Open Policy Agent: An Authorization Solution

Bella Carter
Software Research and Development
Mentor: Matthew Lala   Manager: Lance Garner

Current State

Microservices: microservice architecture is an approach to software development that breaks down an application into smaller services that each serve a single purpose. Each service runs independently, allowing for updating one service without redeploying every service. For example, Lexmark Cloud Services is an application made up of microservices, including Account Management, Analytics, etc.

Authorization: Authorization dictates who is allowed to do what with what resources. For example, you may want to allow a user access to a service only if they have a secure URL. Authorization code is crucial to the security of the application.

Currently, all authorization code resides within each of the necessary microservices.

Problem

Keeping all authorization code within each service leads to a few tedious issues :

1) Across different services, code is written in different languages, with multiple libraries to maintain it.
2) Any time that a change needs to be made to the authorization code, it requires rebuild and redeploy of the service

Open Policy Agent (OPA)

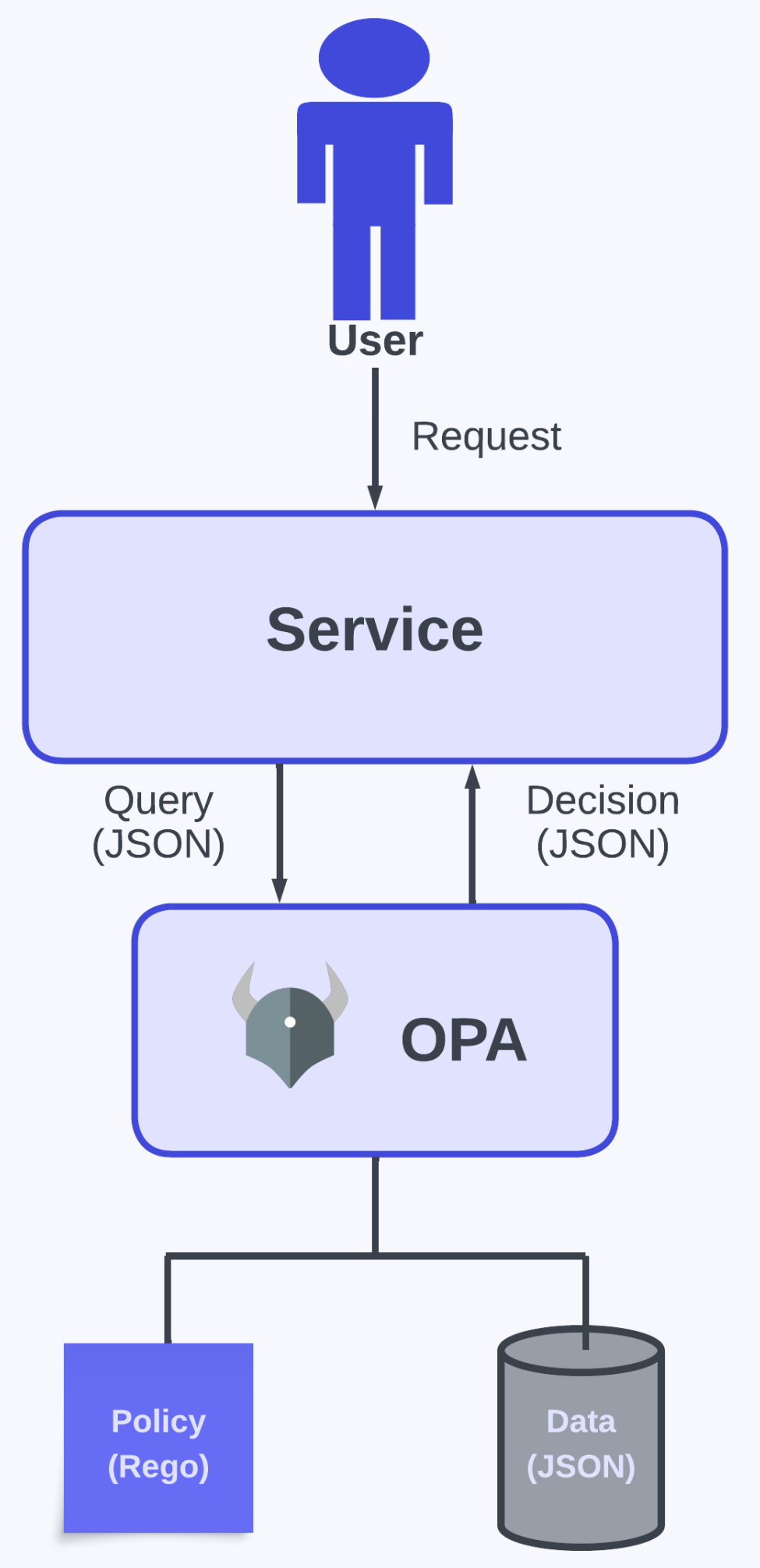
Open Policy Agent (OPA) is a general-purpose policy engine meant to manage and enforce authorization policies. OPA allows you to decouple your authorization code (policies) from your application code (code built into the service). Services offload policy decisions to the OPA engine. OPA evaluates against given policies and returns a decision back to the service.

