Multi-tenant SaaS authorization and API access control

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Multi-tenant SaaS authorization and API access control

Table of Contents

Multi-tenant SaaS authorization and API access control

Implementation options and best practices

Tabby Ward, Thomas Davis, Gideon Landeman, and Tomas Riha, Amazon Web Services (AWS)

January 2024 (document history)

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.

> **Note**
>
> Although this guide focuses primarily on OPA, AWS has a new service offering called Amazon Verified Permissions that shares much of the same functionality and advantages as OPA. For details, we recommend that you consult the Amazon Verified Permissions documentation. Many of the concepts, techniques, and best practices discussed in this guide are relevant to Amazon Verified Permissions.

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.

Factor	RBAC	ABAC	Hybrid
Flexibility	Medium	High	High
Simplicity	High	Low	Medium
Granularity	Low	High	Medium
Dynamic decisions and rules	No	Yes	Yes
Context-aware	No	Yes	Somewhat

Factor	RBAC	ABAC	Hybrid
Implementation effort	Low	High	Medium

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.

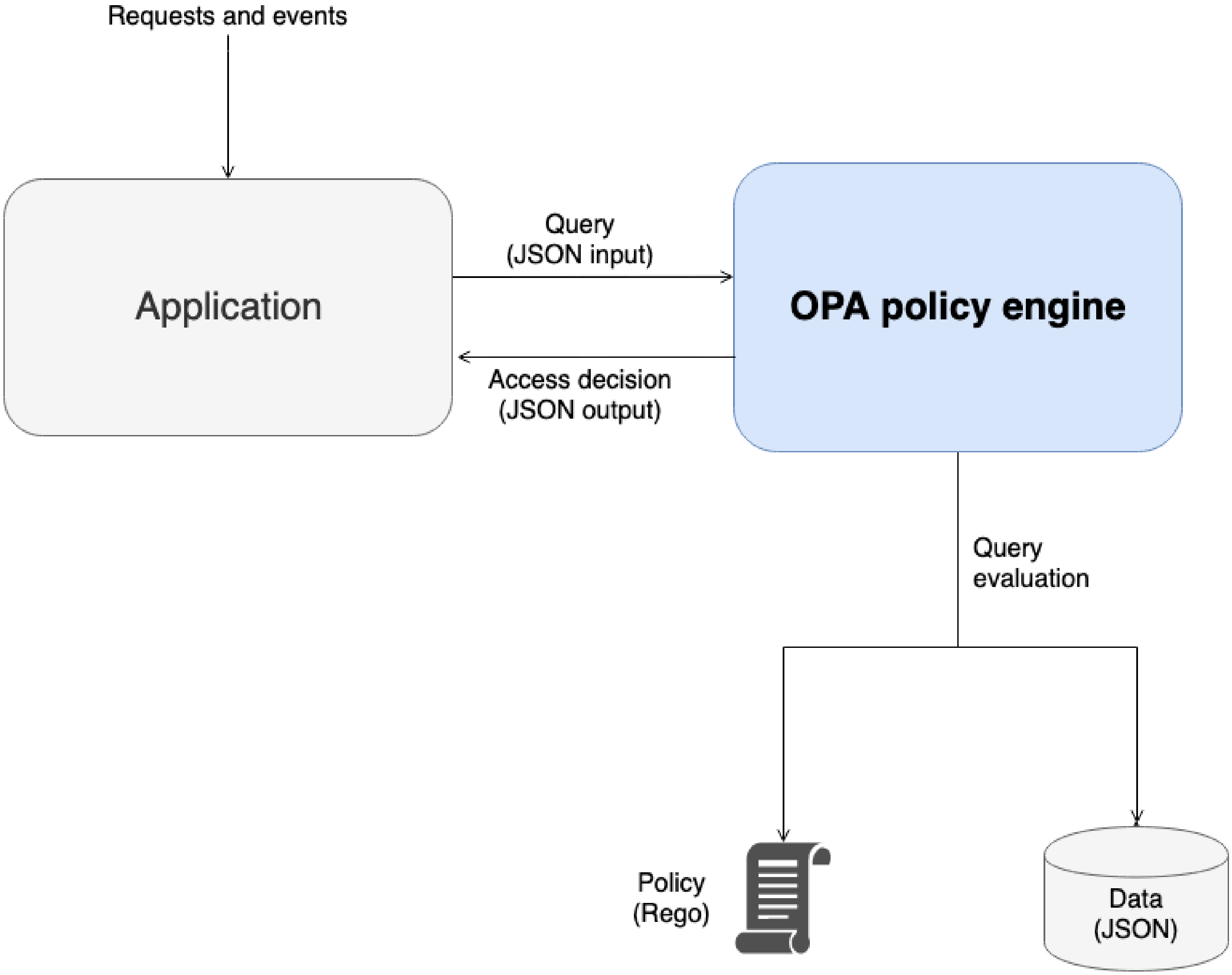
PDP implementation methods:

- Using OPA
- Using a custom policy engine

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

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?

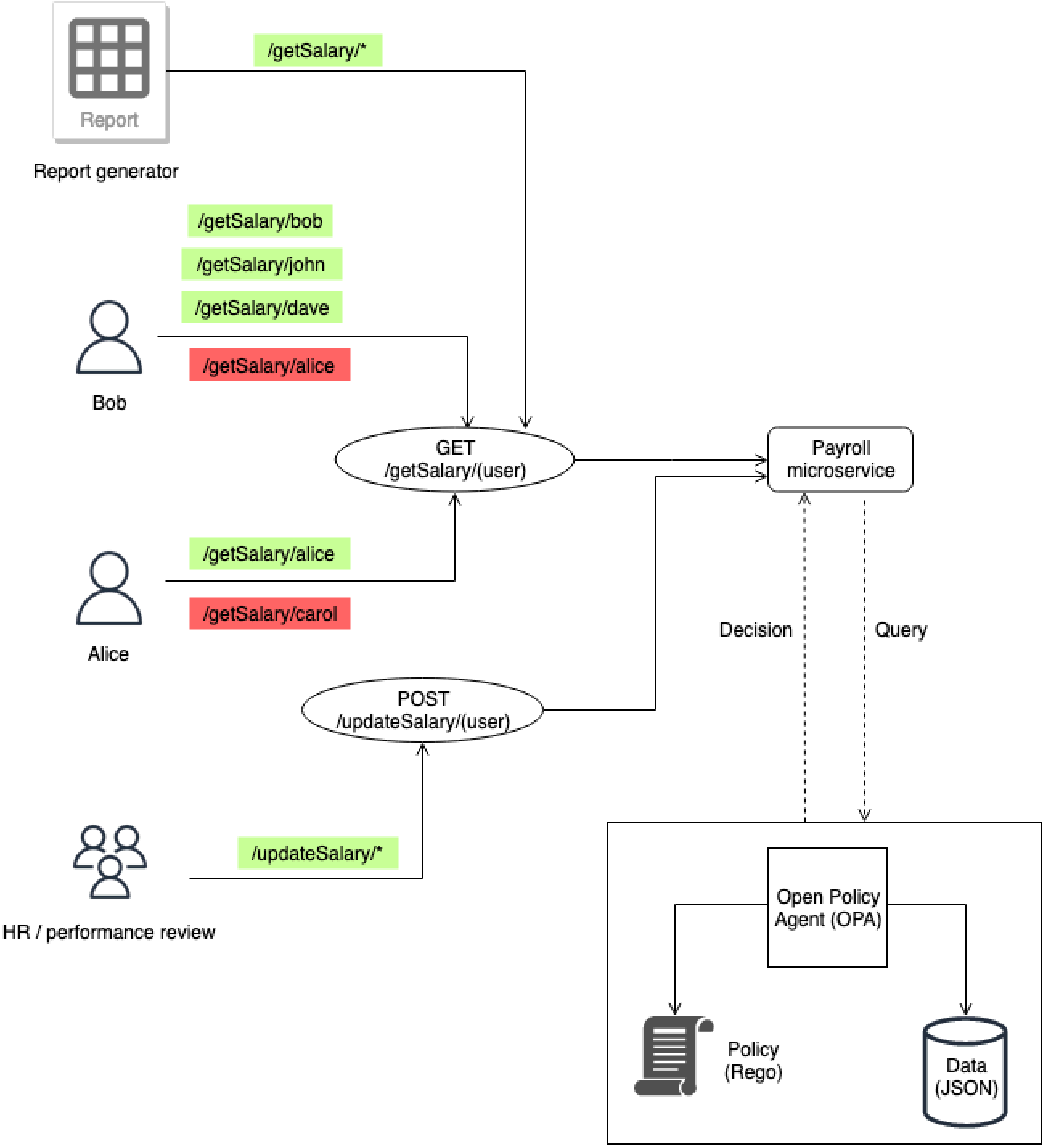
To answer these questions, Rego employs a basic philosophy about how these decisions can be made. The two key tenets when drafting policy in Rego are:

- 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:

```
{
    "user": "bob",
    "method": "GET",
    "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.

