AWS Prescriptive Guidance
Multi-tenant SaaS authorization and API access control
Table of Contents

Introduction .................................................................................................................. 1
Targeted business outcomes ......................................................................................... 1
Types of access control ................................................................................................. 3
RBAC ........................................................................................................................... 3
ABAC .......................................................................................................................... 3
RBAC-ABAC hybrid approach ..................................................................................... 3
Access control model comparison .............................................................................. 4
Implementing a PDP ...................................................................................................... 5
Using OPA .................................................................................................................. 5
Rego overview ............................................................................................................ 7
Example 1: Basic ABAC with OPA and Rego ............................................................... 7
Example 2: Multi-tenant access control and user-defined RBAC with OPA and Rego .. 10
Example 3: Multi-tenant access control for RBAC and ABAC with OPA and Rego .... 13
Example 4: UI filtering with OPA and Rego ................................................................ 15
Using a custom policy engine ....................................................................................... 17
Implementing a PEP ..................................................................................................... 18
Requesting an authorization decision ......................................................................... 18
Evaluating an authorization decision ......................................................................... 18
Design models for multi-tenant SaaS architectures ................................................... 20
Centralized PDP with PEPs on APIs .......................................................................... 20
Distributed PDP with PEPs on APIs .......................................................................... 22
Distributed PDP as a library ....................................................................................... 24
Implementation considerations ................................................................................... 25
DevOps, monitoring, and logging ............................................................................... 25
Retrieving external data for a PDP ............................................................................ 25
  Bundling .................................................................................................................. 25
  Replication (pushing data) ..................................................................................... 26
  Dynamic data retrieval ............................................................................................ 26
Using an authorization service for implementation .................................................. 26
Recommendations for tenant isolation and privacy of external data for RBAC ........... 27
Best practices ............................................................................................................ 29
  Select an access control model that works for your application ............................. 29
  Implement a PDP ................................................................................................... 29
  Implement PEPs for every API in your organization ............................................. 29
  Consider using OPA as a policy engine for your PDP ......................................... 29
  Implement a control plane for OPA for DevOps, monitoring, and logging .......... 30
Determine whether external data is required for authorization decisions, and select a model to accommodate it ................................................................................. 30
FAQ .......................................................................................................................... 31
  What is the difference between authorization and authentication? ...................... 31
  Why do I need to consider authorization for my SaaS application? ..................... 31
  Why do I need an access control model? ............................................................... 31
  Is API access control necessary for my application? .......................................... 31
  Why are policy engines or PDPs recommended for authorization? .................... 31
  What is a PEP? ...................................................................................................... 32
  Are there open-source alternatives to OPA? ......................................................... 32
  Do I need to write an authorization service to use OPA, or can I interact with OPA directly? ................................................................. 32
  How do I monitor my OPA agents for uptime and auditing purposes? ................. 32
  Which operating systems and AWS services can I use to run OPA? .................... 32
  Can I run OPA on AWS Lambda? ........................................................................ 32
  How should I decide between a distributed PDP and centralized PDP approach? ................................................................. 33
  Can I use OPA for use cases besides APIs? ......................................................... 33
Next steps .................................................................................................................. 34
Resources .................................................................................................................. 35
References ........................................................................................................................................... 35
Tools .................................................................................................................................................. 35
Partners ............................................................................................................................................. 35
Document history ............................................................................................................................... 36
Glossary .............................................................................................................................................. 37
Modernization terms ......................................................................................................................... 37
Multi-tenant SaaS authorization and API access control

Implementation options and best practices

Tabby Ward and Thomas Davis, Amazon Web Services (AWS)

August 2021 (document history (p. 36))

Authorization and API access control for multi-tenant software as a service (SaaS) applications are complex topics. This complexity is evident when you consider the proliferation of microservice APIs that must be secured and the large number of access conditions that arise from different tenants, user characteristics, and application states. To address these issues effectively, a solution must enforce access control across the many APIs presented by microservices, Backend for Frontend (BFF) layers, and other components of a multi-tenant SaaS application. This pervasive approach must be coupled with a flexible enforcement paradigm that can make complicated access decisions based on many factors and attributes.

Traditionally, API access control and authorization were handled by custom logic in the application code. This approach was error prone and not secure, because developers who had access to this code could accidentally or deliberately change authorization logic, which could result in unauthorized access. Furthermore, API access control was generally unnecessary, because there weren’t as many APIs to secure. The paradigm shift in application design to favor microservices and service-oriented architectures has increased the number of APIs that must use a form of authorization and access control. In addition, the need to maintain tenant-based access in a multi-tenant SaaS application can become very complex, and siloed or highly inefficient models are often used to preserve this tenancy.

The best practices outlined in this guide provide several benefits:

- Authorization logic can be centralized and written in a high-level declarative language that is not specific to any programming language.
- Authorization logic is abstracted from the application code and can be applied idempotently to all APIs in an application.
- The abstraction prevents accidental changes to authorization logic and makes its integration into a SaaS application consistent and simple.
- The abstraction prevents the need to write custom authorization logic for each API endpoint.
- The approach outlined in this guide supports the use of multiple access control paradigms depending on the requirements of an organization.
- This authorization and access control approach provides a simple and straightforward way to maintain tenant data isolation at the API layer in a SaaS application.

Targeted business outcomes

This prescriptive guidance describes idempotent design patterns for authorization and API access controls that can be implemented for multi-tenant SaaS applications. This guidance is intended for any team that develops applications with complex authorization requirements or strict API access control needs. The architecture details the creation of a policy decision point (PDP) or policy engine with the Open Policy Agent (OPA) and the integration of policy enforcement points (PEP) in APIs. The guide also
discusses making access decisions based upon an attribute-based access control (ABAC) model or role-based access control (RBAC) model, or a combination of both models.

We recommend that you use the design patterns and concepts provided in this guide to inform and standardize your implementation of authorization and API access control in multi-tenant SaaS applications. This guidance helps achieve the following business outcomes:

• **Standardized API authorization architecture for multi-tenant SaaS applications** – This architecture is accomplished through PDPs, PEPs, policy engines, and OPA. These concepts and technologies help maintain tenant isolation in a multi-tenant SaaS application and provide a holistic approach to API authorization for the application.

