mia Platform

Open Policy Agent + Rönd = ❤️

A policy-as-code approach to RBAC authorization.

mia Platform

Just two cents about me.

Developer Relations Engineer @ mia Platform

castograziano.com | blog.mia-platform.eu

Casto Graziano ✌🏻

01

Understand policies.

mia Platform

```python
def process_order(user, order):
    if user.role == "store_manager" and order.store in user.stores:
        delete_order(order)
    else:
        deny()
```

If the user is a store manager and the order is assigned to one of their stores, **then** the user can cancel the order.

mia Platform

```
def process_order(user, order):
    if user.role == "store_manager" and order.store in user.stores:
        delete_order(order)
    else:
        deny()
```

If the user is a store manager and the order is assigned to one of their stores, **then** the user can cancel the order.

```
def process_cart(product):
    if product.stock > 0:
        add_to_cart(product)
    else:
        deny()
```

If the product is available in stock, **then** it can be added to the cart.

mia Platform

```
def process_order(user, order):
    if user.role == "store_manager" and order.store in user.stores:
        delete_order(order)
    else:
        deny()
```

If the user is a store manager and the order is assigned to one of their stores, **then** the user can cancel the order.

```
def process_cart(product):
    if product.stock > 0:
        add_to_cart(product)
    else:
        deny()
```

If the product is available in stock, **then** it can be added to the cart.

```
def sell_alcohol(user):
    if user.age ⩾ 21 and time(6, 0) ⩽ current_time ⩽ time(23, 0):
        sell_alcohol()
    else:
        deny()
```

If the buyer is over 21 years old and the transaction takes place between 6 a.m. and 11 p.m., **then** alcohol can be sold.

mia Platform

Definition.

A policy is **a set of rules** that govern the behaviour of a software service.

It simply **describe invariants** that must hold in a software system.

01 Not necessarily associated with users and roles.

02 **Authorization** is a special kind of policy that often dictates which people or machine can run which action on which resources.

02

Access control strategies.

RBAC: Why use roles?

