aws

多租户 SaaS 授权和 API 访问控制：实施选项和最佳实践

AWS 规范性指导

AWS 规范性指导: 多租户 SaaS 授权和 API 访问控制：实施选项和最佳实践

Table of Contents

多租户 SaaS 授权和 API 访问控制：实施选项和最佳实践

Tabby Ward、Thomas Davis、Gideon Landeman 和 Amazon Web Services 的 Tomas Riha ()AWS

2024 年 5 月（文件历史记录）

授权和 API 访问控制是许多软件应用程序面临的挑战，尤其是多租户软件即服务 (SaaS) 应用程序。当你考虑到必须保护的微服务 API 激增以及来自不同租户、用户特征和应用程序状态的大量访问条件时，这种复杂性就显而易见了。为了有效地解决这些问题，解决方案必须对微服务、前端后端 (BFF) 层以及多租户 SaaS 应用程序的其他组件提供的许多 API 实施访问控制。这种方法必须辅之以一种能够根据许多因素和属性做出复杂的访问决策的机制。

传统上，API 访问控制和授权由应用程序代码中的自定义逻辑处理。这种方法容易出错且不安全，因为有权访问此代码的开发人员可能会意外或故意更改授权逻辑，从而导致未经授权的访问。审计应用程序代码中的自定义逻辑做出的决策很困难，因为审计员必须全身心投入到自定义逻辑中，才能确定其在维护任何特定标准方面的有效性。此外，API 访问控制通常是不必要的，因为需要保护的 API 不多。应用程序设计向偏向于微服务和面向服务的架构的范式转变增加了必须使用某种形式的授权和访问控制的 API 的数量。此外，需要在多租户 SaaS 应用程序中维护基于租户的访问权限，这给保留租赁带来了额外的授权挑战。本指南中概述的最佳实践具有以下几个好处：

- 授权逻辑可以集中化，并使用高级声明性语言编写，该语言不适用于任何编程语言。
- 授权逻辑是从应用程序代码中抽象出来的，可以作为可重复的模式应用于应用程序中的所有 API。
- 抽象可以防止开发人员意外更改授权逻辑。
- 与 SaaS 应用程序的集成既一致又简单。
- 通过抽象，无需为每个 API 端点编写自定义授权逻辑。
- 由于审计员不再需要审查代码来确定权限，因此简化了审计。
- 本指南中概述的方法支持根据组织的要求使用多种访问控制范例。
- 这种授权和访问控制方法为在 SaaS 应用程序的 API 层维护租户数据隔离提供了一种简单明了的方法。
- 在授权方面，最佳实践为租户的入职和离职提供了一种一致的方法。
- 这种方法提供了不同的授权部署模型（池化或孤岛），它们既有优点也有缺点，如本指南所述。

目标业务成果

本规范性指南描述了可重复的授权和 API 访问控制设计模式，这些模式可以用于多租户 SaaS 应用程序。本指南适用于任何开发具有复杂授权要求或严格的 API 访问控制需求的应用程序的团队。该架构详细说明了策略决策点 (PDP) 或策略引擎的创建以及 API 中策略执行点 (PEP) 的集成。讨论了创建 PDP 的两个具体选项：在 Cedar SDK 中使用亚马逊验证权限，以及使用带有 Rego 策略语言的开放政策代理 (OPA)。该指南还讨论了根据基于属性的访问控制 (ABAC) 模型或基于角色的访问控制 (RBAC) 模型或两种模型的组合做出访问决策。我们建议您使用本指南中提供的设计模式和概念来指导和标准化您在多租户 SaaS 应用程序中实现授权和 API 访问控制。本指南有助于实现以下业务成果：

- 适用于多租户 SaaS 应用程序的标准化 API 授权架构 — 该架构区分了三个组件：存储和管理策略的策略管理点 (PAP)、用于评估这些策略以做出授权决策的策略决策点 (PDP) 以及执行该决策的策略执行点 (PEP)。托管的授权服务 “已验证权限” 既是 PAP 又是 PDP。或者，您可以使用 Cedar 或 OPA 等开源引擎自己构建 PDP。
- 将@@ 授权逻辑与应用程序分离 — 授权逻辑嵌入到应用程序代码中或通过临时强制机制实现时，可能会受到意外或恶意更改的影响，从而导致无意的跨租户数据访问或其他安全漏洞。为了帮助减少这些可能性，您可以使用 PAP（例如 Verified Permissions）来存储独立于应用程序代码的授权策略，并对这些策略的管理进行强有力的监管。策略可以用高级声明性语言集中维护，这使得维护授权逻辑比在应用程序代码的多个部分中嵌入策略要简单得多。这种方法还可以确保以一致的方式应用更新。
- 灵活的访问控制模型方法 ——基于角色的访问控制 (RBAC)、基于属性的访问控制 (ABAC) 或两种模型的组合都是有效的访问控制方法。这些模型试图通过使用不同的方法来满足企业的授权要求。本指南对这些模型进行了比较和对比，以帮助您选择适合您的组织的模型。该指南还讨论了这些模型如何应用于不同的授权策略语言，例如 OPa/Rego 和 Cedar。本指南中讨论的架构可以成功采用其中一个或两个模型。
- 严格的 API 访问控制 — 本指南提供了一种在应用程序中以最少的努力一致且普遍地保护 API 的方法。这对于通常使用大量 API 来促进应用程序内部通信的面向服务的应用程序架构或微服务应用程序架构特别有价值。严格的 API 访问控制有助于提高应用程序的安全性，使其不易受到攻击或利用。

租户隔离和多租户授权

本指南介绍了租户隔离和多租户授权的概念。租户隔离是指您在 SaaS 系统中使用的明确机制，用于确保每个租户的资源（即使它们在共享基础架构上运行）是隔离的。多租户授权是指授权入站操作并防止在错误的租户上实施这些操作。假设的用户可以经过身份验证和授权，但仍然可以访问其他租户的资源。身份验证和授权不会阻止此访问——您需要实施租户隔离才能实现此目标。有关这两个概念之间差异的更广泛讨论，请参阅《SaaS 架构基础知识》白皮书的 “租户隔离” 部分。

访问控制的类型

您可以使用两个广泛定义的模型来实现访问控制：基于角色的访问控制 (RBAC) 和基于属性的访问控制 (ABAC)。每种型号都有优缺点，本节将简要讨论这些优缺点。您应该使用的模型取决于您的具体用例。本指南中讨论的架构支持这两种模型。

RBAC

基于角色的访问控制 (RBAC) 根据通常与业务逻辑一致的角色来确定对资源的访问权限。权限会根据需要与角色相关联。例如，营销角色将授权用户在受限系统内执行营销活动。这是一个相对简单的访问控制模型，因为它与易于识别的业务逻辑非常吻合。

在以下情况下，RBAC 模型的效果会降低：

- 您有独特的用户，其职责包括多个角色。
- 您的业务逻辑很复杂，因此很难定义角色。
- 要向上扩展到较大的规模，需要持续管理权限并将权限映射到新角色和现有角色。
- 授权基于动态参数。