• **Decoupling of authorization logic from applications** – Authorization logic, when embedded in application code or implemented through an ad hoc enforcement mechanism, can be subject to accidental or malicious changes that cause unintentional cross-tenant data access or other security breaches. To help mitigate these possibilities, you can use policy engines such as OPA to separate authorization decisions from application code and to enforce consistent policies across an application. Policies can be maintained centrally in a high-level declarative language, which makes maintaining authorization logic far simpler than when you embed policies in multiple sections of application code. This approach also ensures that updates are applied consistently.

• **Flexible approach to access control models** – Role-based access control (RBAC), attribute-based access control (ABAC), or a combination of both models are all valid approaches to access control. These models attempt to meet the authorization requirements for a business by using different approaches. This guide compares and contrasts these models to help you select a model that works for your organization. The guide also discusses how these models apply to policy engines, and OPA in particular, in detail. The architectures discussed in this guide enable either or both models to be adopted successfully.

• **Strict API access control** – This guide provides a method to secure APIs consistently and pervasively in an application with minimal effort. This is particularly valuable for service-oriented or microservice application architectures that generally use a large number of APIs to facilitate intra-application communications. Strict API access control helps increase the security of an application and makes it less vulnerable to attack or exploitation.
Types of access control

You can use two broadly defined models to implement access control: role-based access control (RBAC) and attribute-based access control (ABAC). Each model has advantages and disadvantages, which are briefly discussed in this section. The model you should use depends on your specific use case. The architecture discussed in this guide supports both models.

RBAC

Role-based access control (RBAC) determines access to resources based on a role that usually aligns with business logic. Permissions are associated with the role as appropriate. For instance, a marketing role would authorize a user to perform marketing activities within a restricted system. This is a relatively simple access control model to implement because it aligns well to easily recognizable business logic.

The RBAC model is less effective when:

- You have unique users whose responsibilities encompass several roles.
- You have complex business logic that makes roles difficult to define.
- Scaling up to a large size requires constant administration and mapping of permissions to new and existing roles.
- Authorizations are based on dynamic parameters.

ABAC

Attribute-based access control (ABAC) determines access to resources based on attributes. Attributes can be associated with a user, resource, environment, or even application state. Your policies or rules reference attributes and can use basic Boolean logic to determine whether a user is permitted to perform an action. Here’s a basic example of permissions:

In the payments system, all users in the Finance department are allowed to process payments at the API endpoint /payments during business hours.

Membership in the Finance department is a user attribute that determines access to /payments. There is also a resource attribute associated with the /payments API endpoint that permits access only during business hours. In ABAC, whether or not a user can process a payment is determined by a policy that includes the Finance department membership as a user attribute, and the time as a resource attribute of /payments.

The ABAC model is very flexible in allowing dynamic, contextual, and granular authorization decisions. However, the ABAC model is difficult to implement initially. Defining rules and policies as well as enumerating attributes for all relevant access vectors require a significant upfront investment to implement.

RBAC-ABAC hybrid approach

Combining RBAC and ABAC can provide some of the advantages of both models. RBAC, being aligned so closely to business logic, is simpler to implement than ABAC. To provide an additional layer of granularity when making authorization decisions, you can combine ABAC with RBAC. This hybrid
approach determines access by combining a user’s role (and its assigned permissions) with additional attributes to make access decisions. Using both models enables simple administration and assignment of permissions while also permitting increased flexibility and granularity pertaining to authorization decisions.

**Access control model comparison**

The following table compares the three access control models discussed previously. This comparison is meant to be informative and high-level. Using an access model in a specific situation might not necessarily correlate to the comparisons made in this table.

<table>
<thead>
<tr>
<th>Factor</th>
<th>RBAC</th>
<th>ABAC</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexibility</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Simplicity</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Granularity</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Dynamic decisions and rules</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Context-aware</td>
<td>No</td>
<td>Yes</td>
<td>Somewhat</td>
</tr>
<tr>
<td>Implementation effort</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
</tr>
</tbody>
</table>
Implementing a PDP

The policy decision point (PDP) can be characterized as a policy or rules engine. This component is responsible for applying policies or rules and returning a decision on whether a particular access is permitted. A PDP can function with role-based access control (RBAC) and attribute-based access control (ABAC) models; however, a PDP is a requirement for ABAC. A PDP allows authorization logic in application code to be offloaded to a separate system. This can simplify application code. It also provides an easy-to-use idempotent interface for making authorization decisions for APIs, microservices, Backend for Frontend (BFF) layers, or any other application component.

The following sections discuss two methods for implementing a PDP: Open Policy Agent (OPA) and custom solutions. However, this is not an exhaustive list.

**PDP implementation methods:**
- Using OPA (p. 5)
- Using a custom policy engine (p. 17)

**Using OPA**

The preferred method for implementing a PDP is to use the Open Policy Agent (OPA). OPA is an open-source, general-purpose policy engine. OPA has many use cases, but the use case relevant for PDP implementation is its ability to decouple authorization logic from an application. This is called policy decoupling.

OPA is useful in implementing a PDP for several reasons. It uses a high-level declarative language called Rego to draft policies and rules. These policies and rules exist separately from an application and can render authorization decisions without any application-specific logic. OPA also exposes a RESTful API to make retrieving authorization decisions simple and straightforward. To make an authorization decision, an application queries OPA with JSON input, and OPA evaluates the input against the specified policies to return an access decision in JSON. OPA is also capable of importing external data that might be relevant in making an authorization decision.
OPA has several advantages over custom policy engines:

- OPA and its policy evaluation with Rego provide a flexible, pre-built policy engine that requires only the insertion of policies and any data necessary to make authorization decisions. This policy evaluation logic would have to be recreated in a custom policy engine solution.
- OPA simplifies authorization logic by having policies written in a declarative language. You can modify and administer these policies and rules independently of any application code, without application development skills.
- OPA exposes a RESTful API, which simplifies integration with policy enforcement points (PEPs).
- OPA provides built-in support for validating and decoding JSON Web Tokens (JWTs).
- OPA is a recognized authorization standard, which means that documentation and examples are plentiful if you need assistance or research to solve a particular problem.
- Adopting an authorization standard such as OPA allows policies written in Rego to be shared across teams regardless of the programming language used by a team's application.