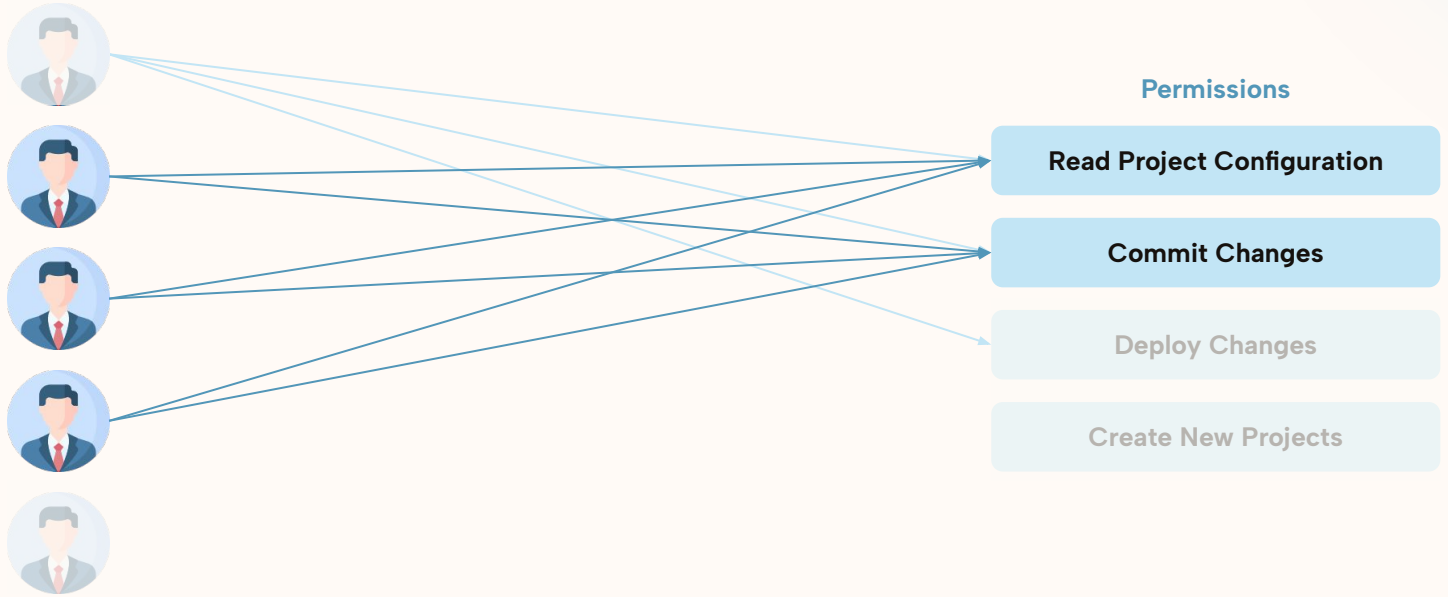
Senior Developer

Permissions

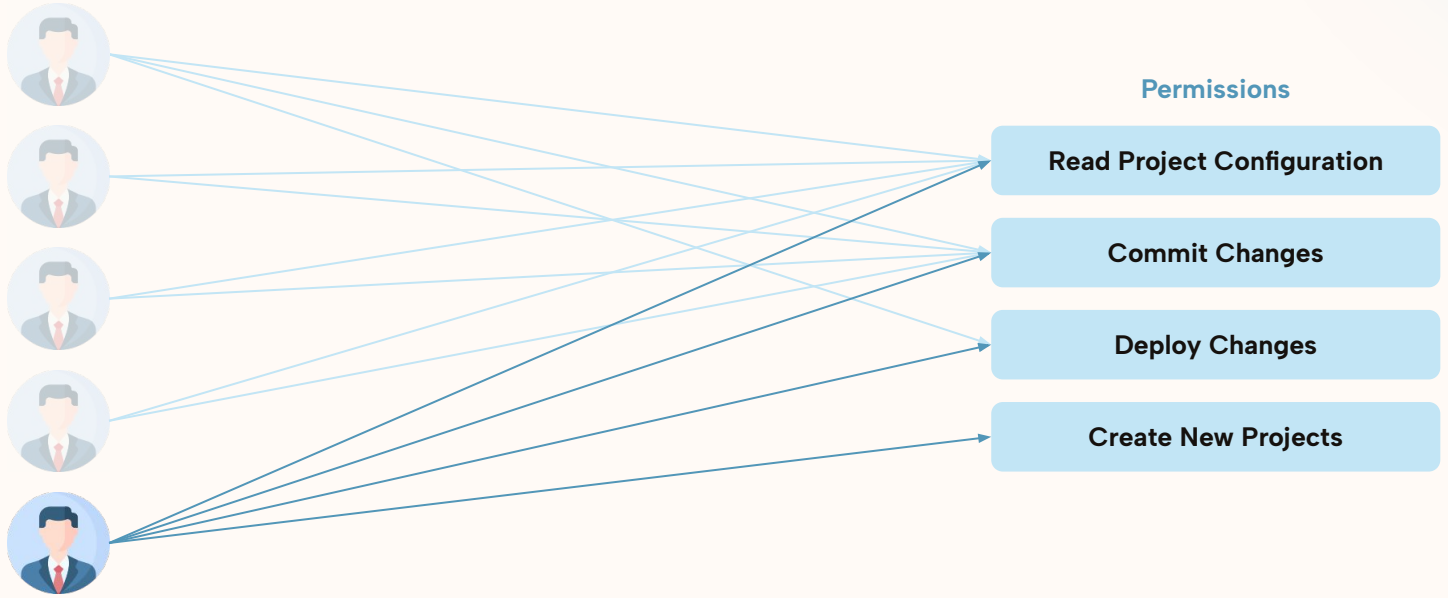
- Read Project Configuration
- Commit Changes
- Deploy Changes
- Create New Projects

RBAC: Why use roles?