```
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:

```
{
    "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:

```
"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:

```
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:

```
{
    "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
```

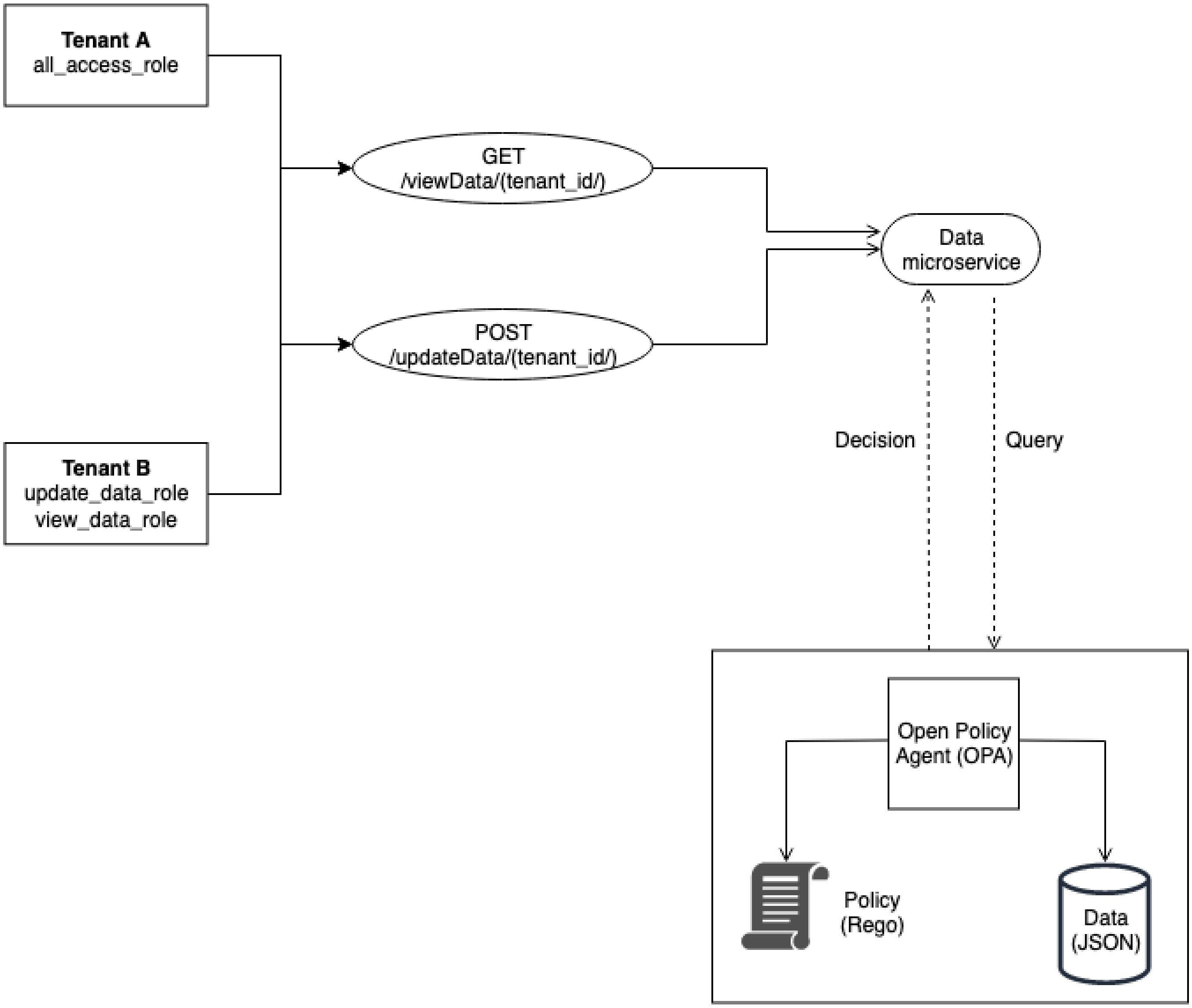
```
allow = true {
    input.method == "GET"
    input.path = ["getSalary", user]
    input.user == user
}

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:

```
{
    "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.

```
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")
}
```

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

```
{
    "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:

```
{
    "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:

```
{
    "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:

```
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`:

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

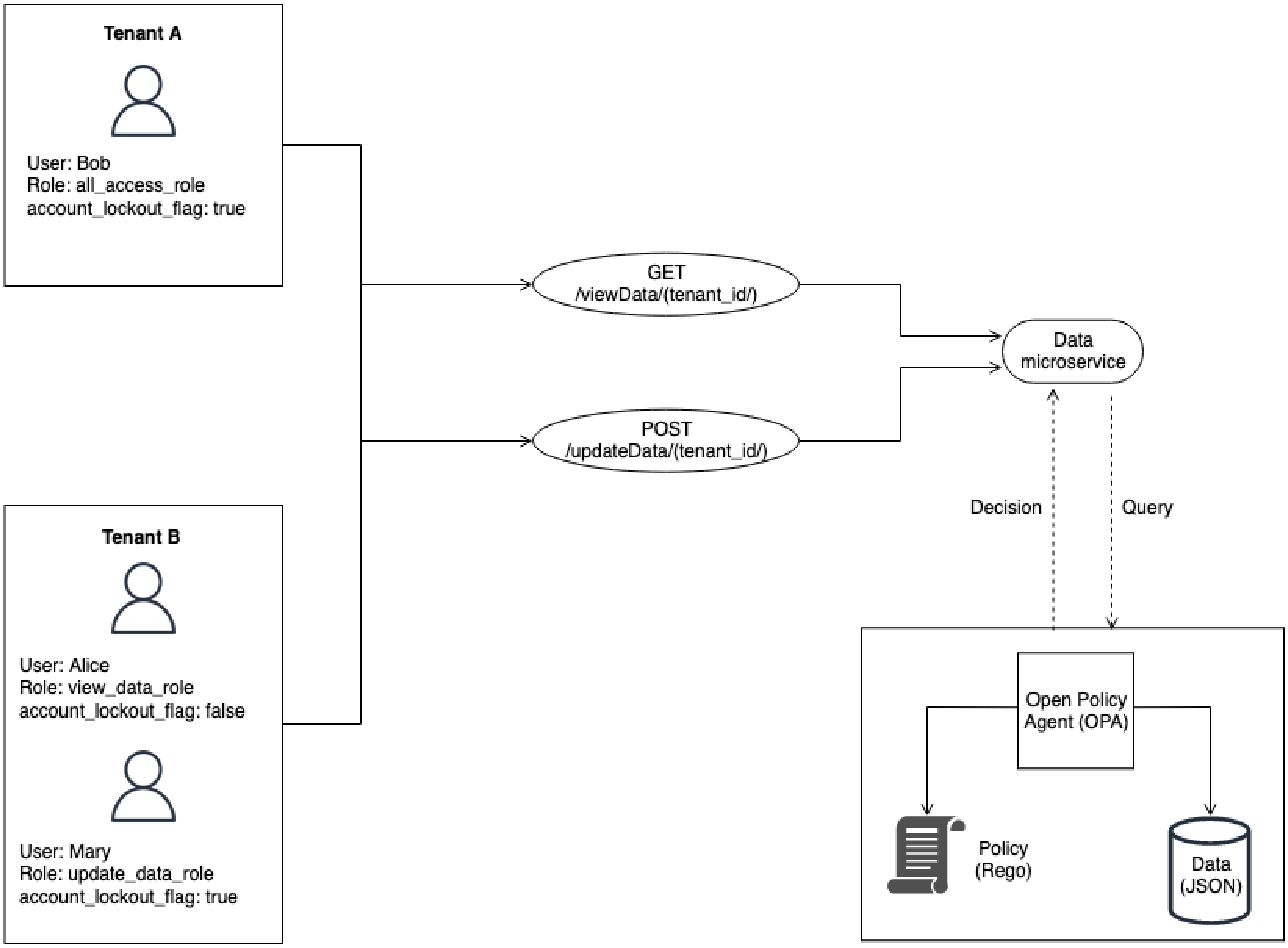
This query input, when evaluated with the `allowUpdateData` rule, returns the following authorization decision:

```
{
    "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:

```
{
    "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:

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

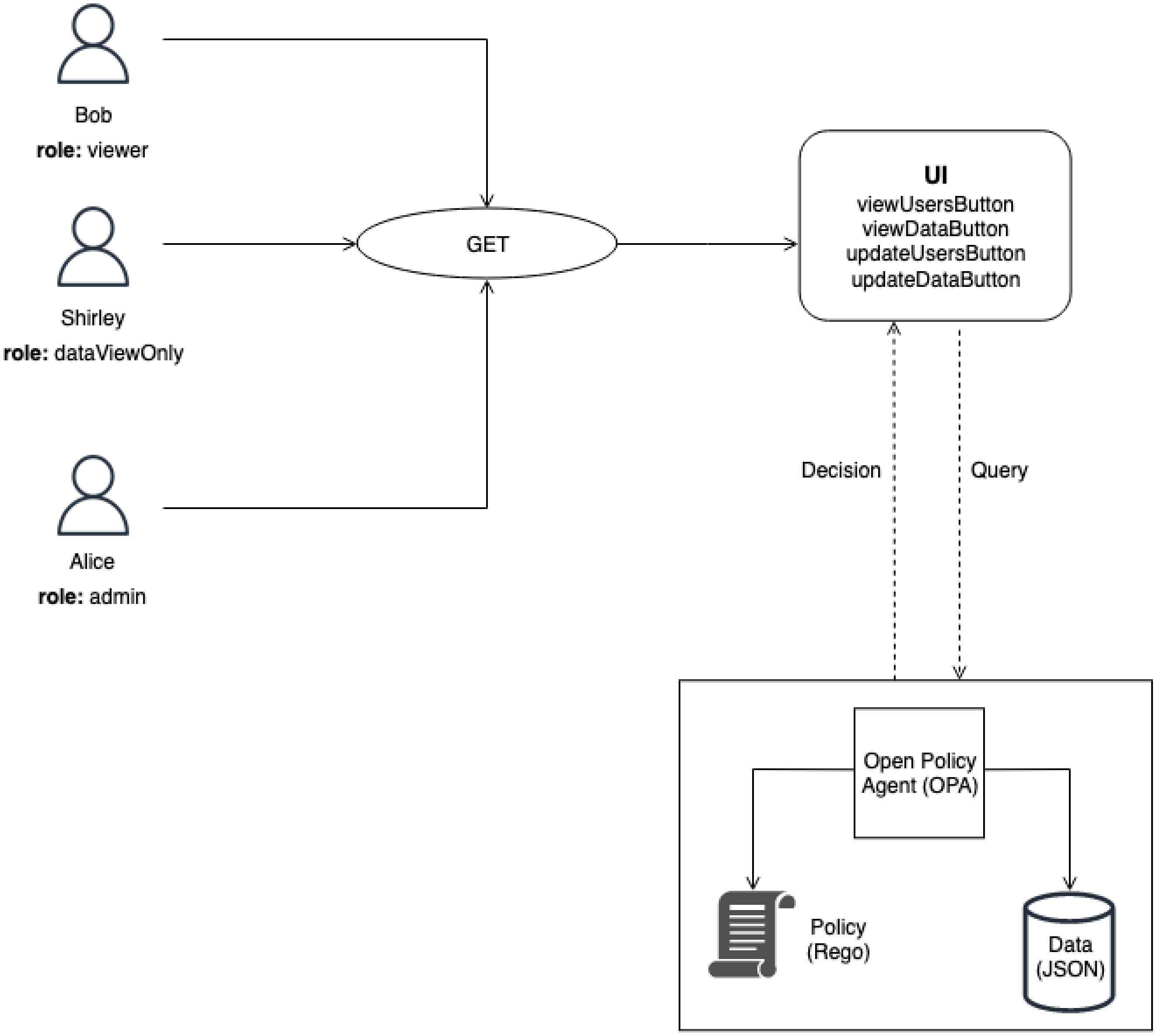
The rule that is queried for the access decision is similar to the previous examples, but includes an additional line to check for the `account_lockout_flag` attribute:

```
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:

```
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:

```
{
    "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:

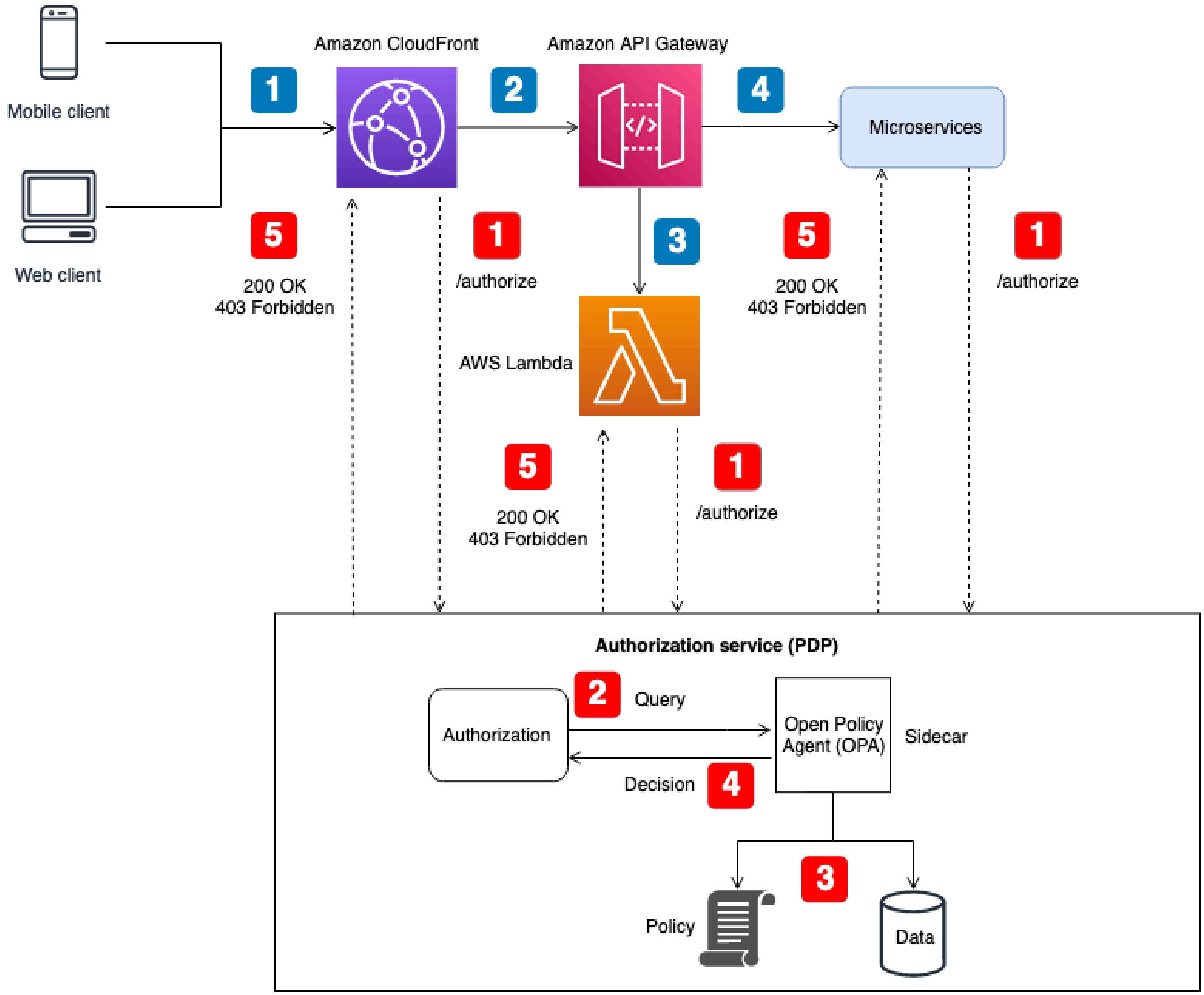
- Centralized PDP with PEPs on APIs
- Distributed PDP with PEPs on APIs
- Distributed PDP as a library

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.