ABAC

基于属性的访问控制 (ABAC) 根据属性确定对资源的访问权限。属性可以与用户、资源、环境甚至应用程序状态相关联。您的策略或规则引用属性，并且可以使用基本的布尔逻辑来确定是否允许用户执行操作。以下是权限的基本示例：

在支付系统中，财务部门的所有用户都可以在工作*/payments*时间通过 API 端点处理付款。

财务部门的成员资格是一种决定访问权限的用户属性/payments。还有一个与 /payments API 端点关联的资源属性，该属性仅允许在工作时间内进行访问。在ABAC中，用户能否处理付款由一项策略决定，该策略将财务部门成员资格作为用户属性，将时间作为资源属性。/payments

ABAC 模型在允许动态、上下文和精细的授权决策方面非常灵活。但是，ABAC模型最初很难实现。定义规则和策略以及列举所有相关访问向量的属性需要大量的前期投资才能实施。

RBAC-ABAC 混合方法

将 RBAC 和 ABAC 结合使用可以提供这两种模型的一些优势。RBAC 与业务逻辑非常接近，实施起来比 ABAC 更简单。为了在做出授权决策时提供额外的精细度，您可以将 ABAC 与 RBAC 结合使用。这种混合方法通过将用户的角色（及其分配的权限）与其他属性相结合来做出访问决策，从而确定访问权限。使用这两种模型可以简化权限的管理和分配，同时还可以提高与授权决策相关的灵活性和精细度。

访问控制模型比较

下表对前面讨论的三种访问控制模型进行了比较。这种比较旨在提供信息丰富且内容丰富。在特定情况下使用访问模型不一定与本表中的比较相关。

因子	RBAC	ABAC	混合
弹性	中	高	高
简单性	高	低	中
粒度	低	高	中
动态决策和规则	否	是	是
情境感知	否	是	有点像
实施工作	低	高	中

实施 PDP

政策决策点 (PDP) 可以描述为策略或规则引擎。此组件负责应用策略或规则，并返回有关是否允许特定访问的决定。PDP 可以与基于角色的访问控制 (RBAC) 和基于属性的访问控制 (ABAC) 模型一起运行；但是，PDP 是 ABAC 的必备条件。PDP 允许将应用程序代码中的授权逻辑转移到单独的系统中。这可以简化应用程序代码。它还提供了一个 easy-to-use 可重复的接口，用于为 API、微服务、前端后端 (BFF) 层或任何其他应用程序组件做出授权决策。

以下各节讨论实施 PDP 的三种方法。但是，这不是一个完整的清单。

PDP 实现方法：

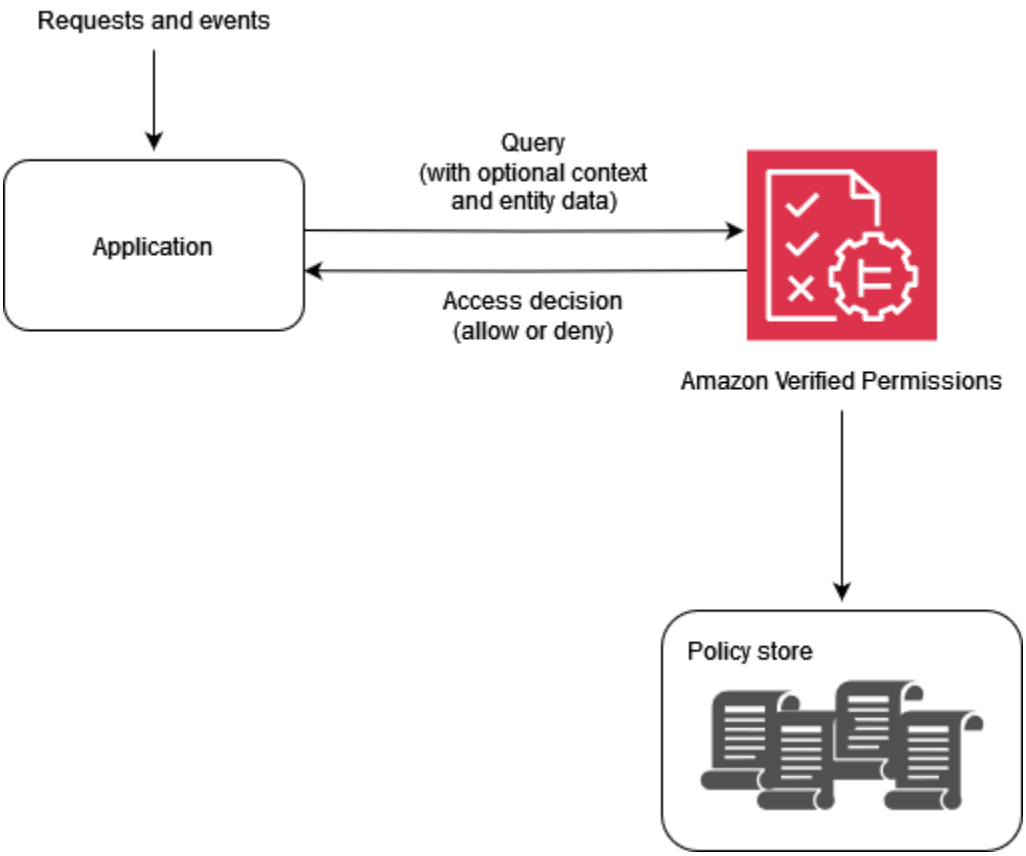
- 使用 Amazon 验证权限实施 PDP
- 使用 OPA 实现 PDP
- 使用自定义策略引擎

使用 Amazon 验证权限实施 PDP

Amazon Verified Permissions 是一项可扩展、精细的权限管理和授权服务，可用于实施政策决策点 (PDP)。作为策略引擎，它可以帮助您的应用程序实时验证用户操作，并突出显示权限过高或无效的权限。它通过外部化授权和集中策略管理和管理，帮助您的开发人员更快地构建更安全的应用程序。通过将授权逻辑与应用程序逻辑分开，Verified Permissions 支持策略解钩。

通过使用已验证的权限来实施 PDP，并在应用程序中实现最低权限和持续验证，开发人员可以使其应用程序访问权限与 Zero Trust 原则保持一致。此外，安全和审计团队可以更好地分析和审计谁有权访问应用程序中的哪些资源。Verified Permissions 使用 C ed ar（一种专门构建且安全至上的开源策略语言）根据基于角色的访问控制 (RBAC) 和基于属性的访问控制 (ABAC) 来定义基于策略的访问控制，以实现更精细的上下文感知访问控制。