Junior Developers

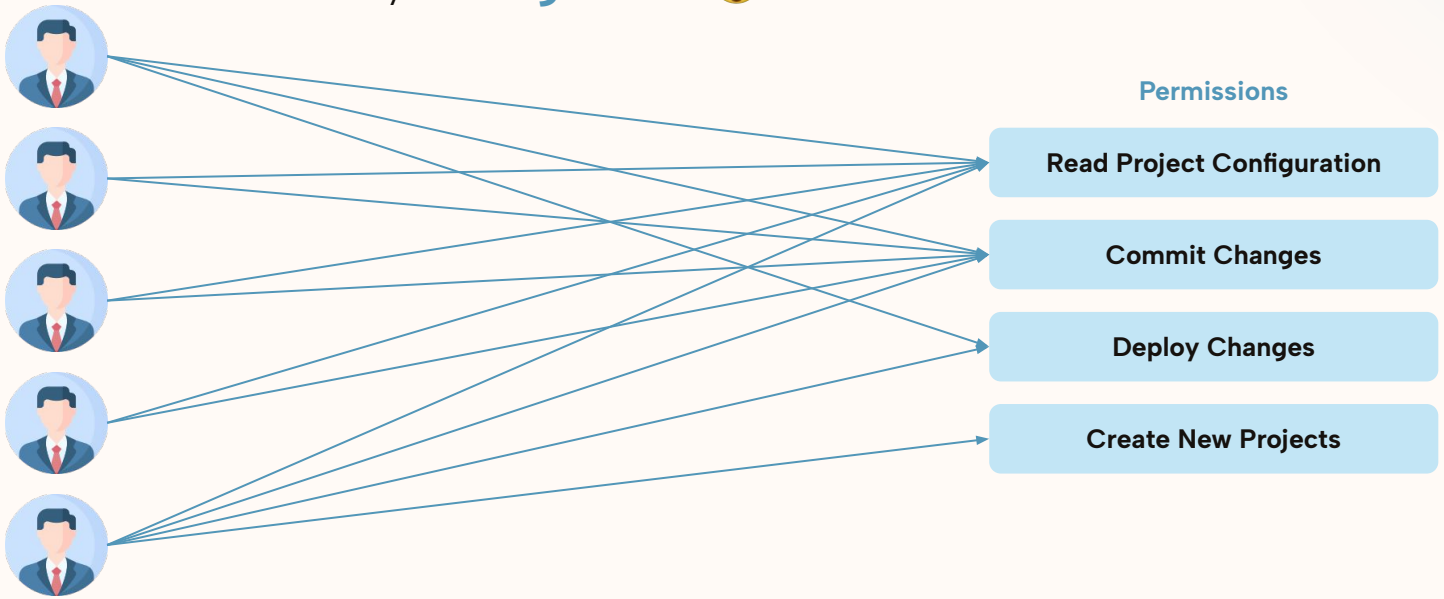


Platform Administrator



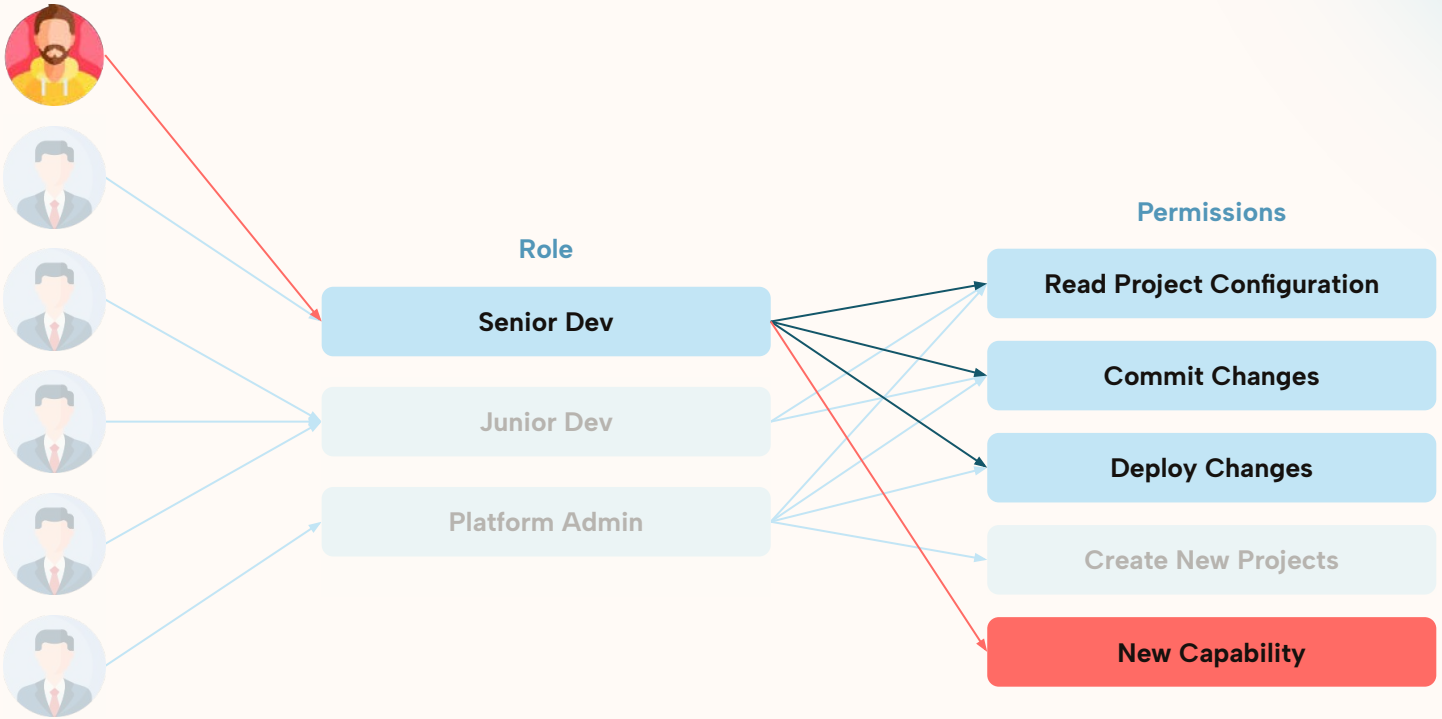
RBAC: Why use roles?

Scaling access management in this way is a **big mess**! 🤯



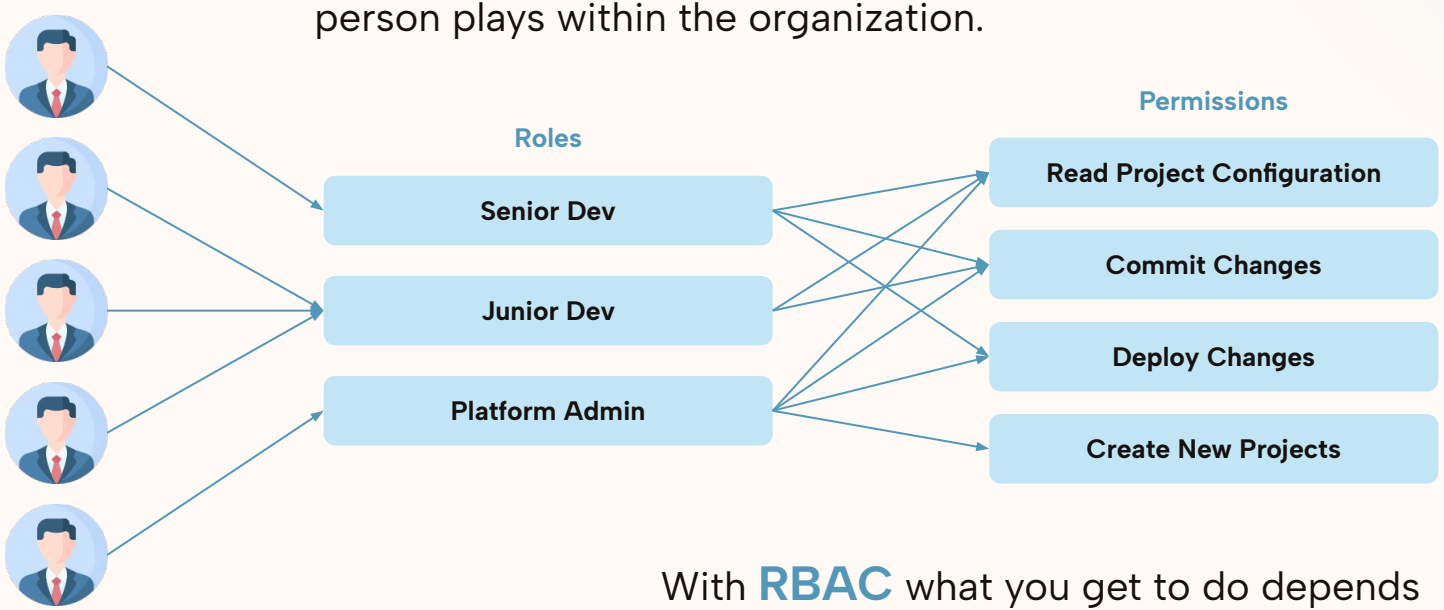
mia Platform

RBAC: Why use roles?