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:

1	The PEP calls the authorization service and passes request data, including any JWT tokens.
2	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.
3	OPA evaluates the input based on the relevant policies specified by the query. Data is imported to make an authorization decision if necessary.
4	OPA returns a decision to the authorization service.
5	The authorization decision to 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 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.

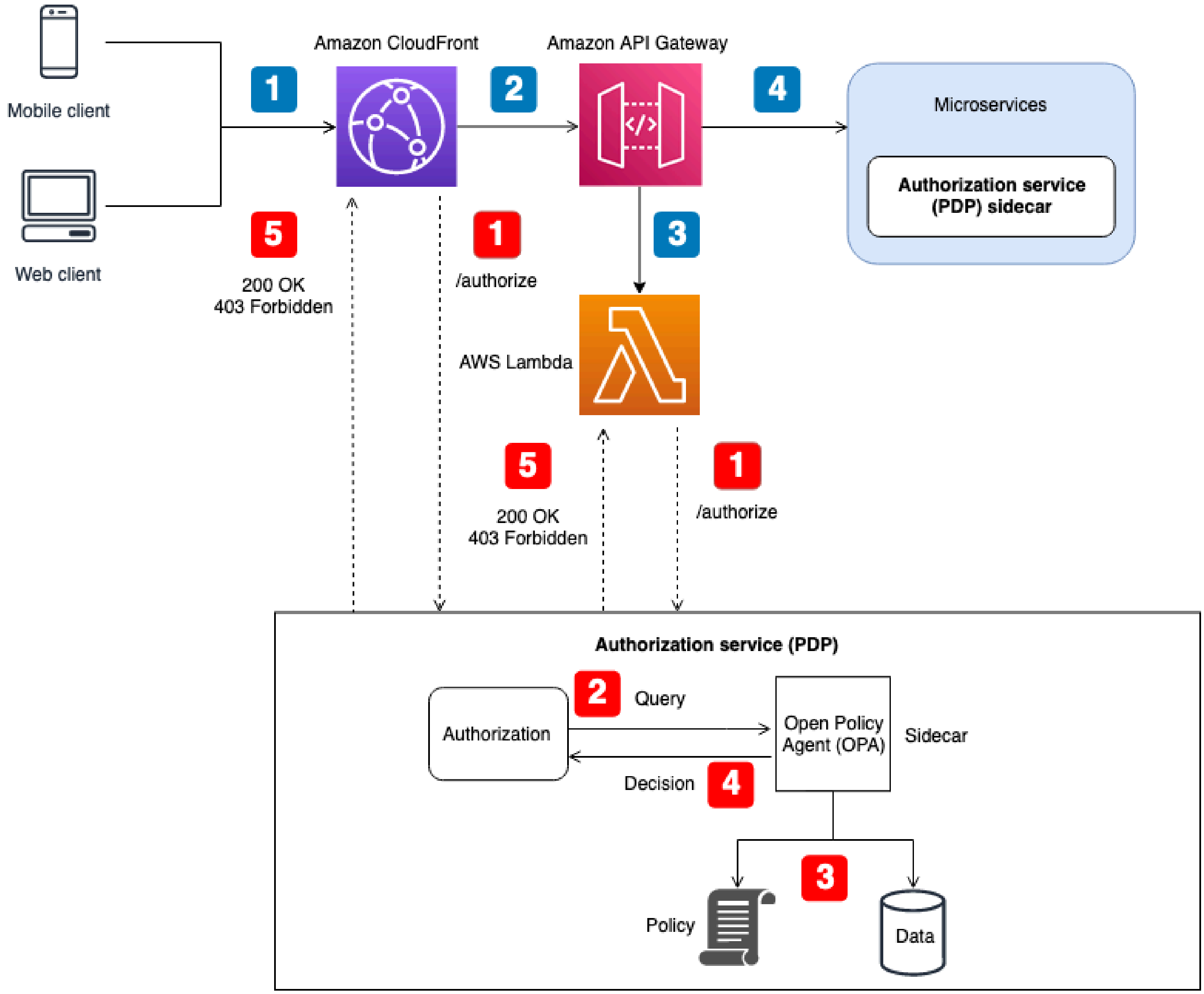
Distributed PDP with PEPs on APIs

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 a PDP.

In this approach, PDPs are implemented in multiple places in the application. For application components that have onboard compute capabilities that can run OPA and support a PDP, such as a containerized service with a sidecar or an Amazon Elastic Compute Cloud (Amazon EC2) instance, PDP decisions can be integrated directly into the application component without having to make a RESTful API call to a centralized PDP service. This has the benefit of reducing the latency that you might encounter in the centralized PDP model, because not every application component has to make additional API calls to obtain authorization decisions. However, a centralized PDP is still necessary in this model for application components that do not have onboard compute capabilities that enable direct integration of a PDP—such as the Amazon CloudFront or Amazon API Gateway services. The following diagram shows how this combination of a centralized PDP and a distributed PDP 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:

1	The PEP calls the authorization service and passes request data, including any JWT tokens.
2	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.
3	OPA evaluates the input based on the relevant policies specified by the query. Data is imported to make an authorization decision if necessary.
4	OPA returns a decision to the authorization service.
5	The authorization decision to 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 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

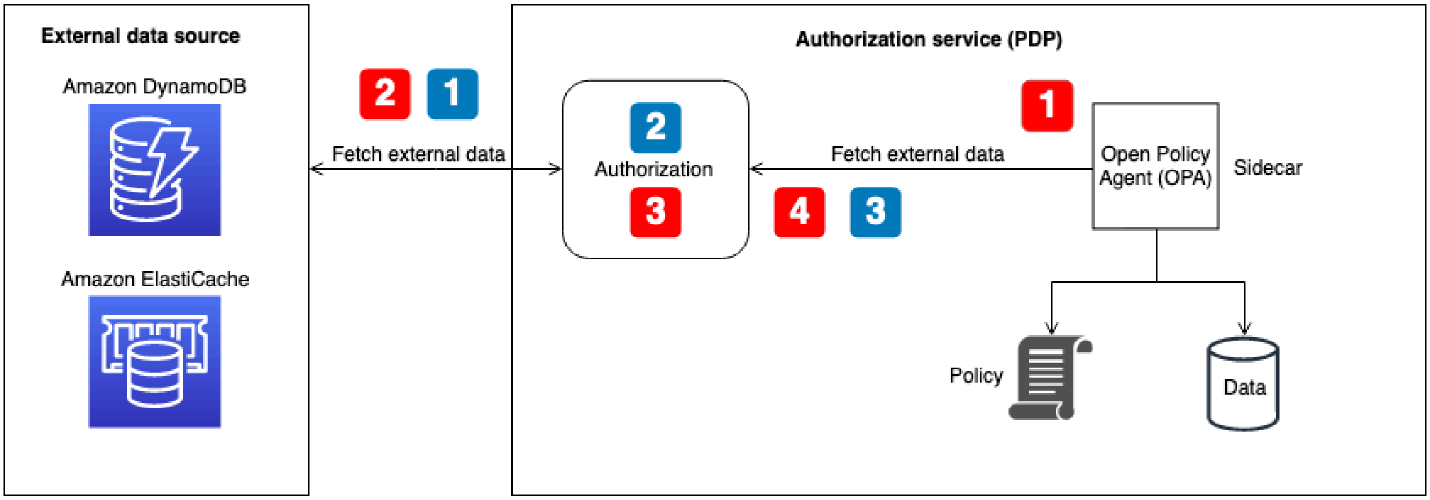
If the external data to be retrieved is too large to be cached in OPA's memory, the data can be dynamically pulled from an external source during the evaluation of an authorization decision.

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:

1	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.

2	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.)
3	The authorization service performs any transformation necessary on the returned data to turn it into the expected JSON input.
4	The data is returned to OPA. It is cached in memory for bundle configuration and used immediately for dynamic authorization decisions.