已验证权限为 SaaS 应用程序提供了一些有用的功能，例如能够使用多个身份提供商（例如 Amazon Cognito、Google 和 Facebook）启用多租户授权。另一项对 SaaS 应用程序特别有用的已验证权限功能是支持基于每个租户的自定义角色。如果您正在设计客户关系管理 (CRM) 系统，则一个租户可能会根据一组特定的标准来定义销售机会访问的粒度。另一个租户可能有其他定义。已验证权限中的底层权限系统可以支持这些变体，这使其成为 SaaS 用例的绝佳选择。Verified Permissions 还支持编写适用于所有租户的策略，因此，作为 SaaS 提供商，可以直接应用护栏策略来防止未经授权的访问。



为什么要使用已验证的权限？

使用身份提供商（例如 Amazon Cognito）的已验证权限，为您的应用程序提供更加动态的、基于策略的访问管理解决方案。您可以构建应用程序来帮助用户共享信息和进行协作，同时维护其数据的安全性、机密性和隐私。Verified Permissions 为您提供精细的授权系统，可根据您的身份和资源的角色和属性强制执行访问权限，从而帮助您降低运营成本。您可以定义策略模型，在中心位置创建和存储策略，并在几毫秒内评估访问请求。

在已验证的权限中，您可以使用一种名为 Cedar 的简单、人类可读的声明性语言来表达权限。使用 Cedar 编写的策略可以在团队之间共享，无论每个团队的应用程序使用何种编程语言。

使用已验证权限时应考虑的事项

在已验证的权限中，您可以创建策略并将其作为配置的一部分自动执行。您还可以在运行时创建策略作为应用程序逻辑的一部分。作为最佳实践，在租户入职和置备过程中创建策略时，应使用持续集成和持续部署 (CI/CD) 管道来管理、修改和跟踪策略版本。或者，应用程序可以管理、修改和跟踪策略版本；但是，应用程序逻辑本身并不执行此功能。要在应用程序中支持这些功能，必须明确设计应用程序以实现此功能。

如果需要提供来自其他来源的外部数据才能做出授权决定，则必须检索这些数据并将其作为授权请求的一部分提供给 Verified Permissions。默认情况下，此服务不会检索其他上下文、实体和属性。

雪松概述

Cedar 是一种灵活、可扩展且可扩展的基于策略的访问控制语言，可帮助开发人员将应用程序权限表示为策略。管理员和开发人员可以定义允许或禁止用户对应用程序资源进行操作的策略。可以将多个策略附加到单个资源。当您的应用程序的用户尝试对资源执行操作时，您的应用程序会向 Cedar 策略引擎请求授权。Cedar 会评估适用的政策并返回`ALLOW`或`DENY`决定。Cedar 支持任何类型的委托人和资源的授权规则，允许基于角色的访问控制 (RBAC) 和基于属性的访问控制 (ABAC)，并支持通过自动推理工具进行分析。

Cedar 允许您将业务逻辑与授权逻辑分开。当您使用应用程序的代码发出请求时，您可以调用 Cedar 的授权引擎来确定该请求是否获得授权。如果获得授权（决定是`ALLOW`），则您的应用程序可以执行所请求的操作。如果未获得授权（决定是`DENY`），则您的应用程序可能会返回错误消息。Cedar 的主要特点包括：

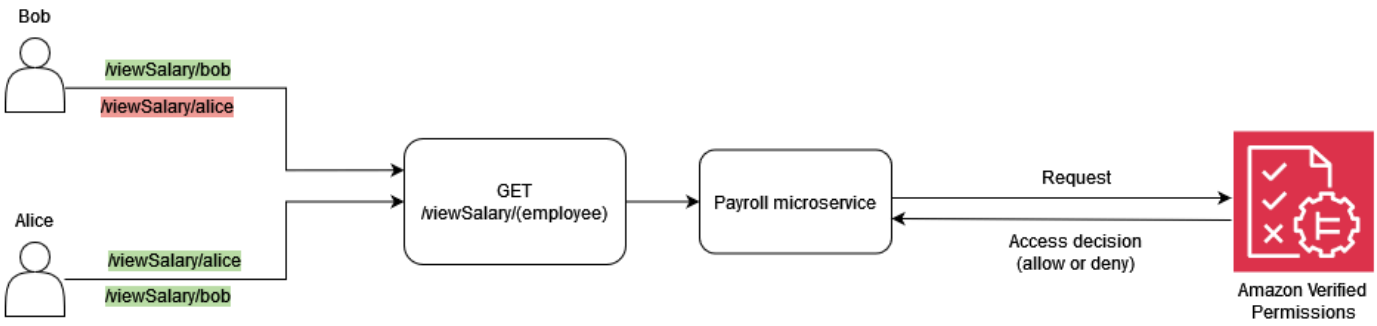
- 表现力 — Cedar 专为支持授权用例而设计，并且在开发时考虑了人类的可读性。
- 性能 — Cedar 支持索引策略以实现快速检索，并提供具有有限延迟的快速且可扩展的实时评估。
- 分析 — Cedar 支持分析工具，这些工具可以优化您的策略并验证您的安全模型。

欲了解更多信息，请访问 Cedar 网站。

示例 1：具有经过验证的权限和 Cedar 的基本 ABAC

在此示例场景中，Amazon 验证权限用于确定允许哪些用户访问虚构的 Payroll 微服务中的信息。本节包含 Cedar 代码片段，用于演示如何使用 Cedar 来做出访问控制决策。这些示例并不是为了全面探索 Cedar 和已验证权限提供的功能。有关 Cedar 的更全面概述，请参阅 Cedar 文档。

在下图中，我们想强制执行与该`viewSalaryGET`方法相关的两条一般业务规则：员工可以查看自己的工资，员工可以查看向他们汇报的任何人的工资。您可以使用已验证的权限策略来强制执行这些业务规则。



员工可以查看自己的工资。

在 Cedar 中，基本结构是一个实体，它代表委托人、行动或资源。要提出授权请求并使用已验证权限策略开始评估，您需要提供委托人、操作、资源和实体列表。

- 委托人 (principal) 是登录的用户或角色。
- 操作 (action) 是请求所评估的操作。
- 资源 (resource) 是操作正在访问的组件。
- 实体列表 (entityList) 包含评估请求所需的所有必需实体。

为了满足业务规则员工可以查看自己的工资，您可以提供如下所示的已验证权限策略。

```
permit (
    principal,
    action == Action::"viewSalary",
    resource
)
when {
    principal == resource.owner
};
```

此策略的评估结果Action是viewSalary，请求中的资源ALLOW是否具有等于委托人的属性所有者。例如，如果 Bob 是请求薪金报告的登录用户，同时也是薪金报告的所有者，则策略的评估结果为。ALLOW

以下授权请求已提交给已验证的权限，以供示例策略进行评估。在此示例中，Bob 是viewSalary发出请求的登录用户。因此，Bob 是实体类型的主体Employee。Bob 正在尝试执行的操作是viewSalary,，viewSalary将显示的资源Salary-Bob与类型相同Salary。为了评估 Bob 是否可以查看Salary-Bob资源，您需要提供一个实体结构，该结构将值为的类型 EmployeeBob（委托人）链接到具有该类型的资源的所有者属性Salary。您可以在中提供此结构entityList，其