There are two things that OPA doesn't provide automatically:

- OPA doesn't have a robust control plane for updating and managing policies. OPA does provide some basic patterns for implementing policy updates, monitoring, and log aggregation by exposing a management API, but integration with this API must be handled by the OPA user. As a best practice, you should use a continuous integration and continuous deployment (CI/CD) pipeline to administer, modify, and track policy versions and manage policies in OPA.
• OPA can’t retrieve data from external sources by default. An external source of data for an authorization decision could be a database that holds user attributes. There is some flexibility in how external data is provided to OPA—it can be cached locally in advance or retrieved dynamically from an API when an authorization decision is requested—but getting this information is not something OPA can do on your behalf.

In this section:
• Rego overview (p. 7)
• Example 1: Basic ABAC with OPA and Rego (p. 7)
• Example 2: Multi-tenant access control and user-defined RBAC with OPA and Rego (p. 10)
• Example 3: Multi-tenant access control for RBAC and ABAC with OPA and Rego (p. 13)
• Example 4: UI filtering with OPA and Rego (p. 15)

Rego overview

Rego is a general-purpose policy language, which means that it works for any layer of the stack and any domain. The primary purpose of Rego is to accept JSON/YAML inputs and data that are evaluated to make policy-enabled decisions about infrastructure resources, identities, and operations. Rego enables you to write policy about any layer of a stack or domain without requiring a change or extension of the language. Here are some examples of decisions that Rego can make:

• Is this API request allowed or denied?
• What is the hostname of the backup server for this application?
• What is the risk score for this proposed infrastructure change?
• Which clusters should this container be deployed to for high availability?
• What routing information should be used for this microservice?

To answer these questions, Rego employs a basic philosophy about how these decisions can be made. The two key tenets when drafting policy in Rego are:

• Every resource, identity, or operation can be represented as JSON or YAML data.
• Policy is logic that is applied to data.

Rego helps software systems make authorization decisions by defining logic about how inputs of JSON/YAML data are evaluated. Programming languages such as C, Java, Go, and Python are the usual solution to this problem, but Rego was designed to focus on the data and inputs that represent your system, and the logic for making policy decisions with this information.

Example 1: Basic ABAC with OPA and Rego

This section describes a scenario where OPA is used to make access decisions about which users are allowed to access information in a fictional Payroll microservice. Rego code snippets are provided to demonstrate how you can use Rego to render access control decisions. These examples are neither exhaustive nor a full exploration of Rego and OPA capabilities. For a more thorough overview of Rego, we recommend that you consult the Rego documentation on the OPA website.
Basic OPA rules example

In the previous diagram, one of the access control rules enforced by OPA for the Payroll microservice is:

Employees can read their own salary.

If Bob tries to access the Payroll microservice to see his own salary, the Payroll microservice can redirect the API call to the OPA RESTful API to make an access decision. The Payroll service queries OPA for a decision with the following JSON input:

```json
{
    "user": "bob",
    "method": "GET"
}
```
Example 1: Basic ABAC with OPA and Rego

```json
"path": ["getSalary", "bob"]
}
```

OPA selects a policy or policies based on the query. In this case, the following policy, which is written in Rego, evaluates the JSON input.

```rego
default allow = false
allow = true {
  input.method == "GET"
  input.path = ["getSalary", user]
  input.user == user
}
```

This policy denies access by default. It then evaluates the input in the query by binding it to the global variable input. The dot operator is used with this variable to access the variable’s values. The Rego rule allow returns true if the expressions in the rule are also true. The Rego rule verifies that the method in the input is equal to GET. It then verifies that the first element in the list path is getSalary before assigning the second element in the list to the variable user. Lastly, it checks that the path being accessed is /getSalary/bob by checking that the user making the request, input.user, matches the user variable. The rule allow applies if-then logic to return a Boolean value, as shown in the output:

```json
{
  "allow": true
}
```

Partial rule using external data

To demonstrate additional OPA capabilities, you can add requirements to the access rule you are enforcing. Let’s assume that you want to enforce this access control requirement in the context of the previous illustration:

*Employees can read the salary of anyone who reports to them.*

In this example, OPA has access to external data that can be imported to help make an access decision:

```json
"managers": {
  "bob": ["dave", "john"],
  "carol": ["alice"]
}
```

You can generate an arbitrary JSON response by creating a partial rule in OPA, which returns a set of values instead of a fixed response. This is an example of a partial rule:

```rego
direct_report[user_ids] {
  user_ids = data.managers[input.user][_]
}
```

This rule returns a set of all users that report to the value of input.user, which, in this case, is bob. The [_] construct in the rule is used to iterate over the values of the set. This is the output of the rule:

```json
{
  "direct_report": [
    "dave",
    "john"
  ]
}
```
Retrieving this information can help determine whether a user is a direct report of a manager. For some applications, returning dynamic JSON is preferable to returning a simple Boolean response.

Putting it all together

The last access requirement is more complex than the first two because it combines the conditions specified in both requirements:

Employees can read their own salary and the salary of anyone who reports to them.

To fulfill this requirement, you can use this Rego policy:

```
default allow = false
allow = true {
  input.method == "GET"
  input.path = ["getSalary", user]
  input.user == user
}
allow = true {
  input.method == "GET"
  input.path = ["getSalary", user]
  managers := data.managers[input.user][_]
  contains(managers, user)
}
```

The first rule in the policy allows access for any user who tries to see their own salary information, as discussed previously. Having two rules with the same name, allow, functions as a logical or operator in Rego. The second rule retrieves the list of all direct reports associated with input.user (from the data in the previous diagram) and assigns this list to the managers variable. Lastly, the rule checks whether the user who is trying to see their salary is a direct report of input.user by verifying that their name is contained in the managers variable.

The examples in this section are very basic and do not provide a complete or thorough exploration of the capabilities of Rego and OPA. For more information, review the OPA documentation, see the OPA GitHub README file, and experiment in the Rego playground.

Example 2: Multi-tenant access control and user-defined RBAC with OPA and Rego

This example uses OPA and Rego to demonstrate how access control can be implemented on an API for a multi-tenant application with custom roles defined by tenant users. It also demonstrates how access can be restricted based on a tenant. This model shows how OPA can make granular permission decisions based on information that is provided in a high-level role.
The roles for the tenants are stored in external data (RBAC data) that is used to make access decisions for OPA:

```json
{
    "roles": {
        "tenant_a": {
            "all_access_role": ["viewData", "updateData"],
        },
        "tenant_b": {
            "update_data_role": ["updateData"],
            "view_data_role": ["viewData"
        }
    }
}
```

These roles, when defined by a tenant user, should be stored in an external data source or an identity provider (IdP) that can act as a source of truth when mapping tenant-defined roles to permissions and to the tenant itself.