Retrieving external data for OPA flow – replicator:

1	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.
2	The authorization service performs any transformation necessary on the returned data to turn it into the expected JSON input.
3	The authorization service calls OPA and caches the data in memory. The authorization service can selectively update data, policies, and rules.

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. 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) 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) 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) 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 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.

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 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 (PEPs) 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 Management Partners
- Application Security Partners
- Cloud Governance Partners
- Security 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.

Change	Description	Date
Clarified information	Clarified the distributed PDP with PEPs on APIs design model.	January 10, 2024
Added information about new AWS service	Added information about Amazon Verified Permissions, which provides the same functionality and benefits as OPA.	May 22, 2023
—	Initial publication	August 17, 2021

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.

Numbers

7 Rs

Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
- Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. This migration scenario is specific to VMware Cloud on AWS, which supports virtual machine (VM) compatibility and workload portability between your on-premises environment and AWS. You can use the VMware Cloud Foundation technologies from your on-premises data centers when you migrate your infrastructure to VMware Cloud on AWS. Example: Relocate the hypervisor hosting your Oracle database to VMware Cloud on AWS.
- Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later

time, and legacy applications that you want to retain, because there's no business justification for migrating them.

- Retire – Decommission or remove applications that are no longer needed in your source environment.

A

ABAC

See attribute-based access control.

abstracted services

See managed services.

ACID

See atomicity, consistency, isolation, durability.

active-active migration

A database migration method in which the source and target databases are kept in sync (by using a bidirectional replication tool or dual write operations), and both databases handle transactions from connecting applications during migration. This method supports migration in small, controlled batches instead of requiring a one-time cutover. It's more flexible but requires more work than active-passive migration.

active-passive migration

A database migration method in which in which the source and target databases are kept in sync, but only the source database handles transactions from connecting applications while data is replicated to the target database. The target database doesn't accept any transactions during migration.

aggregate function

A SQL function that operates on a group of rows and calculates a single return value for the group. Examples of aggregate functions include `SUM` and `MAX`.

AI

See artificial intelligence.

AIOps

See artificial intelligence operations.

anonymization

The process of permanently deleting personal information in a dataset. Anonymization can help protect personal privacy. Anonymized data is no longer considered to be personal data.

anti-pattern

A frequently used solution for a recurring issue where the solution is counter-productive, ineffective, or less effective than an alternative.

application control

A security approach that allows the use of only approved applications in order to help protect a system from malware.

application portfolio

A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence (AI)

The field of computer science that is dedicated to using computing technologies to perform cognitive functions that are typically associated with humans, such as learning, solving problems, and recognizing patterns. For more information, see What is Artificial Intelligence?

artificial intelligence operations (AIOps)

The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

asymmetric encryption

An encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption. You can share the public key because it isn't used for decryption, but access to the private key should be highly restricted.

atomicity, consistency, isolation, durability (ACID)

A set of software properties that guarantee the data validity and operational reliability of a database, even in the case of errors, power failures, or other problems.

attribute-based access control (ABAC)

The practice of creating fine-grained permissions based on user attributes, such as department, job role, and team name. For more information, see ABAC for AWS in the AWS Identity and Access Management (IAM) documentation.

authoritative data source

A location where you store the primary version of data, which is considered to be the most reliable source of information. You can copy data from the authoritative data source to other locations for the purposes of processing or modifying the data, such as anonymizing, redacting, or pseudonymizing it.

Availability Zone

A distinct location within an AWS Region that is insulated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region.

AWS Cloud Adoption Framework (AWS CAF)

A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

AWS Workload Qualification Framework (AWS WQF)

A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

B

BCP

See business continuity planning.

behavior graph

A unified, interactive view of resource behavior and interactions over time. You can use a behavior graph with Amazon Detective to examine failed logon attempts, suspicious API calls, and similar actions. For more information, see Data in a behavior graph in the Detective documentation.

big-endian system

A system that stores the most significant byte first. See also endianness.

binary classification

A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?" or "Is this product a book or a car?"

bloom filter

A probabilistic, memory-efficient data structure that is used to test whether an element is a member of a set.

branch

A contained area of a code repository. The first branch created in a repository is the *main branch*. You can create a new branch from an existing branch, and you can then develop features or fix bugs in the new branch. A branch you create to build a feature is commonly referred to as a *feature branch*. When the feature is ready for release, you merge the feature branch back into the main branch. For more information, see About branches (GitHub documentation).

break-glass access

In exceptional circumstances and through an approved process, a quick means for a user to gain access to an AWS account that they don't typically have permissions to access. For more information, see the Implement break-glass procedures indicator in the AWS Well-Architected guidance.

brownfield strategy

The existing infrastructure in your environment. When adopting a brownfield strategy for a system architecture, you design the architecture around the constraints of the current systems and infrastructure. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

buffer cache

The memory area where the most frequently accessed data is stored.

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.

business continuity planning (BCP)

A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

C

CAF

See AWS Cloud Adoption Framework.

CCoE

See Cloud Center of Excellence.

CDC

See change data capture.

change data capture (CDC)

The process of tracking changes to a data source, such as a database table, and recording metadata about the change. You can use CDC for various purposes, such as auditing or replicating changes in a target system to maintain synchronization.

chaos engineering

Intentionally introducing failures or disruptive events to test a system's resilience. You can use AWS Fault Injection Service (AWS FIS) to perform experiments that stress your AWS workloads and evaluate their response.

CI/CD

See continuous integration and continuous delivery.

classification

A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

client-side encryption

Encryption of data locally, before the target AWS service receives it.

Cloud Center of Excellence (CCoE)

A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.

cloud computing

The cloud technology that is typically used for remote data storage and IoT device management. Cloud computing is commonly connected to edge computing technology.

cloud operating model

In an IT organization, the operating model that is used to build, mature, and optimize one or more cloud environments. For more information, see Building your Cloud Operating Model.

cloud stages of adoption

The four phases that organizations typically go through when they migrate to the AWS Cloud:

- Project – Running a few cloud-related projects for proof of concept and learning purposes
- Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)
- Migration – Migrating individual applications
- Re-invention – Optimizing products and services, and innovating in the cloud

These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First & the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.

CMDB

See configuration management database.

code repository

A location where source code and other assets, such as documentation, samples, and scripts, are stored and updated through version control processes. Common cloud repositories include GitHub or AWS CodeCommit. Each version of the code is called a *branch*. In a microservice structure, each repository is devoted to a single piece of functionality. A single CI/CD pipeline can use multiple repositories.

cold cache

A buffer cache that is empty, not well populated, or contains stale or irrelevant data. This affects performance because the database instance must read from the main memory or disk, which is slower than reading from the buffer cache.

cold data

Data that is rarely accessed and is typically historical. When querying this kind of data, slow queries are typically acceptable. Moving this data to lower-performing and less expensive storage tiers or classes can reduce costs.

computer vision

A field of AI used by machines to identify people, places, and things in images with accuracy at or above human levels. Often built with deep learning models, it automates extraction, analysis, classification, and understanding of useful information from a single image or a sequence of images.

configuration management database (CMDB)

A repository that stores and manages information about a database and its IT environment, including both hardware and software components and their configurations. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.

conformance pack

A collection of AWS Config rules and remediation actions that you can assemble to customize your compliance and security checks. You can deploy a conformance pack as a single entity in an AWS account and Region, or across an organization, by using a YAML template. For more information, see Conformance packs in the AWS Config documentation.

continuous integration and continuous delivery (CI/CD)

The process of automating the source, build, test, staging, and production stages of the software release process. CI/CD is commonly described as a pipeline. CI/CD can help you automate processes, improve productivity, improve code quality, and deliver faster. For more information, see Benefits of continuous delivery. CD can also stand for *continuous deployment*. For more information, see Continuous Delivery vs. Continuous Deployment.