中与之关联的属性Salary包括所有者，所有者指定entityIdentifier包含类型Employee和值的Bob。Verified Permissions 将授权请求中principal提供的权限与与Salary资源关联的owner属性进行比较，以做出决策。

```
{
  "policyStoreId": "PAYROLLAPP_POLICYSTOREID",
  "principal": {
    "entityType": "PayrollApp::Employee",
    "entityId": "Bob"
  },
  "action": {
    "actionType": "PayrollApp::Action",
    "actionId": "viewSalary"
  },
  "resource": {
    "entityType": "PayrollApp::Salary",
    "entityId": "Salary-Bob"
  },
  "entities": {
    "entityList": [
      {
        "identifier": {
          "entityType": "PayrollApp::Salary",
          "entityId": "Salary-Bob"
        },
        "attributes": {
          "owner": {
            "entityIdentifier": {
              "entityType": "PayrollApp::Employee",
              "entityId": "Bob"
            }
          }
        }
      },
      {
        "identifier": {
          "entityType": "PayrollApp::Employee",
          "entityId": "Bob"
        },
        "attributes": {}
      }
    ]
  }
```

```
}
```

对已验证权限的授权请求将以下内容作为输出返回，其中属性decision为ALLOW或DENY。

```
{
    "determiningPolicies":
        [
            {
                "determiningPolicyId": "PAYROLLAPP_POLICYSTOREID"
            }
        ],
    "decision": "ALLOW",
    "errors": []
}
```

在本例中，由于 Bob 正在尝试查看自己的工资，因此发送给 “已验证权限” 的授权请求计算为ALLOW。但是，我们的目标是使用经过验证的权限来强制执行两项业务规则。规定以下内容的业务规则也应为真：

员工可以查看向他们汇报的任何人的工资。

为了满足此业务规则，您可以提供其他策略。以下策略评估操作ALLOW是否为viewSalary，请求中的资源是否具有等owner.manager于委托人的属性。例如，如果 Alice 是请求薪资报告的登录用户，而 Alice 是报告所有者的经理，则策略的评估结果为。ALLOW

```
permit (
    principal,
    action == Action::"viewSalary",
    resource
)
when {
    principal == resource.owner.manager
};
```

以下授权请求已提交给已验证的权限，以供示例策略进行评估。在此示例中，Alice 是viewSalary发出请求的登录用户。因此，Alice 是委托人，而实体属于这种类型Employee。Alice 尝试执行的操作是viewSalary，viewSalary将显示的资源属于值为Salary的类型Salary-Bob。为了评估 Alice 能否查看Salary-Bob资源，您需要提供一个实体结构，该结构将值为的类型Employee链接Alice到该manager属性，然后该owner属性必须与值为的类型的Salary属性相关联Salary-Bob。您可以在中提供此结构entityList，其中与之关联的属性Salary包括所有者，所有者指

定entityIdentifier包含类型Employee和值的Bob。“已验证权限” 首先检查owner属性，该属性将根据类型Employee和值Bob进行计算。然后，Verified Permissions 会评估Employee与之关联的manager属性，并将其与提供的委托人进行比较，以做出授权决定。在这种情况下，之所以做出决定，ALLOW是因为principal 和resource.owner.manager属性是等效的。

```
{
  "policyStoreId": "PAYROLLAPP_POLICYSTOREID",
  "principal": {
    "entityType": "PayrollApp::Employee",
    "entityId": "Alice"
  },
  "action": {
    "actionType": "PayrollApp::Action",
    "actionId": "viewSalary"
  },
  "resource": {
    "entityType": "PayrollApp::Salary",
    "entityId": "Salary-Bob"
  },
  "entities": {
    "entityList": [
      {
        "identifier": {
          "entityType": "PayrollApp::Employee",
          "entityId": "Alice"
        },
        "attributes": {
          "manager": {
            "entityIdentifier": {
              "entityType": "PayrollApp::Employee",
              "entityId": "None"
            }
          }
        },
        "parents": []
      },
      {
        "identifier": {
          "entityType": "PayrollApp::Salary",
          "entityId": "Salary-Bob"
        },
        "attributes": {
          "owner": {
```

```
            "entityIdentifier": {
              "entityType": "PayrollApp::Employee",
              "entityId": "Bob"
            }
          }
        },
        "parents": []
      },
      {
        "identifier": {
          "entityType": "PayrollApp::Employee",
          "entityId": "Bob"
        },
        "attributes": {
          "manager": {
            "entityIdentifier": {
              "entityType": "PayrollApp::Employee",
              "entityId": "Alice"
            }
          }
        },
       "parents": []
      }
    ]
  }
}
```

到目前为止，在本示例中，我们提供了与该viewSalary方法相关的两个业务规则，员工可以查看自己的工资，员工可以查看向他们汇报的任何人的工资，将已验证权限作为策略来独立满足每条业务规则的条件。您也可以使用单个 “已验证权限” 策略来满足两个业务规则的条件：

员工可以查看自己的工资和向他们汇报的任何人的工资。

当您使用之前的授权请求时，以下策略的计算结果是操作是viewSalary，请求中的资源ALLOW是否具有等于的属性 principalowner，还是等于的principal属性。owner.manager

```
permit (
    principal,
    action == PayrollApp::Action::"viewSalary",
    resource
)
when {
    principal == resource.owner.manager ||
```

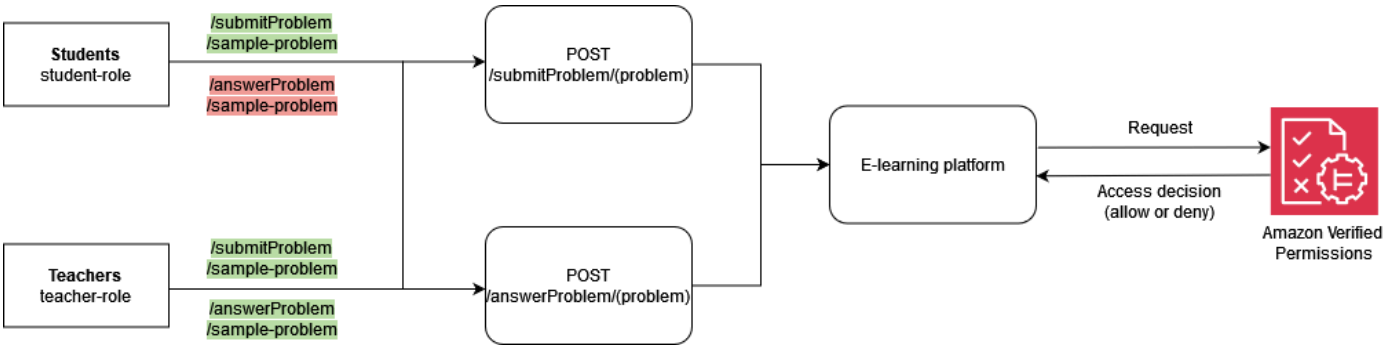
```
    principal == resource.owner
};
```