This example uses two policies in OPA to make authorization decisions and to examine how these policies enforce tenant isolation. These policies use the RBAC data defined earlier.

```plaintext
default allowViewData = false
```
Example 2: Multi-tenant access control and user-defined RBAC with OPA and Rego

```plaintext
allowViewData = true {
    input.method == "GET"
    input.path = ["viewData", tenant_id]
    input.tenant_id == tenant_id
    role_permissions := data.roles[input.tenant_id][input.role][_] contains(role_permissions, "viewData")
}
```

To show how this rule will function, consider an OPA query that has the following input:

```plaintext
{
    "tenant_id": "tenant_a",
    "role": "all_access_role",
    "path": ["viewData", "tenant_a"],
    "method": "GET"
}
```

An authorization decision for this API call is made as follows, by combining the RBAC data, the OPA policies, and the OPA query input:

1. A user from Tenant A makes an API call to /viewData/tenant_a.
2. The Data microservice receives the call and queries the allowViewData rule, passing the input shown in the OPA query input example.
3. OPA uses the queried rule in OPA policies to evaluate the input provided. OPA also uses the data from RBAC data to evaluate the input. OPA does the following:
   a. Verifies that the method used to make the API call is GET.
   b. Verifies that the path requested is viewData.
   c. Checks that the tenant_id in the path is equal to the input.tenant_id associated with the user. This ensures that tenant isolation is maintained. Another tenant, even with an identical role, is unable to be authorized in making this API call.
   d. Pulls a list of role permissions from the roles' external data and assigns them to the variable role_permissions. This list is retrieved by using the tenant-defined role that is associated with the user in input.role.
   e. Checks role_permissions to see whether it contains the permission viewData.
4. OPA returns the following decision to the Data microservice:

```plaintext
{
    "allowViewData": true
}
```

This process shows how RBAC and tenant awareness can contribute to making an authorization decision with OPA. To further illustrate this point, consider an API call to /viewData/tenant_b with the following query input:

```plaintext
{
    "tenant_id": "tenant_b",
    "role": "view_data_role",
    "path": ["viewData", "tenant_b"],
    "method": "GET"
}
```

This rule would return the same output as OPA query input although it is for a different tenant who has a different role. This is because this call is for /tenant_b and the view_data_role in RBAC data still has the viewData permission associated with it. To enforce the same type of access control for /updateData, you can use a similar OPA rule:
Example 3: Multi-tenant access control for RBAC and ABAC with OPA and Rego

default allowUpdateData = false
allowUpdateData = true {
    input.method == "POST"
    input.path = ["updateData", tenant_id]
    input.tenant_id == tenant_id
    role_permissions := data.roles[input.tenant_id][input.role][_
     contains(role_permissions, "updateData")
}

This rule is functionally the same as the allowViewData rule, but it verifies a different path and input method. The rule still ensures tenant isolation and checks that the tenant-defined role grants the API caller permission. To see how this might be enforced, examine the following query input for an API call to /updateData/tenant_b:

```json
{
    "tenant_id": "tenant_b",
    "role": "view_data_role",
    "path": ["updateData", "tenant_b"],
    "method": "POST"
}
```

This query input, when evaluated with the allowUpdateData rule, returns the following authorization decision:

```json
{
    "allowUpdateData": false
}
```

This call will not be authorized. Although the API caller is associated with the correct tenant_id and is calling the API by using an approved method, the input.role is the tenant-defined view_data_role. The view_data_role doesn't have the updateData permission; therefore, the call to /updateData is unauthorized. This call would have been successful for a tenant_b user who has the update_data_role.

**Example 3: Multi-tenant access control for RBAC and ABAC with OPA and Rego**

To enhance the RBAC example in the previous section, you can add attributes to users.
This example includes the same roles from the previous example, but adds the user attribute `account_lockout_flag`. This is a user-specific attribute that isn't associated with any particular role. You can use the same RBAC external data that you used previously for this example:

```json
{
    "roles": {
        "tenant_a": {
            "all_access_role": ["viewData", "updateData"]
        },
        "tenant_b": {
            "update_data_role": ["updateData"],
            "view_data_role": ["viewData"]
        }
    }
}
```

The `account_lockout_flag` user attribute can be passed to the Data service as part of the input to an OPA query for `/viewData/tenant_a` for the user Bob:

```json
{
    "tenant_id": "tenant_a",
    "role": "all_access_role",
    "path": ["viewData", "tenant_a"],
    "method": "GET",
    "account_lockout_flag": "true"
}
```
The rule that is queried for the access decision is similar to the previous examples, but includes an additional line to check for the account_lockout_flag attribute:

```plaintext
default allowViewData = false
allowViewData = true {
    input.method == "GET"
    input.path = ["viewData", tenant_id]
    input.tenant_id == tenant_id
    role_permissions := data.roles[input.tenant_id][input.role]_
    contains(role_permissions, "viewData")
    input.account_lockout_flag == "false"
}
```

This query returns an authorization decision of false. This is because the account_lockout_flag attribute is true for Bob, and the Rego rule allowViewData denies access although Bob has the correct role and tenant.

**Example 4: UI filtering with OPA and Rego**

The flexibility of OPA and Rego supports the ability to filter UI elements. The following example demonstrates how an OPA partial rule can make authorization decisions about which elements should be displayed in a UI with RBAC. This method is one of many different ways you can filter UI elements with OPA.
In this example, a single-page web application has four buttons. Let's say that you want to filter Bob's, Shirley's, and Alice's UI so that they can only see the buttons that correspond to their roles. When the UI receives a request from the user, it queries an OPA partial rule to determine which buttons should be displayed in the UI. The query passes the following as input to OPA when Bob (with the role viewer) makes a request to the UI:

```
{
  "role": "viewer"
}
```

OPA uses external data structured for RBAC to make an access decision:

```
{
  "roles": {
    "viewer": ["viewUsersButton", "viewDataButton"],
    "dataViewOnly": ["viewDataButton"],
    "admin": ["viewUsersButton", "viewDataButton", "updateUsersButton",
              "updateDataButton"]
  }
}
```
The OPA partial rule uses both the external data and the input to produce a set of buttons that a user can view on the UI:

```plaintext
ui_buttons[buttons] {
  buttons := data.roles[input.role][_]
}
```

In the partial rule, OPA uses the `input.role` specified as part of the query to determine which buttons should be displayed. Bob has the role `viewer`, and the external data specifies that viewers can see two buttons: `viewUsersButton` and `viewDataButton`. Therefore, the output of this rule for Bob (and for any other users who have a viewer role) is as follows:

```plaintext
{
  "ui_buttons": [
    "viewDataButton",
    "viewUsersButton"
  ]
}
```

The output for Shirley, who has the `dataViewOnly` role, would contain a single button: `viewDataButton`. The output for Alice, who has the `admin` role, would contain all buttons. These responses are returned to the UI when OPA is queried for `ui_buttons`. The UI can use this response to then hide or display buttons accordingly.