RBAC: Why use roles?

We simplify by **abstracting out complexity** based upon the **role** the person plays within the organization.



With **RBAC** what you get to do depends on the **role** you are assigned to.

mia Platform

ABAC: why use attributes?

Project :: Ecommerce App

Project :: Data Platform

Project :: API Service

Env :: PROD

Env :: NOPROD

User's attributes

If

Project

==

Ecommerce App

&&

Env

==

NOPROD

Condition

Then

Deploy Changes

over

Project :: Ecommerce App

Env :: NOPROD

Action

With **ABAC** evaluates **attributes** rather than roles, to determine access.

mia Platform

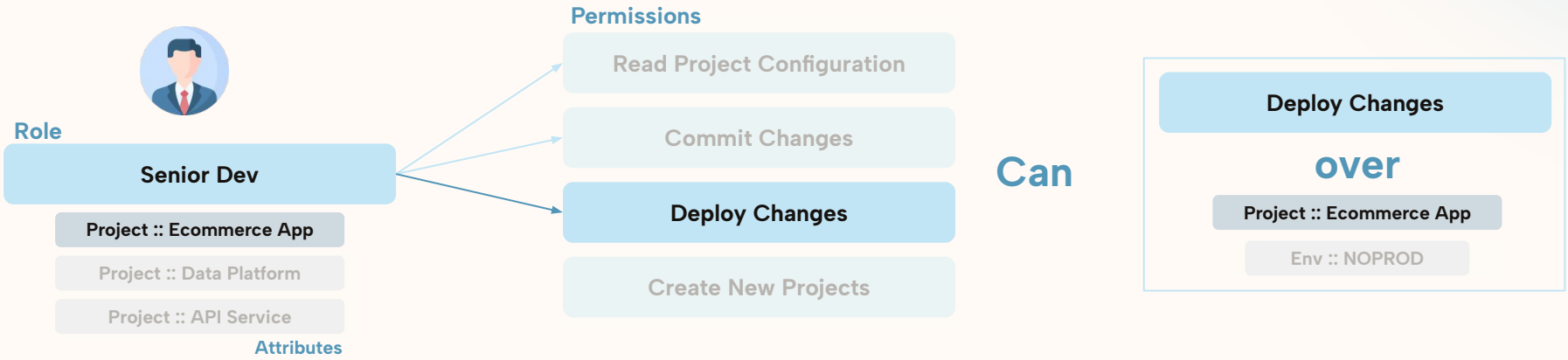
Which one is **the best**?

	Role-Based Access Control	vs	Attribute-Based Access Control
Pros	**Simplicity** – Rules within the RBAC system are simple and easy to execute.		**Granularity** – You can develop very specific and granular rules that protect your assets.
Cons	**Granularity** – To add granularity to their systems, some administrators add more roles. That can lead to a role explosions with hundreds or even thousands of rules to manage.		**Time** – Defining variables and configuring your rules is a massive effort, especially at project kickoff. **Expertise** – Appropriate ABAC rules lead to accurate implementation. If you set up the system wrong at the outset, the fix could be time-consuming.

Which one is **the best**?

Setup an **hybrid approach** with both **RBAC** and **ABAC**. Use **roles** for high-level access control, then use **attributes** to fine grained access control over specific assets.

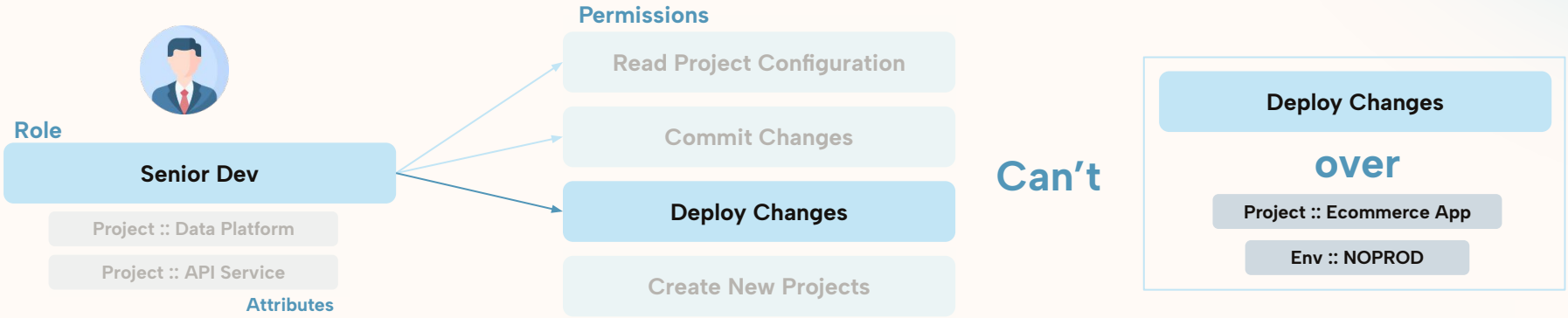
RBAC: Why use roles?