D

data at rest

Data that is stationary in your network, such as data that is in storage.

data classification

A process for identifying and categorizing the data in your network based on its criticality and sensitivity. It is a critical component of any cybersecurity risk management strategy because it helps you determine the appropriate protection and retention controls for the data. Data classification is a component of the security pillar in the AWS Well-Architected Framework. For more information, see Data classification.

data drift

A meaningful variation between the production data and the data that was used to train an ML model, or a meaningful change in the input data over time. Data drift can reduce the overall quality, accuracy, and fairness in ML model predictions.

data in transit

Data that is actively moving through your network, such as between network resources.

data minimization

The principle of collecting and processing only the data that is strictly necessary. Practicing data minimization in the AWS Cloud can reduce privacy risks, costs, and your analytics carbon footprint.

data perimeter

A set of preventive guardrails in your AWS environment that help make sure that only trusted identities are accessing trusted resources from expected networks. For more information, see Building a data perimeter on AWS.

data preprocessing

To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

data provenance

The process of tracking the origin and history of data throughout its lifecycle, such as how the data was generated, transmitted, and stored.

data subject

An individual whose data is being collected and processed.

data warehouse

A data management system that supports business intelligence, such as analytics. Data warehouses commonly contain large amounts of historical data, and they are typically used for queries and analysis.

database definition language (DDL)

Statements or commands for creating or modifying the structure of tables and objects in a database.

database manipulation language (DML)

Statements or commands for modifying (inserting, updating, and deleting) information in a database.

DDL

See database definition language.

deep ensemble

To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning

An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

defense-in-depth

An information security approach in which a series of security mechanisms and controls are thoughtfully layered throughout a computer network to protect the confidentiality, integrity, and availability of the network and the data within. When you adopt this strategy on AWS, you add multiple controls at different layers of the AWS Organizations structure to help secure resources. For example, a defense-in-depth approach might combine multi-factor authentication, network segmentation, and encryption.

delegated administrator

In AWS Organizations, a compatible service can register an AWS member account to administer the organization's accounts and manage permissions for that service. This account is called the *delegated administrator* for that service. For more information and a list of compatible services, see Services that work with AWS Organizations in the AWS Organizations documentation.

deployment

The process of making an application, new features, or code fixes available in the target environment. Deployment involves implementing changes in a code base and then building and running that code base in the application's environments.

development environment

See environment.

detective control

A security control that is designed to detect, log, and alert after an event has occurred. These controls are a second line of defense, alerting you to security events that bypassed the preventative controls in place. For more information, see Detective controls in *Implementing security controls on AWS*.

development value stream mapping (DVSM)

A process used to identify and prioritize constraints that adversely affect speed and quality in a software development lifecycle. DVSM extends the value stream mapping process originally designed for lean manufacturing practices. It focuses on the steps and teams required to create and move value through the software development process.

digital twin

A virtual representation of a real-world system, such as a building, factory, industrial equipment, or production line. Digital twins support predictive maintenance, remote monitoring, and production optimization.

dimension table

In a star schema, a smaller table that contains data attributes about quantitative data in a fact table. Dimension table attributes are typically text fields or discrete numbers that behave like text. These attributes are commonly used for query constraining, filtering, and result set labeling.

disaster

An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location. These events can be natural disasters, technical failures, or the result of human actions, such as unintentional misconfiguration or a malware attack.

disaster recovery (DR)

The strategy and process you use to minimize downtime and data loss caused by a disaster. For more information, see Disaster Recovery of Workloads on AWS: Recovery in the Cloud in the AWS Well-Architected Framework.

DML

See database manipulation language.

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.

DR

See disaster recovery.

drift detection

Tracking deviations from a baselined configuration. For example, you can use AWS CloudFormation to detect drift in system resources, or you can use AWS Control Tower to detect changes in your landing zone that might affect compliance with governance requirements.

DVSM

See development value stream mapping.

E

EDA

See exploratory data analysis.

edge computing

The technology that increases the computing power for smart devices at the edges of an IoT network. When compared with cloud computing, edge computing can reduce communication latency and improve response time.

encryption

A computing process that transforms plaintext data, which is human-readable, into ciphertext.

encryption key

A cryptographic string of randomized bits that is generated by an encryption algorithm. Keys can vary in length, and each key is designed to be unpredictable and unique.

endianness

The order in which bytes are stored in computer memory. Big-endian systems store the most significant byte first. Little-endian systems store the least significant byte first.

endpoint

See service endpoint.

endpoint service

A service that you can host in a virtual private cloud (VPC) to share with other users. You can create an endpoint service with AWS PrivateLink and grant permissions to other AWS accounts or to AWS Identity and Access Management (IAM) principals. These accounts or principals can connect to your endpoint service privately by creating interface VPC endpoints. For more information, see Create an endpoint service in the Amazon Virtual Private Cloud (Amazon VPC) documentation.

envelope encryption

The process of encrypting an encryption key with another encryption key. For more information, see Envelope encryption in the AWS Key Management Service (AWS KMS) documentation.

environment

An instance of a running application. The following are common types of environments in cloud computing:

- development environment – An instance of a running application that is available only to the core team responsible for maintaining the application. Development environments are used to test changes before promoting them to upper environments. This type of environment is sometimes referred to as a *test environment*.
- lower environments – All development environments for an application, such as those used for initial builds and tests.
- production environment – An instance of a running application that end users can access. In a CI/CD pipeline, the production environment is the last deployment environment.
- upper environments – All environments that can be accessed by users other than the core development team. This can include a production environment, preproduction environments, and environments for user acceptance testing.

epic

In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the program implementation guide.

exploratory data analysis (EDA)

The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

F

fact table

The central table in a star schema. It stores quantitative data about business operations. Typically, a fact table contains two types of columns: those that contain measures and those that contain a foreign key to a dimension table.

fail fast

A philosophy that uses frequent and incremental testing to reduce the development lifecycle. It is a critical part of an agile approach.

fault isolation boundary

In the AWS Cloud, a boundary such as an Availability Zone, AWS Region, control plane, or data plane that limits the effect of a failure and helps improve the resilience of workloads. For more information, see AWS Fault Isolation Boundaries.

feature branch

See branch.

features

The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance

How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see Machine learning model interpretability with :AWS.

feature transformation

To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the "2021-05-27 00:15:37" date into "2021", "May", "Thu", and "15", you can help the learning algorithm learn nuanced patterns associated with different data components.

FGAC

See fine-grained access control.

fine-grained access control (FGAC)

The use of multiple conditions to allow or deny an access request.

flash-cut migration

A database migration method that uses continuous data replication through change data capture to migrate data in the shortest time possible, instead of using a phased approach. The objective is to keep downtime to a minimum.

G

geo blocking

See geographic restrictions.

geographic restrictions (geo blocking)

In Amazon CloudFront, an option to prevent users in specific countries from accessing content distributions. You can use an allow list or block list to specify approved and banned countries. For more information, see Restricting the geographic distribution of your content in the CloudFront documentation.

Gitflow workflow

An approach in which lower and upper environments use different branches in a source code repository. The Gitflow workflow is considered legacy, and the trunk-based workflow is the modern, preferred approach.

greenfield strategy

The absence of existing infrastructure in a new environment. When adopting a greenfield strategy for a system architecture, you can select all new technologies without the restriction of compatibility with existing infrastructure, also known as brownfield. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

guardrail

A high-level rule that helps govern resources, policies, and compliance across organizational units (OUs). *Preventive guardrails* enforce policies to ensure alignment to compliance standards. They are implemented by using service control policies and IAM permissions boundaries. *Detective guardrails* detect policy violations and compliance issues, and generate alerts for remediation. They are implemented by using AWS Config, AWS Security Hub, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector, and custom AWS Lambda checks.

H

HA

See high availability.

heterogeneous database migration

Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides AWS SCT that helps with schema conversions.

high availability (HA)

The ability of a workload to operate continuously, without intervention, in the event of challenges or disasters. HA systems are designed to automatically fail over, consistently deliver high-quality performance, and handle different loads and failures with minimal performance impact.

historian modernization

An approach used to modernize and upgrade operational technology (OT) systems to better serve the needs of the manufacturing industry. A *historian* is a type of database that is used to collect and store data from various sources in a factory.

homogeneous database migration

Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.

hot data

Data that is frequently accessed, such as real-time data or recent translational data. This data typically requires a high-performance storage tier or class to provide fast query responses.

hotfix

An urgent fix for a critical issue in a production environment. Due to its urgency, a hotfix is usually made outside of the typical DevOps release workflow.

hypercare period

Immediately following cutover, the period of time when a migration team manages and monitors the migrated applications in the cloud in order to address any issues. Typically, this period is 1–4 days in length. At the end of the hypercare period, the migration team typically transfers responsibility for the applications to the cloud operations team.