### Using a custom policy engine

An alternative method for implementing a PDP is to create a custom policy engine. The goal of this policy engine is to decouple authorization logic from an application. The custom policy engine is responsible for making authorization decisions, similar to OPA, to achieve policy decoupling. The primary difference between this solution and OPA is that the logic for writing and evaluating policies is custom-built. Any interactions with the engine must be exposed through an API or some other method to enable authorization decisions to reach an application. You can write a custom policy engine in any programming language or use other mechanisms for policy evaluation such as the Common Expression Language (CEL).
Implementing a PEP

A policy enforcement point (PEP) is responsible for receiving authorization requests that are sent to the policy decision point (PDP) for evaluation. A PEP can be anywhere in an application where data and resources must be protected, or where authorization logic is applied. PEPs are relatively simple compared with PDPs. A PEP is responsible only for requesting and evaluating an authorization decision and doesn’t require any authorization logic. PEPs, unlike PDPs, cannot be centralized in a SaaS application. This is because authorization and access control are required to be implemented throughout an application and its access points. PEPs can be applied to APIs, microservices, Backend for Frontend (BFF) layers, or any point in the application where access control is desired or required. Making PEPs pervasive in an application ensures that authorization is verified often and independently at multiple points.

To implement a PEP, the first step is to determine where access control enforcement should occur in an application. Consider this principle when deciding where PEPs should be integrated into your application:

*If an application exposes an API, there should be authorization and access control on that API.*

This is because in a microservices-oriented or service-oriented architecture, APIs serve as separators between different application functions. It makes sense to include access control as logical checkpoints between application functions. We strongly recommend that you include PEPs as a prerequisite for access to each API in a SaaS application. It is also possible to integrate authorization at other points in an application. In monolithic applications, it might be necessary to have PEPs integrated within the logic of the application itself. There is no one-size-fits-all solution to where PEPs should be included, but consider using the API principle as starting point.

Requesting an authorization decision

A PEP must request an authorization decision from the PDP. The request can take several forms. The easiest and most accessible method for requesting an authorization decision is to send an authorization request or query (using OPA terms) to a RESTful API that is exposed by the PDP. This is the suggested method of integrating PEPs with a PDP, because it is a familiar pattern for exposing functionality to multiple services in an application. The only function of a PEP in this pattern is to forward the information that the authorization request or query needs. This can be as simple as forwarding a request received by an API as input to the PDP. There are other methods for creating PEPs. For example, you can integrate an OPA PDP locally with an application written in the Go programming language as a library instead of using an API.

Evaluating an authorization decision

PEPs need to include logic to evaluate the results of an authorization decision. When PDPs are exposed as APIs, the authorization decision is likely in JSON format and returned by an API call. The PEP must evaluate this JSON code to determine whether the action being taken is authorized. For example, if a PDP is designed to provide a Boolean allow or deny authorization decision, the PEP might simply check this value, and then return HTTP status code 200 for allow and HTTP status code 403 for deny. This pattern of incorporating a PEP as a prerequisite for accessing an API is an easily implemented and highly effective pattern for implementing access control across a SaaS application. In more complicated scenarios, the PEP might be responsible for evaluating arbitrary JSON code returned by the PDP. The PEP must be written to include whatever logic is necessary to interpret the authorization decision that the PDP returns. Because a PEP is likely to be implemented in many different places in your application, we recommend that you package your PEP code as a reusable library or artifact in your programming
language of choice. This way, your PEP can be easily integrated at any point in your application with minimal rework.
Design models for multi-tenant SaaS architectures

There are many ways to implement API access control and authorization. This guide focuses on three design models that are effective for multi-tenant SaaS architectures. These designs serve as a high-level reference for the implementation of policy decision points (PDPs) and policy enforcement points (PEPs), to form a cohesive and ubiquitous authorization model for applications.

Design models:
- Centralized PDP with PEPs on APIs (p. 20)
- Distributed PDP with PEPs on APIs (p. 22)
- Distributed PDP as a library (p. 24)

Centralized PDP with PEPs on APIs

The centralized policy decision point (PDP) with policy enforcement points (PEPs) on APIs model follows industry best practices to create an effective and easily maintained system for API access control and authorization. This approach supports several key principles:

- Authorization and API access control are applied at multiple points in the application.
- Authorization logic is independent of the application.
- Access control decisions are centralized.

This model uses a centralized PDP to make authorization decisions. PEPs are implemented at all APIs to make authorization requests to the PDP. The following diagram shows how you can implement this model in a hypothetical multi-tenant SaaS application.
AWS Prescriptive Guidance Multi-tenant
SaaS authorization and API access control
Centralized PDP with PEPs on APIs

Application flow:

1. An authenticated user with a JWT token generates an HTTP request to Amazon CloudFront.
2. CloudFront forwards the request to Amazon API Gateway configured as a CloudFront origin.
3. An API Gateway Lambda authorizer is called to verify the JWT token.
4. Microservices respond to request.

Authorization and API access control flow:
In this architecture, PEPs request authorization decisions at the service endpoints for CloudFront and API Gateway, and for each microservice. The authorization decision is made by an authorization service (the PDP) with an OPA sidecar. You can operate this authorization service as a container or as a traditional server instance. The OPA sidecar exposes its RESTful API locally so the API is accessible only to the authorization service. The authorization service exposes a separate API that is available to PEPs. Having the authorization service act as an intermediary between PEPs and OPA allows for the insertion of any transformation logic between PEPs and OPA that may be necessary—for example, when the authorization request from a PEP doesn’t conform to the query input expected by OPA.

You can also use this architecture with custom policy engines. However, any advantages gained from OPA must be replaced with logic provided by the custom policy engine.