A **Senior Developer** can deploy changes for his projects, it doesn't matter the environment.

mia Platform

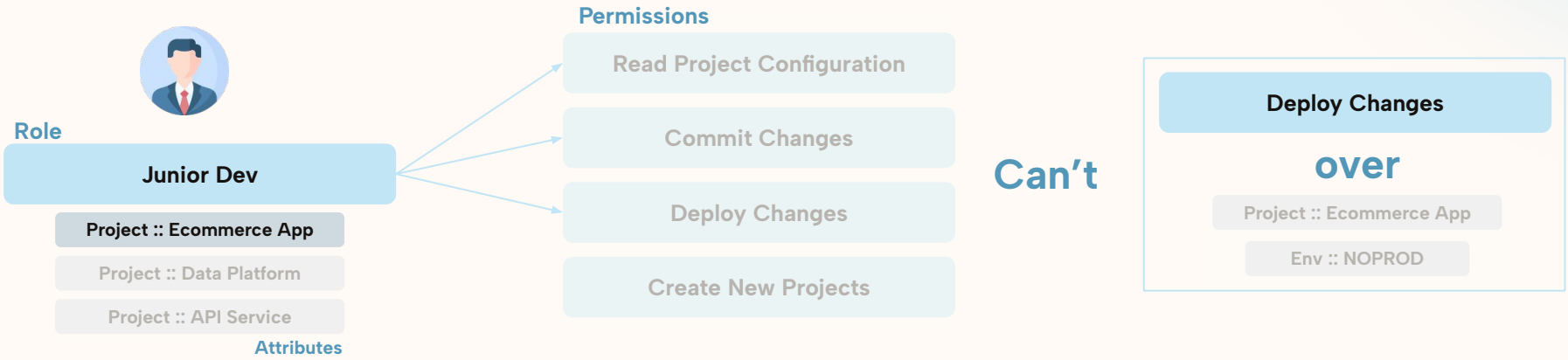
RBAC: Why use roles?



A **Senior Developer** can't deploy changes for projects not assigned to him.

mia Platform

RBAC: Why use roles?



A **Junior Developer** can't deploy any changes,
it doesn't matter the project.

RBAC: Why use roles?

A **Junior Developer** can deploy changes for his projects only if he is allowed for the specific environment.

03

Enforce policies.

mia Platform

Old but gold: just make things works!

```python
class DeploymentRoute:

    def is_authorized(user, project, env):
        if user.role == "Senior Developer" and project in user.projects:
            return True
        elif user.role == "Junior Developer" and project in user.projects and env in user.envs:
            return True
        return False

    def deploy_changes(user, project, env, changes):
        if not is_authorized(user, project, env):
            Logger.log("You are not authorized to deploy changes!")
            return
        try:
            commit = GitProvider("gitlab").commit(changes)
            CICDProvider("gitlab").pipeline("deploy").run(commit.sha, env)
            Logger.log(f"Deployment successful for {env} environment!")
        except Exception as e:
            Logger.log(f"Deployment failed: {str(e)}")
```

Policy Enforcement.

mia Platform

Old but gold: just make things works!

```python
class DeploymentRoute:

    def is_authorized(user, project, env):
        if user.role == "Senior Developer" and project in user.projects:
            return True
        elif user.role == "Junior Developer" and project in user.projects and env in user.envs:
            return True
        return False

    def deploy_changes(user, project, env, changes):
        if not is_authorized(user, project, env):
            Logger.log("You are not authorized to deploy changes!")
            return
        try:
            commit = GitProvider("gitlab").commit(changes)
            CICDProvider("gitlab").pipeline("deploy").run(commit.sha, env)
            Logger.log(f"Deployment successful for {env} environment!")
        except Exception as e:
            Logger.log(f"Deployment failed: {str(e)}")
```

Business Logic.

mia Platform

Old but gold: just make things works!

```python
class DeploymentRoute:

    def is_authorized(user, project, env):
        if user.role == "Senior Developer" and project in user.projects:
            return True
        elif user.role == "Junior Developer" and project in user.projects and env in user.envs:
            return True
        return False

    def deploy_changes(user, project, env, changes):
        if not is_authorized(user, project, env):
            Logger.log("You are not authorized to deploy changes!")
            return
        try:
            commit = GitProvider("gitlab").commit(changes)
            CICDProvider("gitlab").pipeline("deploy").run(commit.sha, env)
            Logger.log(f"Deployment successful for {env} environment!")
        except Exception as e:
            Logger.log(f"Deployment failed: {str(e)}")
```

Policy Enforcement and **Business Logic** in the same codebase

Faster – Just make it works.

Simpler – Everything is in the same place.

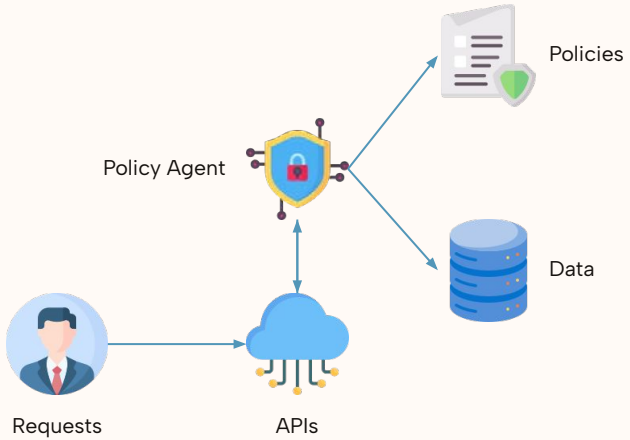
Repetition – If the same rule is needed in another component of your system you have to repeat it.