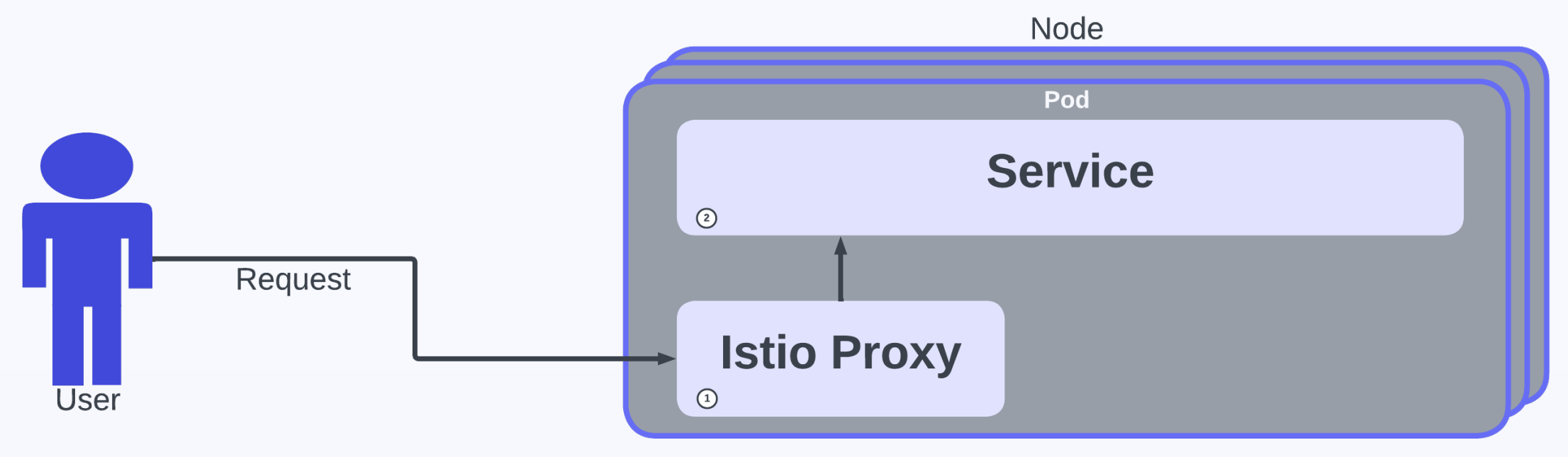
Solution

Using OPA to decouple policy code would keep authorization code separate from the application code. Policy code could be kept in one central locations allowing for easy access and management. Updating policy code wouldn't cause any rebuilding or redeployment of services. In fact, OPA periodically uploads the latest version of policies without any manual intervention.