A centralized PDP with PEPs on APIs provides an easy option to create a robust authorization system for APIs. It's simple to implement and also provides an easy-to-use idempotent interface for making authorization decisions for APIs, microservices, Backend for Frontend (BFF) layers, or other application components. However, this approach might create too much latency in your application, because authorization decisions require calling a separate API. If network latency is a problem, you might consider a distributed PDP.

Distributed PDP with PEPs on APIs

The distributed policy decision point (PDP) with policy enforcement points (PEPs) on APIs model follows industry best practices to create an effective system for API access control and authorization. As with the centralized PDP with PEPs on APIs model, this approach supports the following key principles:

- Authorization and API access control are applied at multiple points in the application.
- Authorization logic is independent of the application.
- Access control decisions are centralized.

You might wonder why access control decisions are centralized when the PDP is distributed. Although the PDP might exist in multiple places in an application, it must use the same authorization logic to make access control decisions. All PDPs provide the same access control decisions given the same inputs. PEPs
are implemented at all APIs to make authorization requests to the PDP. The following diagram shows how this distributed model can be implemented in a hypothetical multi-tenant SaaS application.

**Application flow:**

1. An authenticated user with a JWT token generates an HTTP request to Amazon CloudFront.
2. CloudFront forwards the request to Amazon API Gateway configured as a CloudFront origin.
3. An API Gateway Lambda authorizer is called to verify the JWT token.
4. Microservices respond to request.

**Authorization and API access control flow:**
The PEP calls the authorization service and passes request data, including any JWT tokens.

The authorization service (PDP) takes the request data and queries an OPA agent REST API running as a sidecar, with the request data serving as an input to the query.

OPA evaluates the input based on the relevant policies specified by the query. Data is imported to make an authorization decision if necessary.

OPA returns a decision to the authorization service.

The authorization decision is returned to the PEP and evaluated.

In this architecture, PEPs request authorization decisions at the service endpoints for CloudFront and API Gateway, and for each microservice. The authorization decision for microservices is made by an authorization service (the PDP) that operates as a sidecar with the application component. This model is possible for microservices (or services) that run on containers or Amazon Elastic Compute Cloud (Amazon EC2) instances. Authorization decisions for services such as API Gateway and CloudFront would still require contacting an external authorization service. Regardless, the authorization service exposes an API that is available to PEPs. Having the authorization service act as an intermediary between PEPs and OPA allows for the insertion of any transformation logic between PEPs and OPA that might be necessary—for example, when the authorization request from a PEP doesn't conform to the query input expected by OPA.

You can also use this architecture with custom policy engines. However, any advantages gained from OPA must be replaced with logic provided by the custom policy engine.

A distributed PDP with PEPs on APIs provides an option to create a robust authorization system for APIs. It's simple to implement and provides an easy-to-use idempotent interface for making authorization decisions for APIs, microservices, Backend for Frontend (BFF) layers, or other application components. This approach also has the advantage of reducing the latency that you might encounter in the centralized PDP model.

### Distributed PDP as a library

You can also request authorization decisions from a PDP that is made available as a library or package for use within an application. OPA can be used as a Go third-party library. For other programming languages, adopting this model generally means that you must create a custom policy engine.
Implementation considerations

DevOps, monitoring, and logging

In this proposed authorization paradigm, policies are centralized in the authorization service. This centralization is deliberate because one of the goals of the design models discussed in this guide is to achieve policy decoupling, or the removal of authorization logic from other components in the application. The Open Policy Agent (OPA) provides a mechanism for updating policies when changes to authorization logic are necessary. This functionality is offered by a simple REST API that you can configure to pull new versions of policies (or bundles) from an established location or to push policies on demand. We recommend that you create a robust CI/CD pipeline to augment a control plane for versioning, verifying, and updating policies used by OPA to make authorization decisions. Additionally, OPA offers a basic discovery service where new agents can be configured dynamically and managed centrally by a control plane that distributes discovery bundles. This feature can reduce the administrative burden of managing a distributed policy decision point (PDP).

The control plane provides additional benefits for monitoring and auditing as well. You can monitor OPA status through a control plane. Perhaps more importantly, logs that contain OPA's authorization decisions can be exported to remote HTTP servers for log aggregation. These decision logs are invaluable for auditing purposes. If you are considering adopting an authorization model where access control decisions are decoupled from your application, make sure that your authorization service has effective monitoring, logging, and CI/CD management capabilities for onboarding new PDPs or updating policies.

Retrieving external data for a PDP

A PDP might require additional data to make an authorization decision beyond what is provided as input. For example, you might have user or tenant-specific attributes stored in a database that must be referenced by a PDP in order to make an authorization decision. For OPA, if all data required for an authorization decision can be provided as input or as part of a JSON Web Token (JWT) passed as a component of the query, no additional configuration is required. (It is relatively simple to pass JWTs and SaaS context data to OPA as part of query input.) OPA can accept arbitrary JSON input in what is called the overload input approach. If a PDP requires data beyond what can be included as input or a JWT token, OPA provides several options for retrieving this data. These include bundling, pushing data (replication), and dynamic data retrieval.

Bundling

The OPA bundling feature supports the following process for external data retrieval:

1. The policy enforcement point (PEP) requests an authorization decision.
2. OPA downloads new policy bundles, including external data.
3. The bundling service replicates data from data source(s).

When you use the bundling feature, OPA periodically downloads policy and data bundles from a centralized bundle service. (The implementation and setup of a bundle service isn't provided by OPA.) All policies and external data that are pulled from the bundle service are stored in memory. This option will not work if the external data size is too large to be stored in memory, or if the data changes too frequently.
For more information about the bundling feature, see the OPA documentation.

**Replication (pushing data)**

The OPA replication approach supports the following process for external data retrieval:

1. The policy enforcement point (PEP) requests an authorization decision.
2. The data replicator pushes data to OPA.
3. The data replicator replicates data from data source(s).

In this alternative to the bundling approach, data is pushed to, instead of being periodically pulled by, OPA. (The implementation and setup of a replicator isn't provided by OPA.) The push approach has the same data size limitations as the bundling approach, because OPA stores all the data in memory. The primary advantage of the push option is that you can update data in OPA with deltas instead of replacing all the external data each time. This makes the push option more appropriate for datasets that change frequently.

For more information about the replication option, see the OPA documentation.