例如，如果 Alice 是请求薪资报告的登录用户，如果 Alice 是报告所有者的经理或所有者，则策略的评估结果为。ALLOW

有关在 Cedar 策略中使用逻辑运算符的更多信息，请参阅 Cedar 文档。

示例 2：具有经过验证的权限和 Cedar 的基本 RBAC

此示例使用已验证的权限和 Cedar 来演示基本的 RBAC。如前所述，Cedar的基本构造是一个实体。开发人员定义自己的实体，并且可以选择在实体之间创建关系。以下示例包括三种类型的实体：UsersRoles、和Problems。 Students并且Teachers可以被视为该类型的实体Role，，每个实体User都可以与零或任何一个相关联Roles。



在 Cedar 中，这些关系是通过将链接RoleStudent到UserBob作为其父项来表达的。这种关联在逻辑上将所有学生用户分组到一个组中。有关在 Cedar 中进行分组的更多信息，请参阅 Cedar 文档。

以下策略评估与该类型的逻辑组相关联的所有委托Students人的操作submitProblem,决策ALLOW。Role

```
permit (
    principal in ElearningApp::Role::"Students",
    action == ElearningApp::Action::"submitProblem",
    resource
);
```

以下策略根据操作的决策进行评估answerProblem，submitProblem或者ALLOW针对与该类型的逻辑组相关联的所有委托Teachers人进行评估。Role

```
permit (
    principal in ElearningApp::Role::"Teachers",
    action in [
```

```
        ElearningApp::Action::"submitProblem",
        ElearningApp::Action::"answerProblem"
    ],
    resource
);
```

为了评估使用这些策略的请求，评估引擎需要知道授权请求中引用的委托人是否为相应组的成员。因此，应用程序必须将相关的群组成员资格信息作为授权请求的一部分传递给评估引擎。这是通过属性完成的，它使您能够向 Cedar 评估引擎提供授权调用中涉及的委托人和资源的属性和组成员身份数据。`entities`在以下代码中，组成员资格通过定义`User::"Bob"`为调用父代来表示`Role::"Students"`。

```
{
  "policyStoreId": "ELEARNING_POLICYSTOREID",
  "principal": {
    "entityType": "ElearningApp::User",
    "entityId": "Bob"
  },
  "action": {
    "actionType": "ElearningApp::Action",
    "actionId": "answerProblem"
  },
  "resource": {
    "entityType": "ElearningApp::Problem",
    "entityId": "SomeProblem"
  },
  "entities": {
    "entityList": [
        {
            "identifier": {
                "entityType": "ElearningApp::User",
                "entityId": "Bob"
            },
            "attributes": {},
            "parents": [
                {
                    "entityType": "ElearningApp::Role",
                    "entityId": "Students"
                }
            ]
        },
        {
          "identifier": {
```

```
            "entityType": "ElearningApp::Problem",
            "entityId": "SomeProblem"
          },
          "attributes": {},
          "parents": []
        }
      ]
  }
}
```

在此示例中，Bob 是answerProblem发出请求的登录用户。因此，Bob 是委托人，而实体属于该类型User。Bob 想要执行的操作是answerProblem。为了评估 Bob 能否执行answerProblem操作，您需要提供一个实体结构，该结构将该实体User与值关联起来，Bob并通过将父实体列为Role::"Students"，来分配其组成员资格。由于用户组Role::"Students"中的实体只能执行操作，因此此授权请求的计算结果为。submitProblem DENY

另一方面，如果值为Alice且属于该组的类型UserRole::"Teachers"尝试执行操作，则授权请求的计算结果为ALLOW，因为该策略规定允许组中的委托人对Role::"Teachers"所有资源执行操作。answerProblem answerProblem以下代码显示了这种类型的授权请求，其计算结果为。ALLOW

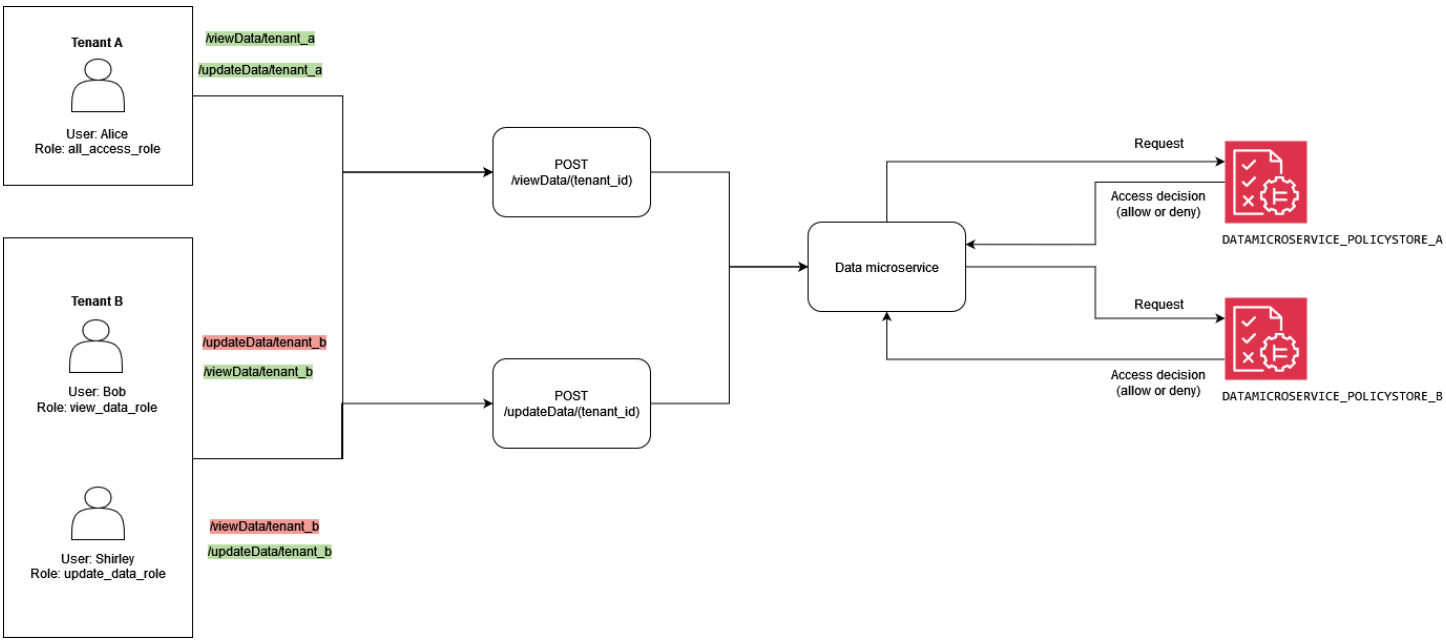
```
{
  "policyStoreId": "ELEARNING_POLICYSTOREID",
  "principal": {
    "entityType": "ElearningApp::User",
    "entityId": "Alice"
  },
  "action": {
    "actionType": "ElearningApp::Action",
    "actionId": "answerProblem"
  },
  "resource": {
    "entityType": "ElearningApp::Problem",
    "entityId": "SomeProblem"
  },
  "entities": {
    "entityList": [
        {
            "identifier": {
                "entityType": "ElearningApp::User",
                "entityId": "Alice"
            },
            "attributes": {},
```

```
            "parents": [
                {
                    "entityType": "ElearningApp::Role",
                    "entityId": "Teachers"
                }
            ]
        },
        {
            "identifier": {
                "entityType": "ElearningApp::Problem",
                "entityId": "SomeProblem"
            },
            "attributes": {},
            "parents": []
        }
      ]
  }
}
```