Maintainability – If rules changes you have to change your codebase even if your business logic doesn't change.

Messy – At large scale is difficult to keep track and maintain all your rules over the codebase.

mia Platform

Policy-as-code is the way to go!



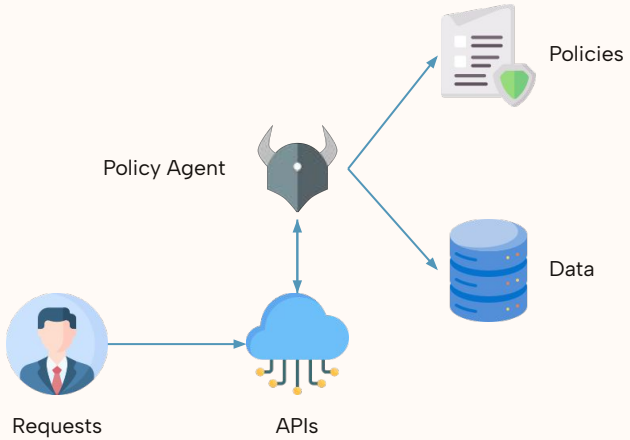
Policy-as-code is an approach to policy management in which policies are defined, updated, and enforced using code.
This approach allow us to **decouple** policy definition from policy enforcement.

Transparency – When policies are defined in code, it's easy for all stakeholders to use the code to understand what is happening within a system.

Accuracy – When teams define and manage policies using code, they avoid the risk of making configuration mistakes when managing a system manually.

Testability – When policies are written in code, it's easy to validate them using automated auditing tools.

Decoupling policies with **Open Policy Agent** (OPA)



OPA is an open-source **CNCF graduated project**. It provides a high-level declarative language (**Rego**) that lets you specify policy as code and simple APIs to offload policy decision-making from your software.

```
package example.authz

default allow := false

allow if {
    input.method == "POST"
    input.path == "/order/"
    input.user.store == input.body.store
}
```

mia Platform

Integrate OPA inside our project

It become a Rego policy, and...

```python
class DeploymentRoute:

    def is_authorized(user, project, env):
        if user.role == "Senior Developer" and project in user.projects:
            return True
        elif user.role == "Junior Developer" and project in user.projects and env in user.envs:
            return True
        return False

    def deploy_changes(user, project, env, changes):
        if not is_authorized(user, project, env):
            Logger.log("You are not authorized to deploy changes!")
            return
        try:
            commit = GitProvider("gitlab").commit(changes)
            CICDProvider("gitlab").pipeline("deploy").run(commit.sha, env)
            Logger.log(f"Deployment successful for {env} environment!")
        except Exception as e:
            Logger.log(f"Deployment failed: {str(e)}")
```

```
package authz

default allow = false

user_has_role(required_role) {
    authz_jwt := input.request.headers["Authorization"][0]
    decoded_jwt_data := io.jwt.decode(authz_jwt)
    decoded_jwt := decoded_jwt_data[1]
    role := decoded_jwt["role"]
    role == required_role
}

user_belong_to_project(required_project) {
    authz_jwt := input.request.headers["Authorization"][0]
    decoded_jwt_data := io.jwt.decode(authz_jwt)
    decoded_jwt := decoded_jwt_data[1]
    projects := decoded_jwt["projects"]
    some project in projects
    project == required_project
}

user_allowed_environments(required_env) {
    authz_jwt := input.request.headers["Authorization"][0]
    decoded_jwt_data := io.jwt.decode(authz_jwt)
    decoded_jwt := decoded_jwt_data[1]
    envs := decoded_jwt["env"]
    some env in envs
    env == required_env
}

allow {
    user_has_role("Senior Developer")
    user_belong_to_project(input.parsed_body.project)
}

allow {
    user_has_role("Junior Developer")
    user_belong_to_project(input.parsed_body.project)
    user_allowed_environments(input.parsed_body.env)
}
```

mia Platform

Integrate OPA inside our project

```
class DeploymentRoute:

    def is_authorized(user, project, env):
        if user.role == "Senior Developer" and project in use
            return True
        elif user.role == "Junior Developer" and project in u
            return True
        return False

    def deploy_changes(user, project, env, changes):
        if not is_authorized(user, project, env):
            Logger.log("You are not authorized to deploy chan
            return
        try:
            commit = GitProvider("gitlab").commit(changes)
            CICDProvider("gitlab").pipeline("deploy").run(commit.sha, env)
            Logger.log(f"Deployment successful for {env} environment!")
        except Exception as e:
            Logger.log(f"Deployment failed: {str(e)}")
```

```
def deploy_changes(user, project, env, changes):
        allowed_request = request.post("<your-opa-url>/v1/data/authz")
        if not allowed_request:
            Logger.log("You are not authorized to deploy changes!")
            return
        try:
            commit = GitProvider("gitlab").commit(changes)
            CICDProvider("gitlab").pipeline("deploy").run(commit.sha, env)
            Logger.log(f"Deployment successful for {env} environment!")
        except Exception as e:
            Logger.log(f"Deployment failed: {str(e)}")
```

... our business logic is just business logic, without any authz rule!