I

IaC

See infrastructure as code.

identity-based policy

A policy attached to one or more IAM principals that defines their permissions within the AWS Cloud environment.

idle application

An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.

IIoT

See industrial Internet of Things.

immutable infrastructure

A model that deploys new infrastructure for production workloads instead of updating, patching, or modifying the existing infrastructure. Immutable infrastructures are inherently more consistent, reliable, and predictable than mutable infrastructure. For more information, see the Deploy using immutable infrastructure best practice in the AWS Well-Architected Framework.

inbound (ingress) VPC

In an AWS multi-account architecture, a VPC that accepts, inspects, and routes network connections from outside an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

incremental migration

A cutover strategy in which you migrate your application in small parts instead of performing a single, full cutover. For example, you might move only a few microservices or users to the new system initially. After you verify that everything is working properly, you can incrementally move additional microservices or users until you can decommission your legacy system. This strategy reduces the risks associated with large migrations.

infrastructure

All of the resources and assets contained within an application's environment.

infrastructure as code (IaC)

The process of provisioning and managing an application's infrastructure through a set of configuration files. IaC is designed to help you centralize infrastructure management, standardize resources, and scale quickly so that new environments are repeatable, reliable, and consistent.

industrial Internet of Things (IIoT)

The use of internet-connected sensors and devices in the industrial sectors, such as manufacturing, energy, automotive, healthcare, life sciences, and agriculture. For more information, see Building an industrial Internet of Things (IIoT) digital transformation strategy.

inspection VPC

In an AWS multi-account architecture, a centralized VPC that manages inspections of network traffic between VPCs (in the same or different AWS Regions), the internet, and on-premises networks. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

Internet of Things (IoT)

The network of connected physical objects with embedded sensors or processors that communicate with other devices and systems through the internet or over a local communication network. For more information, see What is IoT?

interpretability

A characteristic of a machine learning model that describes the degree to which a human can understand how the model's predictions depend on its inputs. For more information, see Machine learning model interpretability with AWS.

IoT

See Internet of Things.

IT information library (ITIL)

A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.

IT service management (ITSM)

Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the operations integration guide.

ITIL

See IT information library.

ITSM

See IT service management.

L

label-based access control (LBAC)

An implementation of mandatory access control (MAC) where the users and the data itself are each explicitly assigned a security label value. The intersection between the user security label and data security label determines which rows and columns can be seen by the user.

landing zone

A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

large migration

A migration of 300 or more servers.

LBAC

See label-based access control.

least privilege

The security best practice of granting the minimum permissions required to perform a task. For more information, see Apply least-privilege permissions in the IAM documentation.

lift and shift

See 7 Rs.

little-endian system

A system that stores the least significant byte first. See also endianness.

lower environments

See environment.

M

machine learning (ML)

A type of artificial intelligence that uses algorithms and techniques for pattern recognition and learning. ML analyzes and learns from recorded data, such as Internet of Things (IoT) data, to generate a statistical model based on patterns. For more information, see Machine Learning.

main branch

See branch.

managed services

AWS services for which AWS operates the infrastructure layer, the operating system, and platforms, and you access the endpoints to store and retrieve data. Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB are examples of managed services. These are also known as *abstracted services*.

MAP

See Migration Acceleration Program.

mechanism

A complete process in which you create a tool, drive adoption of the tool, and then inspect the results in order to make adjustments. A mechanism is a cycle that reinforces and improves itself as it operates. For more information, see Building mechanisms in the AWS Well-Architected Framework.

member account

All AWS accounts other than the management account that are part of an organization in AWS Organizations. An account can be a member of only one organization at a time.

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.

Migration Acceleration Program (MAP)

An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.

migration at scale

The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a *migration factory* of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.

migration factory

Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners, migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the Cloud Migration Factory guide in this content set.

migration metadata

The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.

migration pattern

A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA)

The process of gaining insights about an organization’s cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

migration strategy

The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.

ML

See machine learning.

MPA

See Migration Portfolio Assessment.

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.

multiclass classification

A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

mutable infrastructure

A model that updates and modifies the existing infrastructure for production workloads. For improved consistency, reliability, and predictability, the AWS Well-Architected Framework recommends the use of immutable infrastructure as a best practice.

O

OAC

See origin access control.

OAI

See origin access identity.

OCM

See organizational change management.

offline migration

A migration method in which the source workload is taken down during the migration process. This method involves extended downtime and is typically used for small, non-critical workloads.

OI

See operations integration.

OLA

See operational-level agreement.

online migration

A migration method in which the source workload is copied to the target system without being taken offline. Applications that are connected to the workload can continue to function during the migration. This method involves zero to minimal downtime and is typically used for critical production workloads.

operational-level agreement (OLA)

An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

operational readiness review (ORR)

A checklist of questions and associated best practices that help you understand, evaluate, prevent, or reduce the scope of incidents and possible failures. For more information, see Operational Readiness Reviews (ORR) in the AWS Well-Architected Framework.

operations integration (OI)

The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

organization trail

A trail that's created by AWS CloudTrail that logs all events for all AWS accounts in an organization in AWS Organizations. This trail is created in each AWS account that's part of the organization and tracks the activity in each account. For more information, see Creating a trail for an organization in the CloudTrail documentation.

organizational change management (OCM)

A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called *people acceleration*, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

origin access control (OAC)

In CloudFront, an enhanced option for restricting access to secure your Amazon Simple Storage Service (Amazon S3) content. OAC supports all S3 buckets in all AWS Regions, server-side encryption with AWS KMS (SSE-KMS), and dynamic `PUT` and `DELETE` requests to the S3 bucket.

origin access identity (OAI)

In CloudFront, an option for restricting access to secure your Amazon S3 content. When you use OAI, CloudFront creates a principal that Amazon S3 can authenticate with. Authenticated principals can access content in an S3 bucket only through a specific CloudFront distribution. See also OAC, which provides more granular and enhanced access control.

ORR

See operational readiness review.

outbound (egress) VPC

In an AWS multi-account architecture, a VPC that handles network connections that are initiated from within an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

P

permissions boundary

An IAM management policy that is attached to IAM principals to set the maximum permissions that the user or role can have. For more information, see Permissions boundaries in the IAM documentation.

personally identifiable information (PII)

Information that, when viewed directly or paired with other related data, can be used to reasonably infer the identity of an individual. Examples of PII include names, addresses, and contact information.

PII

See personally identifiable information.

playbook

A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

policy

An object that can define permissions (see identity-based policy), specify access conditions (see resource-based policy), or define the maximum permissions for all accounts in an organization in AWS Organizations (see service control policy).

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.

portfolio assessment

A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see Evaluating migration readiness.

predicate

A query condition that returns `true` or `false`, commonly located in a `WHERE` clause.

predicate pushdown

A database query optimization technique that filters the data in the query before transfer. This reduces the amount of data that must be retrieved and processed from the relational database, and it improves query performance.

preventative control

A security control that is designed to prevent an event from occurring. These controls are a first line of defense to help prevent unauthorized access or unwanted changes to your network. For more information, see Preventative controls in *Implementing security controls on AWS*.

principal

An entity in AWS that can perform actions and access resources. This entity is typically a root user for an AWS account, an IAM role, or a user. For more information, see *Principal* in Roles terms and concepts in the IAM documentation.

Privacy by Design

An approach in system engineering that takes privacy into account throughout the whole engineering process.

private hosted zones

A container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs. For more information, see Working with private hosted zones in the Route 53 documentation.

proactive control

A security control designed to prevent the deployment of noncompliant resources. These controls scan resources before they are provisioned. If the resource is not compliant with the control, then it isn't provisioned. For more information, see the Controls reference guide in the AWS Control Tower documentation and see Proactive controls in *Implementing security controls on AWS*.

production environment

See environment.

pseudonymization

The process of replacing personal identifiers in a dataset with placeholder values. Pseudonymization can help protect personal privacy. Pseudonymized data is still considered to be personal data.

Q

query plan

A series of steps, like instructions, that are used to access the data in a SQL relational database system.

query plan regression

When a database service optimizer chooses a less optimal plan than it did before a given change to the database environment. This can be caused by changes to statistics, constraints, environment settings, query parameter bindings, and updates to the database engine.