示例 3：使用 RBAC 进行多租户访问控制

要详细说明前面的 RBAC 示例，您可以扩展您的要求以包括 SaaS 多租户，这是 SaaS 提供商的常见要求。在多租户解决方案中，总是代表给定租户提供资源访问权限。也就是说，租户 A 的用户无法查看租户 B 的数据，即使这些数据在逻辑上或物理上并置在系统中。以下示例说明了如何使用多个已验证权限策略存储来实现租户隔离，以及如何使用用户角色在租户内定义权限。

使用每租户策略存储设计模式是在使用已验证权限实施访问控制的同时保持租户隔离的最佳实践。在这种情况下，租户 A 和租户 B 的用户请求将分别根据不同的策略存储DATAMICROSERVICE_POLICYSTORE_A和进行DATAMICROSERVICE_POLICYSTORE_B验证。有关多租户 SaaS 应用程序的已验证权限设计注意事项的更多信息，请参阅已验证权限多租户设计注意事项部分。



以下策略位于DATAMICROSERVICE_POLICYSTORE_A策略存储中。它验证主体是否将成为该类型Role组allAccessRole的一部分。在这种情况下，将允许委托人对与租户 A 关联的所有资源执行viewData和updateData操作。

```
permit (
    principal in MultitenantApp::Role::"allAccessRole",
    action in [
        MultitenantApp::Action::"viewData",
        MultitenantApp::Action::"updateData"
    ],
    resource
);
```

以下策略位于DATAMICROSERVICE_POLICYSTORE_B策略存储中。第一个策略验证委托人是否属于该类型updateDataRoleRole组。假设是这样，则它允许委托人对与租户 B 关联的资源执行操作。updateData

```
permit (
    principal in MultitenantApp::Role::"updateDataRole",
    action == MultitenantApp::Action::"updateData",
    resource
);
```

第二项政策规定，Role应允许属于该类型viewDataRole组的委托人对与租户 viewData B 关联的资源执行操作。

```
permit (
    principal in MultitenantApp::Role::"viewDataRole",
    action == MultitenantApp::Action::"viewData",
    resource
);
```

租户 A 发出的授权请求需要发送到DATAMICROSERVICE_POLICYSTORE_A策略存储区，并由属于该存储的策略进行验证。在本例中，已通过前面作为本示例一部分讨论的第一个策略对其进行验证。在此授权请求中，类型User为的委托人Alice正在请求执行viewData操作。主体属于allAccessRole类型组Role。Alice 正在尝试对SampleData资源执行viewData操作。由于 Alice 有这个allAccessRole角色，所以这个评估会产生一个ALLOW决策。

```
{
  "policyStoreId": "DATAMICROSERVICE_POLICYSTORE_A",
  "principal": {
      "entityType": "MultitenantApp::User",
      "entityId": "Alice"
  },
  "action": {
      "actionType": "MultitenantApp::Action",
      "actionId": "viewData"
  },
  "resource": {
      "entityType": "MultitenantApp::Data",
      "entityId": "SampleData"
  },
  "entities": {
    "entityList": [
      {
        "identifier": {
            "entityType": "MultitenantApp::User",
            "entityId": "Alice"
        },
        "attributes": {},
        "parents": [
            {
                "entityType": "MultitenantApp::Role",
                "entityId": "allAccessRole"
            }
        ]
      },
      {
```

```
        "identifier": {
            "entityType": "MultitenantApp::Data",
            "entityId": "SampleData"
        },
        "attributes": {},
        "parents": []
      }
    ]
  }
}
```

相反，如果您查看租户 B 发出的请求User Bob，则会看到类似以下授权请求的内容。该请求之所以被发送到DATAMICROSERVICE_POLICYSTORE_B策略存储，是因为它来自租户 B。在此请求中，委托Bob人想要updateData对资源SampleData执行操作。但是，Bob不属于有权访问该资源操作updateData的群组。因此，该请求会导致决DENY策。

```
{
  "policyStoreId": "DATAMICROSERVICE_POLICYSTORE_B",
  "principal": {
      "entityType": "MultitenantApp::User",
      "entityId": "Bob"
  },
  "action": {
      "actionType": "MultitenantApp::Action",
      "actionId": "updateData"
  },
  "resource": {
      "entityType": "MultitenantApp::Data",
      "entityId": "SampleData"
  },
  "entities": {
    "entityList": [
      {
        "identifier": {
            "entityType": "MultitenantApp::User",
            "entityId": "Bob"
        },
        "attributes": {},
        "parents": [
            {
                "entityType": "MultitenantApp::Role",
                "entityId": "viewDataRole"
            }
```

```
        ]
      },
      {
        "identifier": {
            "entityType": "MultitenantApp::Data",
            "entityId": "SampleData"
        },
        "attributes": {},
        "parents": []
      }
    ]
  }
}
```