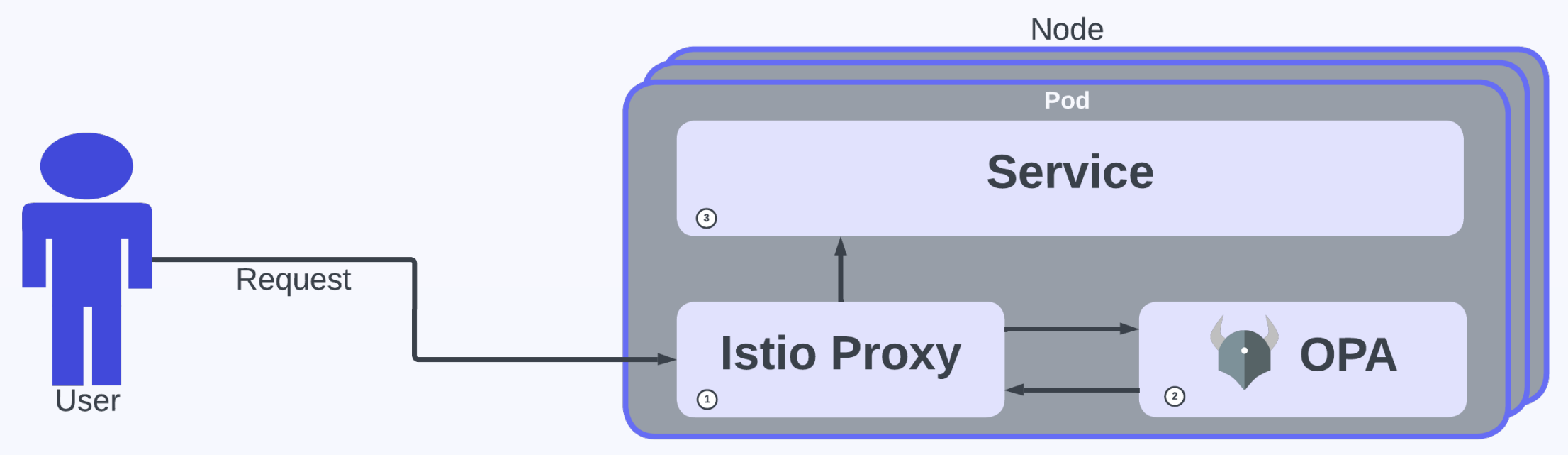
Flowcharts

General OPA flow: Requests move from the user to the service, which offloads the decision to OPA for validation. OPA evaluates the request against given policies and data to make the decision which is sent back to the service for enforcement.





Current authorization flow: All code resides in services. Requests go directly to service.



OPA authorization flow: OPA exists as a side car. Requests are delivered from Istio to query OPA for a decision. If the request is accepted, Istio sends the decision to the service.