Integrate OPA inside our project


```
allow {
    user_has_role("Senior Developer")
    user_belong_to_project(input.parsed_body.project)
}

allow {
    user_has_role("Junior Developer")
    user_belong_to_project(input.parsed_body.project)
    user_allowed_environments(input.parsed_body.env)
}
```

Executing our policy if one of these **“allow”** statements is true then the request is authorized.

Integrate OPA inside our project


```
user_has_role(required_role) {
    authz_jwt := input.request.headers["Authorization"][0]
    decoded_jwt_data := io.jwt.decode(authz_jwt)
    decoded_jwt := decoded_jwt_data[1]
    role := decoded_jwt["role"]
    role == required_role
}
```

User's role, companies and envs are evaluated using functions. User's needed information are taken decoding the **JWT Token**.

mia Platform

Integrate OPA inside our project

```python
def deploy_changes(user, project, env, changes):
    allowed_request = request.post("<your-opa-url>/v1/data/authz")
    if not allowed_request:
        Logger.log("You are not authorized to deploy changes!")
        return
    try:
        commit = GitProvider("gitlab").commit(changes)
        CICDProvider("gitlab").pipeline("deploy").run(commit.sha, env)
        Logger.log(f"Deployment successful for {env} environment!")
    except Exception as e:
        Logger.log(f"Deployment failed: {str(e)}")
```

OPA can be integrated in our project using **REST APIs** or via many **SDKs libraries**...

...but we can **decouple** our policy evaluation even more.😏

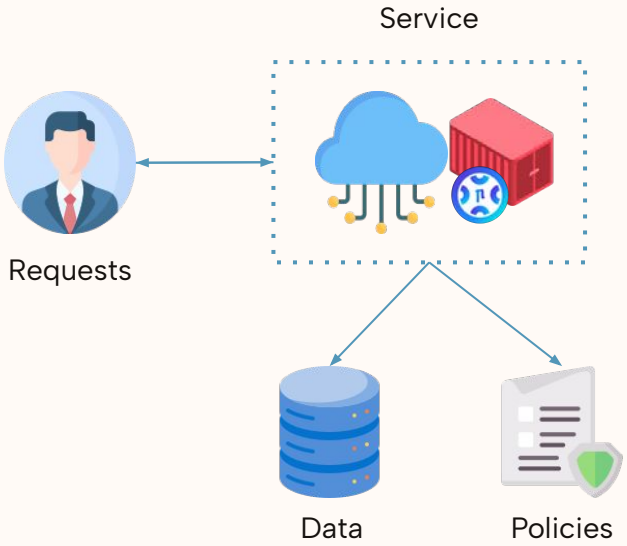
04

Introducing Röndd.

Röndd

It’s an open-source lightweight Kubernetes **sidecar container** that helps you protect your APIs with simple security policies.

It uses **OPA** as security engine for validating authorization rules, and leverages Rego language for writing the security policies.



Röndd is an authorization mechanism, but it also natively allows you to build an **RBAC** solution by defining the concepts of **Roles**, **Permissions**, and **User Groups** as building blocks.

Configure Rönd

```
{
  "paths": {
    "/deploy": {
      "post": {
        "x-rond": {
          "requestFlow": {
            "policyName": "allow_deploy"
          }
        }
      }
    }
  }
}
```

You can configure Rönd by **declaring the routes to protect** and for each route and http verb you just need to define the policy name that will protect the specific request.

Configure Rönd

```
{
   "paths":{
      "/projects":{
         "get":{
            "x-rond":{
               "requestFlow":{
                  "policyName":"filter_projects",
                  "generateQuery":true,
                  "queryOptions":{
                     "headerName":"x-query-header"
                  }
               },
               "responseFlow":{
                  "policyName":"protect_project_info"
               }
            }
         }
      }
   }
}
```

```
filter_projects {
    user_has_role("Senior Developer")
    query := data.resources[_]
} {
    user_has_role("Junior Developer")
    query := data.resources[_]
    query.status == "READY"
}
```

With Rönd you can use query generation to **filter the response** of your apis before returning it to the clients ...

Configure Rönd


```
protect_project_info [response] {
    user_has_role("Senior Developer")
    response := input.response.body
} {
    user_has_role("Junior Developer")
    project_response_list := input.response.body
    result := [new_item |
        item := project_response_list[_]
        new_item = object.remove(item, "sensitive_info")
    ]
    response := result
}
```

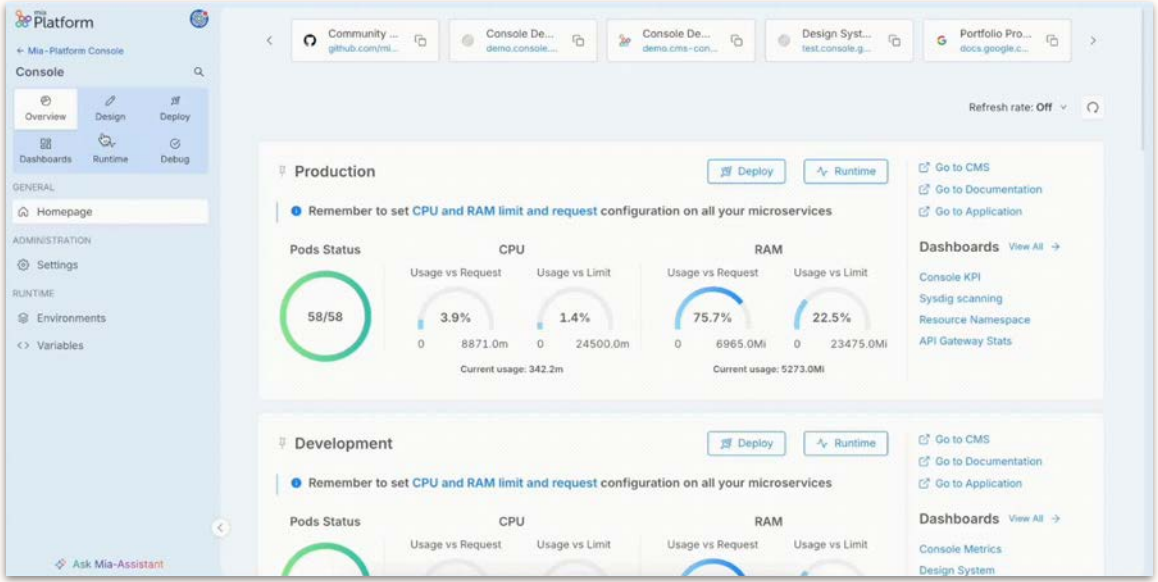
... and you can even define policies that will be executed after api invocation to **change the response** schema before return it to the client