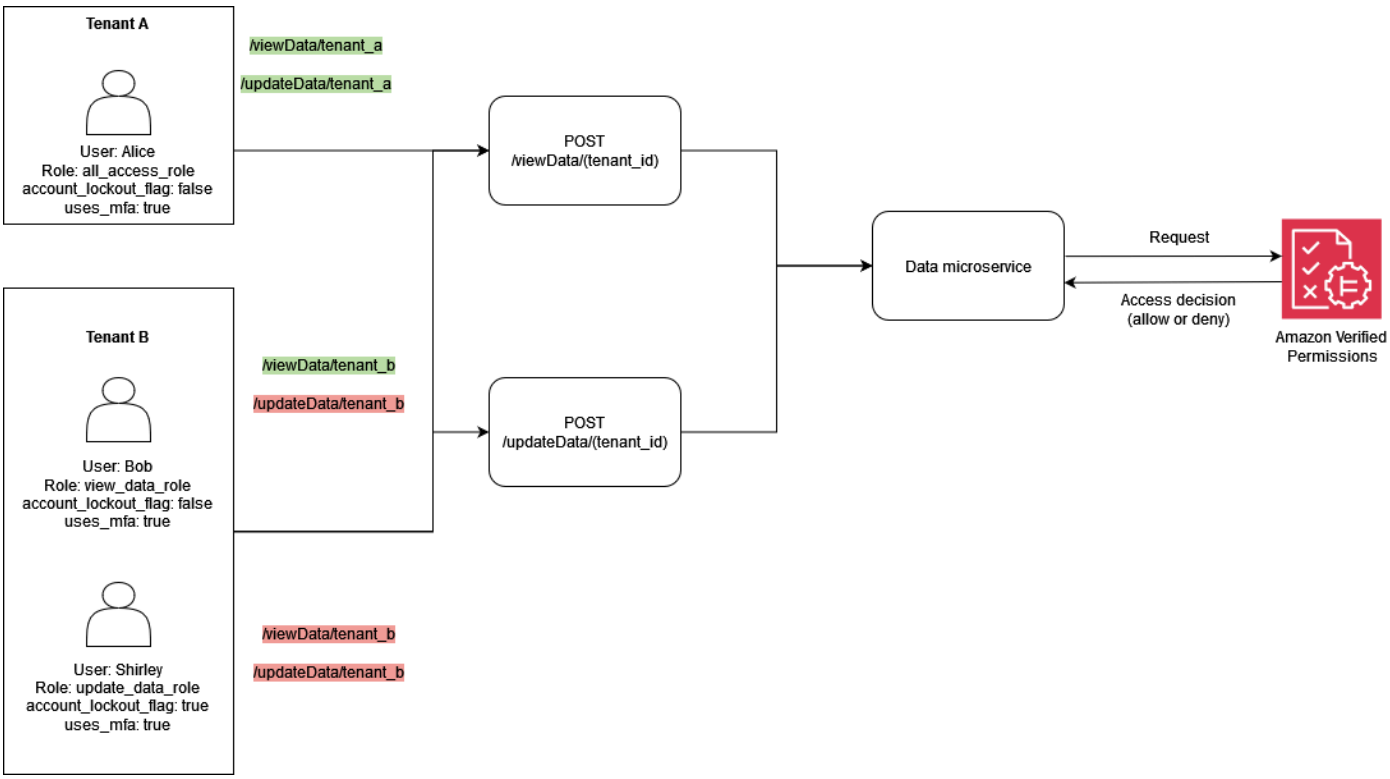
在第三个示例中，User Alice 尝试对资源执行viewData操作SampleData。此请求被定向到DATAMICROSERVICE_POLICYSTORE_A策略存储，因为委托人Alice属于租户 AAlice，属于该allAccessRole类型的组Role，允许她对资源执行操作。viewData因此，请求会导致ALLOW决策。

```
{
  "policyStoreId": "DATAMICROSERVICE_POLICYSTORE_A",
  "principal": {
      "entityType": "MultitenantApp::User",
      "entityId": "Alice"
  },
  "action": {
      "actionType": "MultitenantApp::Action",
      "actionId": "viewData"
  },
  "resource": {
      "entityType": "MultitenantApp::Data",
      "entityId": "SampleData"
  },
  "entities": {
    "entityList": [
      {
        "identifier": {
            "entityType": "MultitenantApp::User",
            "entityId": "Alice"
        },
        "attributes": {},
        "parents": [
            {
```

```
                "entityType": "MultitenantApp::Role",
                "entityId": "allAccessRole"
            }
        ]
      },
      {
        "identifier": {
            "entityType": "MultitenantApp::Data",
            "entityId": "SampleData"
        },
        "attributes": {},
        "parents": []
      }
    ]
  }
}
```

示例 4：使用 RBAC 和 ABAC 进行多租户访问控制

为了增强上一节中的 RBAC 示例，您可以向用户添加属性，以创建用于多租户访问控制的 RBAC-ABAC 混合方法。此示例包含与上一个示例相同的角色，但添加了用户属`account_lockout_flag`性和上下文参数`uses_mfa`。该示例还采用了不同的方法来实现多租户访问控制，即同时使用 RBAC 和 ABAC，并为每个租户使用一个共享策略存储而不是不同的策略存储。



此示例表示一种多租户 SaaS 解决方案，在该解决方案中，您需要为租户 A 和租户 B 提供授权决策，与前面的示例类似。

为了实现用户锁定功能，该示例在授权请求中account_lockout_flag向User实体委托人添加了该属性。此标志锁定用户对系统的访问权限，并将DENY所有权限授予被锁定的用户。该account_lockout_flag属性与User实体相关联，在多个会话中主动撤消该标志User之前一直有效。该示例使用when条件进行评估account_lockout_flag。

该示例还添加了有关请求和会话的详细信息。上下文信息指定已使用多因素身份验证对会话进行身份验证。为了实现此验证，该示例使用when条件来评估作为上下文字段一部分的uses_mfa标志。有关添加上下文的最佳做法的更多信息，请参阅 Cedar 文档。

```
permit (
    principal in MultitenantApp::Role::"allAccessRole",
    action in [
        MultitenantApp::Action::"viewData",
        MultitenantApp::Action::"updateData"
    ],
    resource
)
when {
    principal.account_lockout_flag == false &&
    context.uses_mfa == true &&
    resource in principal.Tenant
};
```

除非资源与请求委托人的Tenant属性属于同一组，否则此策略禁止访问资源。这种维护租户隔离的方法被称为 “一个共享的多租户策略存储” 方法。有关多租户 SaaS 应用程序的已验证权限设计注意事项的更多信息，请参阅已验证权限多租户设计注意事项部分。

该政策还确保委托人是allAccessRole和的成员，并将操作限制为viewData和。updateData此外，此策略还会验证即account_lockout_flag为false以及上下文值的uses_mfa计算结果是否为。true

同样，以下策略可确保委托人和资源都与同一个租户相关联，从而防止跨租户访问。该政策还确保委托人是其成员，viewDataRole并将操作限制为。viewData此外，它还会验证是否account_lockout_flag为，false以及的上下文值的uses_mfa计算结果是否为。true

```
permit (
    principal in MultitenantApp::Role::"viewDataRole",
    action == MultitenantApp::Action::"viewData",
```

```
    resource
)
when {
    principal.account_lockout_flag == false &&
    context.uses_mfa == true &&
    resource in principal.Tenant
};
```

第三个策略与前一个策略类似。该策略要求资源必须是与所代表的实体相对应的组的成员principal.Tenant。这样可以确保委托人和资源都与租户 B 相关联，从而防止跨租户访问。该政策确保委托人是其成员，updateDataRole并限制其操作。updateData此外，此策略还会验证是否account_lockout_flag为，false以及的上下文值是否uses_mfa计算为。true

```
permit (
    principal in MultitenantApp::Role::"updateDataRole",
    action == MultitenantApp::Action::"updateData",
    resource
)
when {
    principal.account_lockout_flag == false &&
    context.uses_mfa == true &&
    resource in principal.Tenant
};
```

以下授权请求由本节前面讨论的三个策略进行评估。在此授权请求中，类型为User且值为的委托人向该角色Alice发出updateData请求allAccessRole。 Alice的属性的Tenant值为Tenant::"TenantA"。正在尝试执行的操作Alice是，updateData,并且要应用该操作SampleData的资源属于该类型Data。 SampleData已TenantA作为父实体。