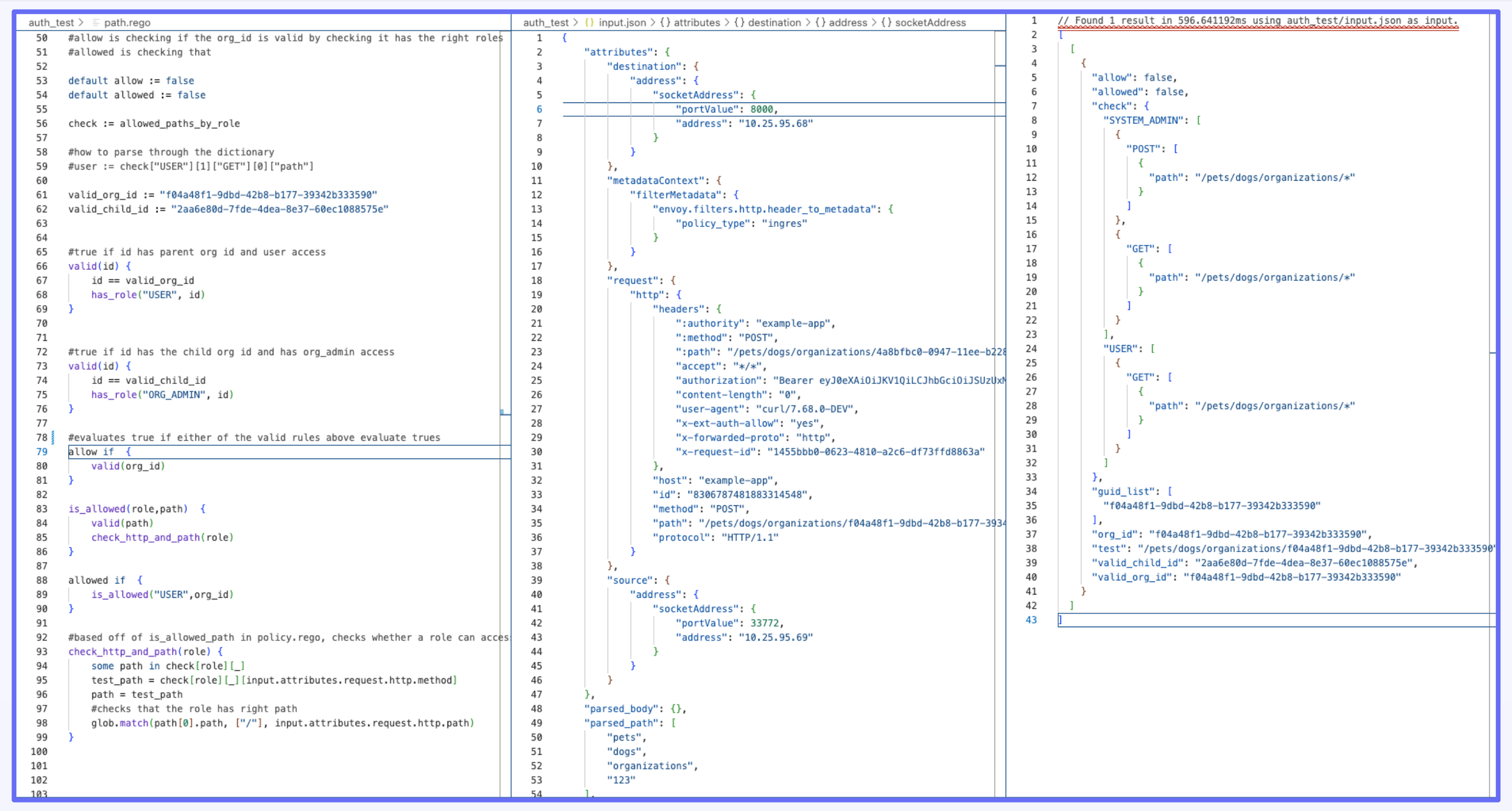
Testing Process

Proof of Concept for OPA:

1. Local set-up and policy development
2. Basic Authorization
   - Java Web Token (JWT) validation
   - Role-Based Access
3. Unit Testing with OPA testing Framework
4. Policy Bundling
   - Structure
   - Upload to Cloud
5. Deployment to local Kubernetes cluster

