520%2520-%2520name%253A%2520%253Ctarget_service_account%253E%250A%2520%2520%2520%2520kind%253A%2520ServiceAccount%250AroleRef%253A%250A%2520%2520apiGroup%253A%2520rbac.authorization.k8s.io%250A%2520%2520kind%253A%2520Role%250A%2520%2520name%253A%2520sa-role)

* Now the Service has the necessary clearance to access the Pipelines API Server in the target namespace.

**Scenario 3:** User wants to access Pipelines from the UI.

* User logs into Kubeflow and goes to the Pipelines UI.
* The Pipelines UI makes requests to the Pipelines API Server. These requests include the user’s credentials from the browser (cookie).
* That cookie gets translated at the Istio Gateway (or the Google IAP) into an http header defining the user.
* The Pipelines API Server knows who the user is and what they request. It performs a SubjectAccessReview to see if the user has the necessary permissions to access the specific API endpoint.

**Scenario 4:** User wants to access Pipelines from the CLI/SDK.

* If the environment is a private one, User can authenticate with their credentials.
* If the environment is a shared one (eg Notebook in shared Kubeflow Profile), User cannot authenticate with their credentials.
  + Instead, the ServiceAccount of the Notebook is authenticated.
  + The Pipelines API Server Audit logs trace the origin of those requests to the specific ServiceAccount, thus the specific Profile (namespace).
* When User creates a Notebook, they select the “Authenticate to Pipelines” PodDefault. This creates a ServiceAccount token with a custom audience of “pipelines.kubeflow.svc.cluster.local”. This token is only accepted by the Pipelines API Server.
* User connects to the Pipelines API Server using that token.

## Resource isolation



* For Kubeflow Pipeline databases stored in the MySQL, a separate **view**( or namespace)[[3]](#footnote-2) column is inserted to isolate metadata resources. The database services including MySQL service and Minio service are unaware of the namespaces. When KFP lists the resources for a namespace, it sends a request with the filtering on **view** to the database services.
* Externally accessible services handle the namespaced resources and expose the namespace parameter in the APIs. These services include KFP API service and KFP frontend service.
* Internal service, i.e. KFP visualization service, is deployed for each namespace when a new namespace is created(on-demand).
* Artifact resources (Minio) are not isolated, since this traffic doesn’t pass through the Pipelines API Server.

## Implementation Plan

**Phase 1 (late Feb - early Mar)**

* Namespace field in experiments. An experiment can only exist in one namespace.
* Runs / Jobs belong to an experiment, so we find their namespace by getting their experiment.
* Access control is enforced per-namespace, not per-experiment. Inter-namespace isolation is generally an anti-pattern in Kubernetes.
* For authentication, the http header is consulted.
* For authorization, KFAM is used.

**Phase 2**

* Move away from KFAM and use Kubernetes RBAC and SubjectAccessReview.
  + KFAM is not intended to be an authorization layer on top of Kubernetes.
  + Users have mentioned the need for more granular access control (cc @animesh)
* Enable authentication with ServiceAccount.

1. KFP internal components are functional components that users do not have to connect to directly. [↑](#footnote-ref-0)
2. KFP external components are user facing components. [↑](#footnote-ref-1)
3. View is a column for filtering. In the multi-user context, it is the namespace. [↑](#footnote-ref-2)