R

RACI matrix

See responsible, accountable, consulted, informed (RACI).

ransomware

A malicious software that is designed to block access to a computer system or data until a payment is made.

RASCI matrix

See responsible, accountable, consulted, informed (RACI).

RCAC

See row and column access control.

read replica

A copy of a database that's used for read-only purposes. You can route queries to the read replica to reduce the load on your primary database.

re-architect

See 7 Rs.

recovery point objective (RPO)

The maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

recovery time objective (RTO)

The maximum acceptable delay between the interruption of service and restoration of service.

refactor

See 7 Rs.

Region

A collection of AWS resources in a geographic area. Each AWS Region is isolated and independent of the others to provide fault tolerance, stability, and resilience. For more information, see Managing AWS Regions in *AWS General Reference*.

regression

An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

rehost

See 7 Rs.

release

In a deployment process, the act of promoting changes to a production environment.

relocate

See 7 Rs.

replatform

See 7 Rs.

repurchase

See 7 Rs.

resource-based policy

A policy attached to a resource, such as an Amazon S3 bucket, an endpoint, or an encryption key. This type of policy specifies which principals are allowed access, supported actions, and any other conditions that must be met.

responsible, accountable, consulted, informed (RACI) matrix

A matrix that defines the roles and responsibilities for all parties involved in migration activities and cloud operations. The matrix name is derived from the responsibility types defined in the matrix: responsible (R), accountable (A), consulted (C), and informed (I). The support (S) type is optional. If you include support, the matrix is called a *RASCI matrix*, and if you exclude it, it's called a *RACI matrix*.

responsive control

A security control that is designed to drive remediation of adverse events or deviations from your security baseline. For more information, see Responsive controls in *Implementing security controls on AWS*.

retain

See 7 Rs.

retire

See 7 Rs.

rotation

The process of periodically updating a secret to make it more difficult for an attacker to access the credentials.

row and column access control (RCAC)

The use of basic, flexible SQL expressions that have defined access rules. RCAC consists of row permissions and column masks.

RPO

See recovery point objective.

RTO

See recovery time objective.

runbook

A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

S

SAML 2.0

An open standard that many identity providers (IdPs) use. This feature enables federated single sign-on (SSO), so users can log into the AWS Management Console or call the AWS API operations without you having to create user in IAM for everyone in your organization. For more information about SAML 2.0-based federation, see About SAML 2.0-based federation in the IAM documentation.

SCP

See service control policy.

secret

In AWS Secrets Manager, confidential or restricted information, such as a password or user credentials, that you store in encrypted form. It consists of the secret value and its metadata. The secret value can be binary, a single string, or multiple strings. For more information, see Secret in the Secrets Manager documentation.

security control

A technical or administrative guardrail that prevents, detects, or reduces the ability of a threat actor to exploit a security vulnerability. There are four primary types of security controls: preventative, detective, responsive, and proactive.

security hardening

The process of reducing the attack surface to make it more resistant to attacks. This can include actions such as removing resources that are no longer needed, implementing the security best practice of granting least privilege, or deactivating unnecessary features in configuration files.

security information and event management (SIEM) system

Tools and services that combine security information management (SIM) and security event management (SEM) systems. A SIEM system collects, monitors, and analyzes data from servers, networks, devices, and other sources to detect threats and security breaches, and to generate alerts.

security response automation

A predefined and programmed action that is designed to automatically respond to or remediate a security event. These automations serve as detective or responsive security controls that help you implement AWS security best practices. Examples of automated response actions include modifying a VPC security group, patching an Amazon EC2 instance, or rotating credentials.

server-side encryption

Encryption of data at its destination, by the AWS service that receives it.

service control policy (SCP)

A policy that provides centralized control over permissions for all accounts in an organization in AWS Organizations. SCPs define guardrails or set limits on actions that an administrator can delegate to users or roles. You can use SCPs as allow lists or deny lists, to specify which services or actions are permitted or prohibited. For more information, see Service control policies in the AWS Organizations documentation.

service endpoint

The URL of the entry point for an AWS service. You can use the endpoint to connect programmatically to the target service. For more information, see AWS service endpoints in *AWS General Reference*.

service-level agreement (SLA)

An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

service-level indicator (SLI)

A measurement of a performance aspect of a service, such as its error rate, availability, or throughput.

service-level objective (SLO)

A target metric that represents the health of a service, as measured by a service-level indicator.

shared responsibility model

A model describing the responsibility you share with AWS for cloud security and compliance. AWS is responsible for security *of* the cloud, whereas you are responsible for security *in* the cloud. For more information, see Shared responsibility model.

SIEM

See security information and event management system.

single point of failure (SPOF)

A failure in a single, critical component of an application that can disrupt the system.

SLA

See service-level agreement.

SLI

See service-level indicator.

SLO

See service-level objective.

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.

SPOF

See single point of failure.

star schema

A database organizational structure that uses one large fact table to store transactional or measured data and uses one or more smaller dimensional tables to store data attributes. This structure is designed for use in a data warehouse or for business intelligence purposes.

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.

subnet

A range of IP addresses in your VPC. A subnet must reside in a single Availability Zone.

symmetric encryption

An encryption algorithm that uses the same key to encrypt and decrypt the data.

synthetic testing

Testing a system in a way that simulates user interactions to detect potential issues or to monitor performance. You can use Amazon CloudWatch Synthetics to create these tests.

T

tags

Key-value pairs that act as metadata for organizing your AWS resources. Tags can help you manage, identify, organize, search for, and filter resources. For more information, see Tagging your AWS resources.

target variable

The value that you are trying to predict in supervised ML. This is also referred to as an *outcome variable*. For example, in a manufacturing setting the target variable could be a product defect.

task list

A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

test environment

See environment.

training

To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

transit gateway

A network transit hub that you can use to interconnect your VPCs and on-premises networks. For more information, see What is a transit gateway in the AWS Transit Gateway documentation.

trunk-based workflow

An approach in which developers build and test features locally in a feature branch and then merge those changes into the main branch. The main branch is then built to the development, preproduction, and production environments, sequentially.

trusted access

Granting permissions to a service that you specify to perform tasks in your organization in AWS Organizations and in its accounts on your behalf. The trusted service creates a service-linked role in each account, when that role is needed, to perform management tasks for you. For more information, see Using AWS Organizations with other AWS services in the AWS Organizations documentation.

tuning

To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

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.

U

uncertainty

A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: *Epistemic uncertainty* is caused by limited, incomplete data, whereas *aleatoric uncertainty* is caused by the noise and randomness inherent in the data. For more information, see the Quantifying uncertainty in deep learning systems guide.

undifferentiated tasks

Also known as *heavy lifting*, work that is necessary to create and operate an application but that doesn't provide direct value to the end user or provide competitive advantage. Examples of undifferentiated tasks include procurement, maintenance, and capacity planning.

upper environments

See environment.

V

vacuuming

A database maintenance operation that involves cleaning up after incremental updates to reclaim storage and improve performance.

version control

Processes and tools that track changes, such as changes to source code in a repository.

VPC peering

A connection between two VPCs that allows you to route traffic by using private IP addresses. For more information, see What is VPC peering in the Amazon VPC documentation.

vulnerability

A software or hardware flaw that compromises the security of the system.

W

warm cache

A buffer cache that contains current, relevant data that is frequently accessed. The database instance can read from the buffer cache, which is faster than reading from the main memory or disk.

warm data

Data that is infrequently accessed. When querying this kind of data, moderately slow queries are typically acceptable.

window function

A SQL function that performs a calculation on a group of rows that relate in some way to the current record. Window functions are useful for processing tasks, such as calculating a moving average or accessing the value of rows based on the relative position of the current row.

workload

A collection of resources and code that delivers business value, such as a customer-facing application or backend process.

workstream

Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

WORM

See write once, read many.

WQF

See AWS Workload Qualification Framework.

write once, read many (WORM)

A storage model that writes data a single time and prevents the data from being deleted or modified. Authorized users can read the data as many times as needed, but they cannot change it. This data storage infrastructure is considered immutable.

Z

zero-day exploit

An attack, typically malware, that takes advantage of a zero-day vulnerability.

zero-day vulnerability

An unmitigated flaw or vulnerability in a production system. Threat actors can use this type of vulnerability to attack the system. Developers frequently become aware of the vulnerability as a result of the attack.

zombie application

An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.