1

2



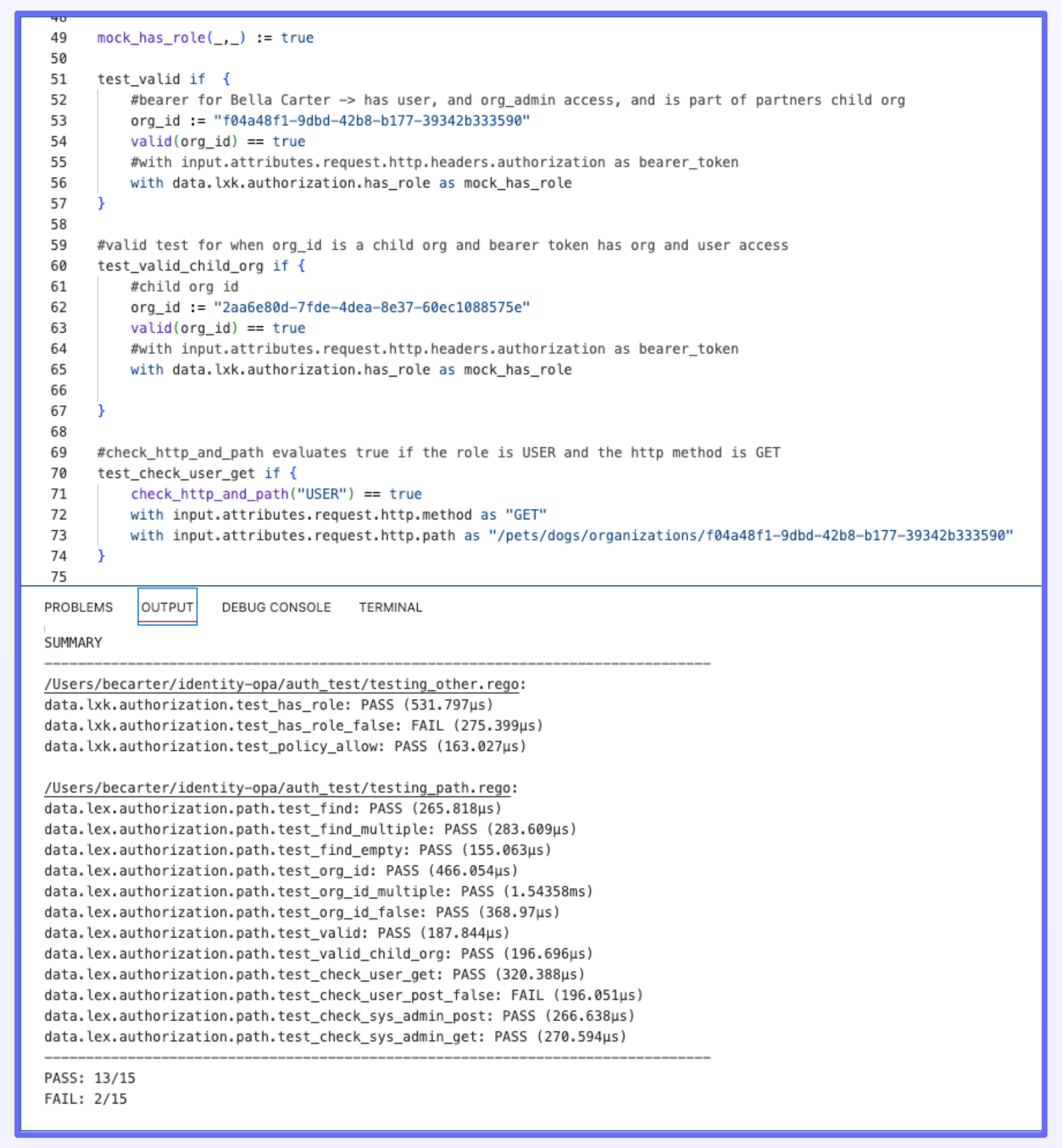
```
auth_test > path.rego
#allow is checking if the org_id is valid by checking it has the right roles
#allowed is checking that

default allow := false
default allowed := false

check := allowed_paths_by_role

#how to parse through the dictionary
#user := check["USER"][1]["GET"][0]["path"]

valid_org_id := "f04a48f1-9dbd-42b8-b177-39342b333590"
valid_child_id := "2aa6e80d-7fde-4dea-8e37-60ec1088575e"


#true if id has parent org id and user access
valid(id) {
    id == valid_org_id
    has_role("USER", id)
}


#true if id has the child org id and has org_admin access
valid(id) {
    id == valid_child_id
    has_role("ORG_ADMIN", id)
}

#evaluates true if either of the valid rules above evaluate trues
allow if  {
    valid(org_id)
}

is_allowed(role,path)  {
    valid(path)
    check_http_and_path(role)
}

allowed if  {
    is_allowed("USER",org_id)
}

#based off of is_allowed_path in policy.rego, checks whether a role can acces
check_http_and_path(role) {
    some path in check[role][_]
    test_path = check[role][_][input.attributes.request.http.method]
    path = test_path
    #checks that the role has right path
    glob.match(path[0].path, ["/"], input.attributes.request.http.path)
}
```

```
auth_test > {} input.json > {} attributes > {} destination > {} address > {} socketAddress
{
    "attributes": {
        "destination": {
            "address": {
                "socketAddress": {
                    "portValue": 8000,
                    "address": "10.25.95.68"
                }
            }
        },
        "metadataContext": {
            "filterMetadata": {
                "envoy.filters.http.header_to_metadata": {
                    "policy_type": "ingres"
                }
            }
        },
        "request": {
            "http": {
                "headers": {
                    ":authority": "example-app",
                    ":method": "POST",
                    ":path": "/pets/dogs/organizations/4a8bfbc0-0947-11ee-b228
                    "accept": "*/*",
                    "authorization": "Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzUxN
                    "content-length": "0",
                    "user-agent": "curl/7.68.0-DEV",
                    "x-ext-auth-allow": "yes",
                    "x-forwarded-proto": "http",
                    "x-request-id": "1455bbb0-0623-4810-a2c6-df73ffd8863a"
                },
                "host": "example-app",
                "id": "8306787481883314548",
                "method": "POST",
                "path": "/pets/dogs/organizations/f04a48f1-9dbd-42b8-b177-3934
                "protocol": "HTTP/1.1"
            }
        },
        "source": {
            "address": {
                "socketAddress": {
                    "portValue": 33772,
                    "address": "10.25.95.69"
                }
            }
        }
    },
    "parsed_body": {},
    "parsed_path": [
        "pets",
        "dogs",
        "organizations",
        "123"
    ],
```

```
// Found 1 result in 596.641192ms using auth_test/input.json as input.
[
  [
    {
      "allow": false,
      "allowed": false,
      "check": {
        "SYSTEM_ADMIN": [
          {
            "POST": [
              {
                "path": "/pets/dogs/organizations/*"
              }
            ]
          },
          {
            "GET": [
              {
                "path": "/pets/dogs/organizations/*"
              }
            ]
          }
        ],
        "USER": [
          {
            "GET": [
              {
                "path": "/pets/dogs/organizations/*"
              }
            ]
          }
        ]
      },
      "guid_list": [
        "f04a48f1-9dbd-42b8-b177-39342b333590"
      ],
      "org_id": "f04a48f1-9dbd-42b8-b177-39342b333590",
      "test": "/pets/dogs/organizations/f04a48f1-9dbd-42b8-b177-39342b333590"
      "valid_child_id": "2aa6e80d-7fde-4dea-8e37-60ec1088575e",
      "valid_org_id": "f04a48f1-9dbd-42b8-b177-39342b333590"
    }
  ]
]
```