**Dynamic data retrieval**

If the external data to be retrieved is too large to be cached in OPA's memory, the data can be dynamically pulled from an external source during the evaluation of an authorization decision. When you use this approach, data is always up-to-date. This approach has two drawbacks: network latency and accessibility. Currently, OPA can retrieve data at runtime only through an HTTP request. If the calls that go to an external data source cannot return data as an HTTP response, they require a custom API or some other mechanism to provide this data to OPA. Because OPA can retrieve data only through HTTP requests, and the speed of retrieving the data is pivotal, we recommend that you use an AWS service like Amazon ElastiCache or Amazon DynamoDB to hold external data when possible.

For more information about the pull approach, see the OPA documentation.

**Using an authorization service for implementation**

When you fetch external data using bundling, replication, or a dynamic pull approach, we recommend that the authorization service facilitate this interaction. This is because the authorization service can retrieve external data and transform it into JSON for OPA to make authorization decisions. The following diagram shows how an authorization service can function with these three external data retrieval approaches.
Retrieving external data for OPA flow – bundle or dynamic data retrieval at decision time:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OPA calls the authorization service local API endpoint. This endpoint is configured to serve as a bundle endpoint or the endpoint for dynamic data retrieval during authorization decisions.</td>
</tr>
<tr>
<td>2</td>
<td>The authorization service queries or calls the external data source to retrieve the data. (For a bundle endpoint, this data should also contain OPA policies and rules. Bundle updates replace everything—both data and policies—in the OPA cache.)</td>
</tr>
<tr>
<td>3</td>
<td>The authorization service performs any transformation necessary on the returned data to turn it into the expected JSON input.</td>
</tr>
<tr>
<td>4</td>
<td>The data is returned to OPA. It is cached in memory for bundle configuration and used immediately for dynamic authorization decisions.</td>
</tr>
</tbody>
</table>

Retrieving external data for OPA flow – replicator:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The replicator (part of the authorization service) calls the external data source and retrieves any data to be updated in OPA. This can include policies, rules, and external data. This call can be on a set cadence or happen in response to data updates in the external source.</td>
</tr>
<tr>
<td>2</td>
<td>The authorization service performs any transformation necessary on the returned data to turn it into the expected JSON input.</td>
</tr>
<tr>
<td>3</td>
<td>The authorization service calls OPA and caches the data in memory. The authorization service can selectively update data, policies, and rules.</td>
</tr>
</tbody>
</table>

Recommendations for tenant isolation and privacy of external data for RBAC

The previous section provided several approaches for importing external data into OPA to assist in making authorization decisions. In cases where it is possible, we recommend that you use the overload input approach for passing SaaS context data to OPA to make authorization decisions. However, in role-based access control (RBAC) or RBAC and attribute-based access control (ABAC) hybrid models, this data will often be insufficient, because roles and permissions will have to be referenced to make authorization decisions. To maintain tenant isolation and the privacy of role mapping, this data should not reside...
within OPA. RBAC data should reside in an external data source, such as a database. Furthermore, OPA should not be used to map predefined roles to specific permissions, because this makes it difficult for tenants to define their own roles and permissions. It also makes your authorization logic rigid and in need of constant update.

Some secure approaches for maintaining the privacy and tenant isolation of RBAC data is to use dynamic data retrieval or the replicator approach for getting external data for OPA. This is because the authorization service pictured in the previous diagram can be used to provide only tenant-specific or user-specific external data for making an authorization decision. For example, you can use a replicator to provide RBAC data or a permissions matrix to the OPA cache when a user logs in, and have the data be referenced based on a user provided in the input data. You can use a similar approach with dynamically pulled data to retrieve only the relevant data for making authorization decisions. Furthermore, in the dynamic data retrieval approach, this data does not have to be cached in OPA. The bundling approach isn't as effective as the dynamic retrieval approach at maintaining tenant isolation, because it updates everything in the OPA cache and can't process precise updates. The bundling model is still a good approach for updating OPA policies and non-RBAC data.
Best practices

This section lists some of the high-level takeaways from this guide. For detailed discussions on each point, follow the links to the corresponding sections.

Select an access control model that works for your application

This guide discusses several access control models (p. 3). Depending on your application and business requirements, you should select a model that works for you. Consider how you can use these models to fulfill your access control needs, and how your access control needs might evolve, requiring changes to your selected approach.

Implement a PDP

The policy decision point (PDP) (p. 5) can be characterized as a policy or rules engine. This component is responsible for applying policies or rules and returning a decision on whether a particular access is permitted. A PDP allows authorization logic in application code to be offloaded to a separate system. This can simplify application code. It also provides an easy-to-use idempotent interface for making authorization decisions for APIs, microservices, Backend for Frontend (BFF) layers, or any other application component. A PDP can be used to enforce tenancy requirements consistently across an application.

Implement PEPs for every API in your organization

The implementation of a policy enforcement point (PEP) (p. 18) requires determining where access control enforcement should occur in an application. As a first step, locate the points in your application where you can incorporate PEPs. Consider this principle when deciding where to add PEPs:

*If an application exposes an API, there should be authorization and access control on that API.*

Consider using OPA as a policy engine for your PDP

The Open Policy Agent (OPA) (p. 5) has advantages over custom policy engines. OPA and its policy evaluation with Rego provide a flexible, pre-built policy engine that supports writing policies in a high-level declarative language. This makes the level of effort required for implementing a policy engine significantly less than building your own solution. Furthermore, OPA is quickly becoming a well-supported authorization standard.
Implement a control plane for OPA for DevOps, monitoring, and logging

Because OPA doesn’t provide a means to update and track changes to authorization logic through source control, we recommend that you implement a control plane (p. 25) to perform these functions. This will allow for updates to be more easily distributed to OPA agents, particularly if OPA is operating in a distributed system, which will reduce the administrative burden of using OPA. Additionally, a control plane can be used to collect logs for aggregation and to monitor the status of OPA agents.

Determine whether external data is required for authorization decisions, and select a model to accommodate it

If it is possible for a PDP to make authorization decisions based solely on data that is contained in a JSON Web Token (JWT), it is usually not necessary to import external data to assist in making authorization decisions. If OPA is used as a PDP, it can also accept arbitrary JSON data as overload input that is passed as part of the request, even if this data isn’t included as part of a JWT. Using a JWT or overload input methods are generally far easier than maintaining external data in another source. If more complex external data is required to make authorization decisions, OPA offers several models for retrieving external data (p. 25).
FAQ

This section provides answers to commonly raised questions about implementing API access control and authorization in multi-tenant SaaS applications.