05

Röndd at large scale.

mia Platform

Why Rönd?



30+
microservices

500+
policies

300+
routes

Rönd was designed to protect an **existing application** made up of different services, without needing to change anything in them. Using the **sidecar pattern** and **OpenAPI Specification–compliant** configs, it can be easily added to any application without changing its core.

Why Rönd?

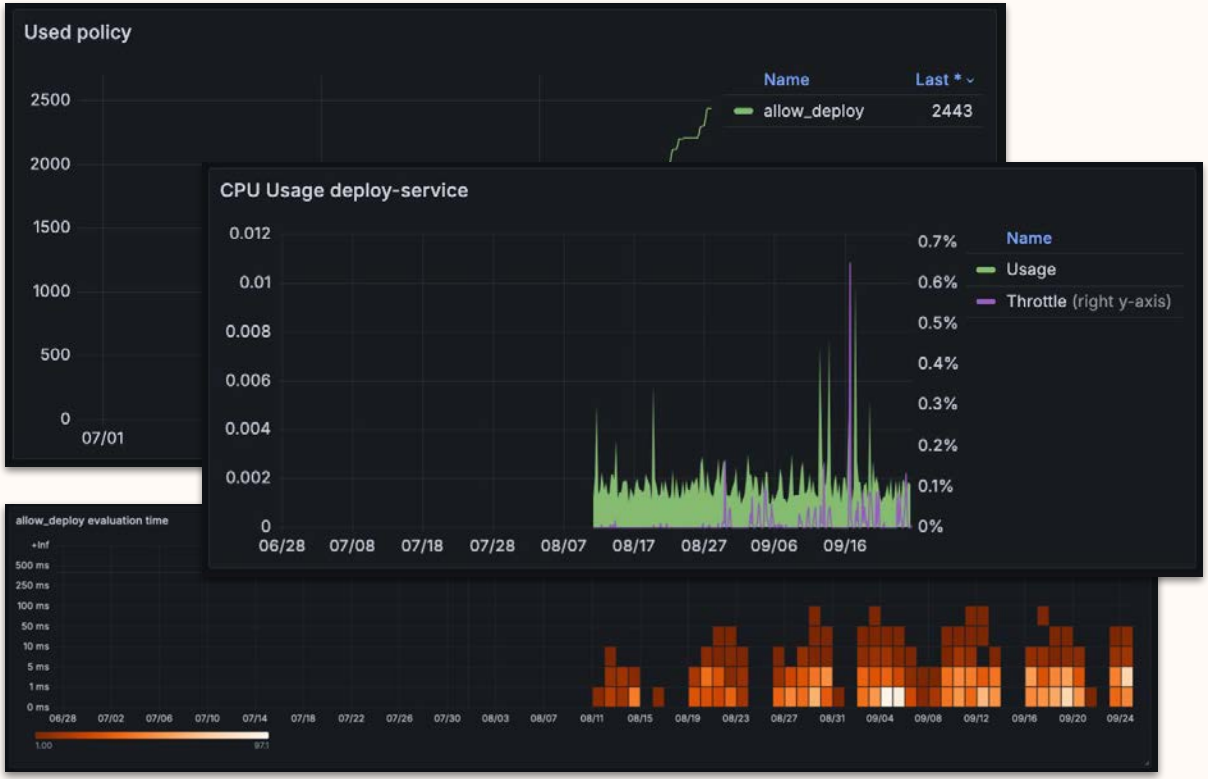
```
allow_deploy {
    projectId := object.get(input, ["request", "pathParams", "projectId"], false)
    projectId
    environment := object.get(input, ["request", "body", "environment"], false)
    environment
    user_has_permission_from_bindings("console.environment.deploy.trigger", concat(":", [projectId, environment]))
}
```

Rönd prepares the **input object**, and in the policy, we check for the presence of **old permissions**, which existed before the introduction of Rond, in the bindings.

```
user_has_permission_from_bindings(permission, resourceId) {
    some i
    binding := input.user.bindings[i]
    binding_has_permission(binding, permission)
    binding.resource.resourceId == resourceId
}
```

mia Platform

Why Rönd?



Rönd expose **policy utilization** and **services monitoring** data to keep track of your authz workflow.

1. Single Policy Usage
2. Policy Evaluation Time
3. Resources utilization per service

Thanks!

Leave your **feedback** or get some in-depth **materials**.

castograziano.com | blog.mia-platform.eu

Casto Graziano ✌🏻