Example Rego code, JSON input, and JSON output for a policy that verifies a user's roles against their path

3

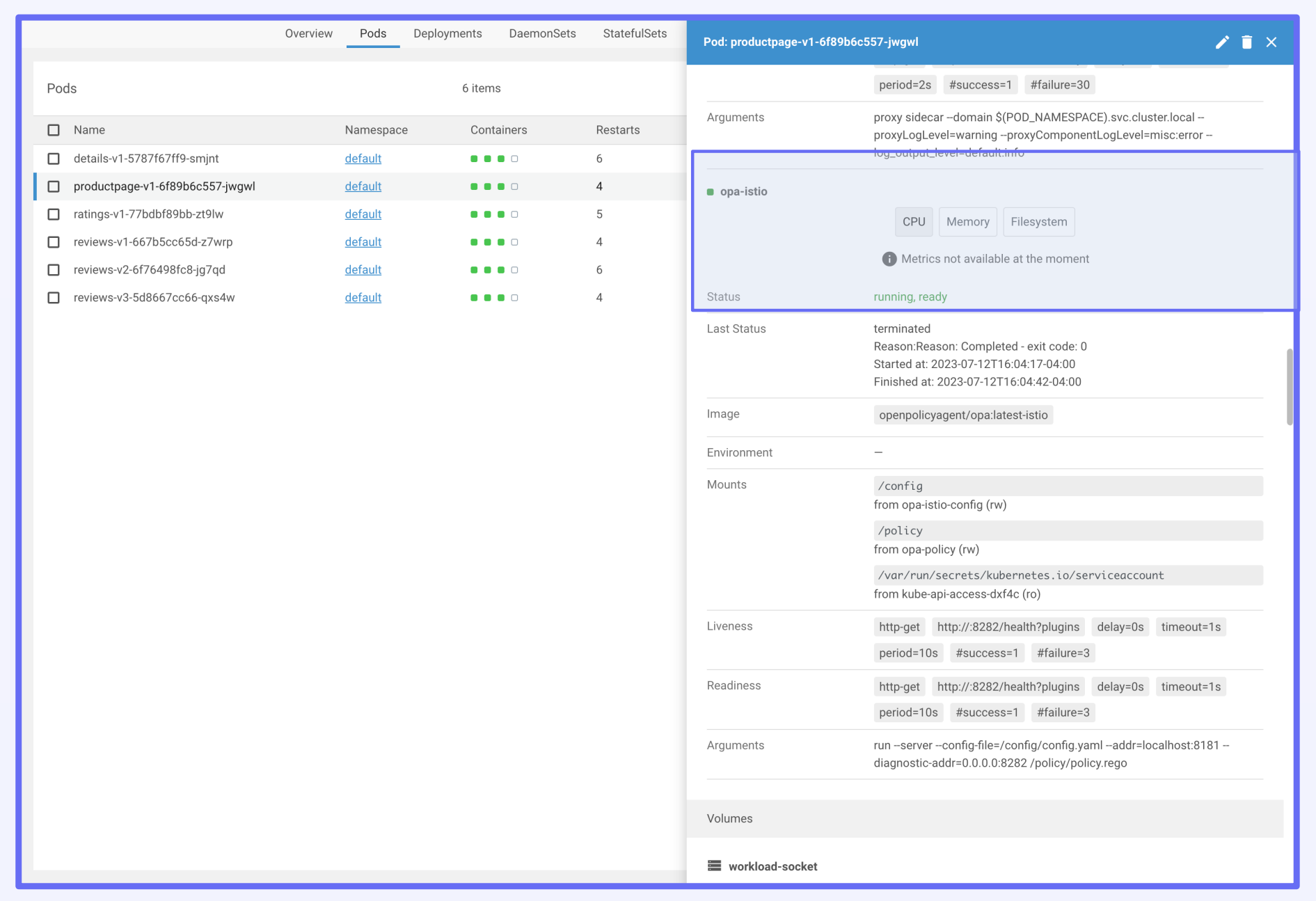


```
mock_has_role(_,_) := true

test_valid if  {
    #bearer for Bella Carter -> has user, and org_admin access, and is part of partners child org
    org_id := "f04a48f1-9dbd-42b8-b177-39342b333590"
    valid(org_id) == true
    #with input.attributes.request.http.headers.authorization as bearer_token
    with data.lxk.authorization.has_role as mock_has_role
}

#valid test for when org_id is a child org and bearer token has org and user access
test_valid_child_org if {
    #child org id
    org_id := "2aa6e80d-7fde-4dea-8e37-60ec1088575e"
    valid(org_id) == true
    #with input.attributes.request.http.headers.authorization as bearer_token
    with data.lxk.authorization.has_role as mock_has_role

}

#check_http_and_path evaluates true if the role is USER and the http method is GET
test_check_user_get if {
    check_http_and_path("USER") == true
    with input.attributes.request.http.method as "GET"
    with input.attributes.request.http.path as "/pets/dogs/organizations/f04a48f1-9dbd-42b8-b177-39342b333590"
}
```

```
SUMMARY
--------------------------------------------------------------------------------
/Users/becarter/identity-opa/auth_test/testing_other.rego:
data.lxk.authorization.test_has_role: PASS (531.797µs)
data.lxk.authorization.test_has_role_false: FAIL (275.399µs)
data.lxk.authorization.test_policy_allow: PASS (163.027µs)

/Users/becarter/identity-opa/auth_test/testing_path.rego:
data.lex.authorization.path.test_find: PASS (265.818µs)
data.lex.authorization.path.test_find_multiple: PASS (283.609µs)
data.lex.authorization.path.test_find_empty: PASS (155.063µs)
data.lex.authorization.path.test_org_id: PASS (466.054µs)
data.lex.authorization.path.test_org_id_multiple: PASS (1.54358ms)
data.lex.authorization.path.test_org_id_false: PASS (368.97µs)
data.lex.authorization.path.test_valid: PASS (187.844µs)
data.lex.authorization.path.test_valid_child_org: PASS (196.696µs)
data.lex.authorization.path.test_check_user_get: PASS (320.388µs)
data.lex.authorization.path.test_check_user_post_false: FAIL (196.051µs)
data.lex.authorization.path.test_check_sys_admin_post: PASS (266.638µs)
data.lex.authorization.path.test_check_sys_admin_get: PASS (270.594µs)
--------------------------------------------------------------------------------
PASS: 13/15
FAIL: 2/15
```