What is the difference between authorization and authentication?

Authentication is the process of verifying who a user is. Authorization grants permissions to users to access a specific resource.

Why do I need to consider authorization for my SaaS application?

SaaS applications have multiple tenants. A tenant can be a customer organization or any external entity that uses that SaaS application. Depending on how the application is designed, this means that tenants may be accessing shared APIs, databases, or other resources. It is important to maintain tenant isolation—that is, the separation of permissions and data by tenant—to prevent users from one tenant accessing another tenant’s private information. Authorization in SaaS applications is often designed to make sure that tenant isolation is maintained throughout an application and that tenants can access only their own resources.

Why do I need an access control model?

Access control models are used to create a consistent method of determining how to grant access to resources in an application. This can be done by assigning roles to users that are closely aligned with business logic, or it can be based on other contextual attributes such as the time of day or whether a user meets a predefined condition. Access control models form the basic logic your application uses when making authorization decisions to determine the user permissions.

Is API access control necessary for my application?

Yes. APIs should always verify that the caller has the appropriate access. Pervasive API access control also ensures that access is only granted based on tenants so that appropriate isolation can be maintained.

Why are policy engines or PDPs recommended for authorization?

A policy decision point (PDP) allows authorization logic in application code to be offloaded to a separate system. This can simplify application code. It also provides an easy-to-use idempotent interface for
making authorization decisions for APIs, microservices, Backend for Frontend (BFF) layers, or any other application component.

What is a PEP?

A policy enforcement point (PEP) is responsible for receiving authorization requests that are sent to the PDP for evaluation. A PEP can be anywhere in an application where data and resources must be protected, or where authorization logic is applied. PEPs are relatively simple compared to PDPs. A PEP is responsible only for requesting and evaluating an authorization decision and does not require any authorization logic to be incorporated into it.

Are there open-source alternatives to OPA?

There are a few open-source systems that are similar to the Open Policy Agent (OPA), such as the Common Expression Language (CEL). This guide focuses on OPA because it is widely adopted, documented, and adaptable to many different types of applications and authorization requirements.

Do I need to write an authorization service to use OPA, or can I interact with OPA directly?

You can interact with OPA directly. An authorization service in the context of this guidance refers to a service that translates authorization decision requests into OPA queries, and vice versa. If your application can query and accept OPA responses directly, there is no need to introduce this additional complexity.

How do I monitor my OPA agents for uptime and auditing purposes?

OPA does provide logging and basic uptime monitoring, though the default configuration will likely be insufficient for enterprise deployments. For more information, see the DevOps, monitoring, and logging (p. 25) section.

Which operating systems and AWS services can I use to run OPA?

You can run OPA on macOS, Windows, and Linux. OPA agents can be configured on Amazon Elastic Compute Cloud (Amazon EC2) agents as well as containerization services such as Amazon Elastic Container Service (Amazon ECS) and Amazon Elastic Kubernetes Service (Amazon EKS).

Can I run OPA on AWS Lambda?

You can run OPA on Lambda as a Go library. The AWS blog post Creating a custom Lambda authorizer using Open Policy Agent discusses how this can be done for an API Gateway Lambda authorizer.
How should I decide between a distributed PDP and centralized PDP approach?

This depends on your application. It will most likely be determined based on the latency difference between a distributed and centralized PDP model. We recommend that you build a proof of concept and test your application's performance to verify your solution.

Can I use OPA for use cases besides APIs?

Yes. The OPA documentation provides examples for Kubernetes, Envoy, Docker, Kafka, SSH and sudo, and Terraform. Additionally, OPA is capable of returning arbitrary JSON in response to queries by using Rego partial rules. Depending on the query, OPA can be used to answer many questions with JSON responses.
Next steps

The complexity of authorization and API access control for multi-tenant SaaS applications can be overcome by adopting a standardized, language-agnostic approach to making authorization decisions. These approaches incorporate policy decision points (PDPs) and policy enforcement points (PDPs) that enforce access in a flexible and pervasive manner. Multiple approaches to access control—such as role-based access control (RBAC), attribute-based access control (ABAC), or a combination of the two—can be incorporated into a cohesive access control strategy. Removing authorization logic from an application eliminates the overhead of including ad hoc solutions in application code to address access control. The implementation and best practices discussed in this guide are intended to inform and standardize an approach to the implementation of authorization and API access control in multi-tenant SaaS applications. You can use this guidance as the first step in gathering information and designing a robust access control and authorization system for your application.

Next steps:

- Review your authorization and tenant isolation needs, and select an access control model for your application.
- Implement the Open Policy Agent (OPA) and build a proof of concept for testing. (Alternatively, write your own custom policy engine.)
- Identify APIs and locations in your application where PEPs should be implemented.
Resources

References

- The OPA official documentation
- Why Enterprises Must Embrace The Most Recently Graduated CNCF Project – Open Policy Agent (Forbes article by Janakiram MSV, February 8, 2021)
- Creating a custom Lambda authorizer using Open Policy Agent (AWS blog post)
- Realize policy as code with AWS Cloud Development Kit through Open Policy Agent (AWS blog post)
- Cloud governance and compliance on AWS with policy as code (AWS blog post)
- Using Open Policy Agent on Amazon EKS (AWS blog post)
- Compliance as Code for Amazon ECS using Open Policy Agent, Amazon EventBridge, and AWS Lambda (AWS blog post)
- Policy-based countermeasures for Kubernetes – Part 1 (AWS blog post)

Tools

- The Rego Playground (for testing Rego in a browser)
- OPA GitHub repository

Partners

- Identity and Access Control Partners
- Application Security Partners
- Governance, Risk, and Compliance Partners
- Security Operations and Automation Partners
- Security Engineering Partners
Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>— (p. 36)</td>
<td>Initial publication</td>
<td>August 17, 2021</td>
</tr>
</tbody>
</table>
AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the Provide feedback link at the end of the glossary.

Modernization terms

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, Domain-Driven Design: Tackling Complexity in the Heart of Software (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

microservice

A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture

An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.
monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application’s features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.

polyglot persistence

Independently choosing a microservice’s data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements. For more information, see Enabling data persistence in microservices.

split-and-seed model

A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your organization’s capabilities and services, improves developer productivity, and supports rapid innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development. For more information, see the Two-pizza team section of the Introduction to DevOps on AWS whitepaper.