Testing policy that tests some of the functions shown in the figure above. Results from the testing framework shown on the bottom.

5



Screenshot of OpenLens, a visualization tool for Kubernetes. Depicts a cluster of pods that each have the OPA-Istio sidecar deployed and running

Writing Policies with OPA

Open Policy Agent utilizes their own policy language called Rego to define authorization policies. Policies are made up of rules which contain conditions to determine whether request is accepted or not. Policies are run in OPA along with JSON data files to make policy decisions. Data can include external data such as JSON Web Tokens (JWTs) that provide live information with information about the user's attributes. OPA also supports loading static data such as a JSON file with different user roles mapped with what actions each role has access to. To test the policy functionality, policies were written to decode JWTS (a JWT was provided as input), and to check a user's role against the request path. The image on the left shows an example of Rego code, input data, and the output provided by OPA.

Set-up with Kubernetes

There are several different ways to implement/integrate OPA to your software. Because Lexmark Cloud Services is made up of microservices managed by Kubernetes, OPA was implemented as a side-car to run alongside services in Kubernetes pods. The OPA sidecar will automatically inject into every pod. See the flowcharts to visualize the authorization flow with an OPA sidecar.

Policy Bundling

Policy bundling allows OPA to dynamically retrieve the latest versions of policies and data without intervention. Bundles are made up of Rego policy files and JSON data files. Using OPA's build command, bundles are compressed into a single zipped file. To test the feature, a bundle was created and uploaded to Azure Blob Storage, and configuration was changed within the OPA side-car.

Technology Utilized

- ➢ Visual Studio Code
- ➢ Command-Line
- ➢ Minikube
- ➢ Istio
- ➢ Kubernetes
- ➢ Azure Blob Storage
- ➢ Open Policy Agent