根据策略存储中的第一个<DATAMICROSERVICE_POLICYSTOREID>策略，Alice可以对资源执行操作，前提是该策略的when子句中的条件得到满足。updateData第一个条件要求该principal.Tenant属性计算为TenantA。第二个条件要求委托人的属性account_lockout_flag为false。最后一个条件要求上下文uses_mfa必须是true。由于所有三个条件都已满足，因此请求会返回一个ALLOW决定。

```
{
  "policyStoreId": "DATAMICROSERVICE_POLICYSTORE",
  "principal": {
      "entityType": "MultitenantApp::User",
      "entityId": "Alice"
  },
```

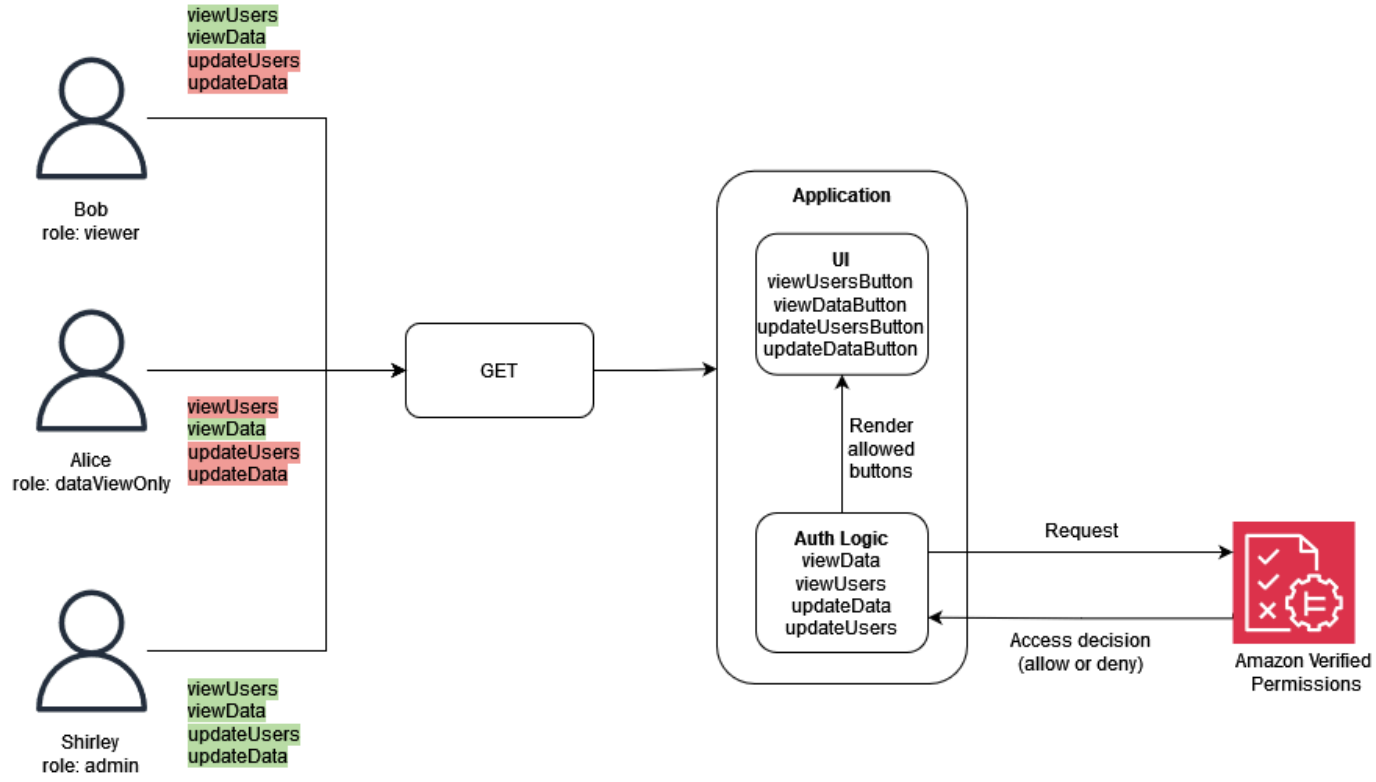
```
  "action": {
      "actionType": "MultitenantApp::Action",
      "actionId": "updateData"
  },
  "resource": {
      "entityType": "MultitenantApp::Data",
      "entityId": "SampleData"
  },
  "context": {
    "contextMap": {
        "uses_mfa": {
            "boolean": true
        }
    }
  },
  "entities": {
    "entityList": [
      {
        "identifier": {
            "entityType": "MultitenantApp::User",
            "entityId": "Alice"
        },
        "attributes": {
            {
                "account_lockout_flag": {
                    "boolean": false
                },
                "Tenant": {
                   "entityIdentifier": {
                        "entityType":"MultitenantApp::Tenant",
                        "entityId":"TenantA"
                   }
                }
            }
        },
        "parents": [
            {
                "entityType": "MultitenantApp::Role",
                "entityId": "allAccessRole"
            }
        ]
        },
     {
        "identifier": {
```

```
            "entityType": "MultitenantApp::Data",
            "entityId": "SampleData"
        },
        "attributes": {},
        "parents": [
            {
                "entityType": "MultitenantApp::Tenant",
                "entityId": "TenantA"
            }
        ]
      }
    ]
  }
}
```

示例 5：使用已验证权限和 Cedar 进行用户界面筛选

您还可以使用已验证的权限根据授权的操作对用户界面元素实施 RBAC 筛选。对于具有上下文敏感用户界面元素的应用程序来说，这非常有价值，这些元素可能与特定用户或租户相关联，如果是多租户 SaaS 应用程序。

在以下示例中`Users`，`Roleviewer`不允许执行更新。对于这些用户，用户界面不应呈现任何更新按钮。



在此示例中，单页 Web 应用程序有四个按钮。哪些按钮可见取决于当前登录应用程序的用户。Role在单页 Web 应用程序呈现 UI 时，它会查询 “已验证权限” 以确定用户有权执行哪些操作，然后根据授权决定生成按钮。

以下策略指定值为Role的类型viewer可以同时查看用户和数据。此策略的ALLOW授权决策需要viewData或viewUsers操作，还需要将资源与Data或类型相关联Users。ALLOW决定允许 UI 呈现两个按钮：viewDataButton和viewUsersButton。

```
permit (
    principal in GuiAPP::Role::"viewer",
    action in [GuiAPP::Action::"viewData", GuiAPP::Action::"viewUsers"],
    resource
)
when {
   resource in [GuiAPP::Type::"Data", GuiAPP::Type::"Users"]
};
```

以下策略指定值为Role的类型viewerDataOnly只能查看数据。此策略的ALLOW授权决策需要viewData采取行动，还需要与该类型关联的资源Data。ALLOW决定允许 UI 呈现按钮viewDataButton。

```
permit (
    principal in GuiApp::Role::"viewerDataOnly",
    action in [GuiApp::Action::"viewData"],
    resource in [GuiApp::Type::"Data"]
);
```

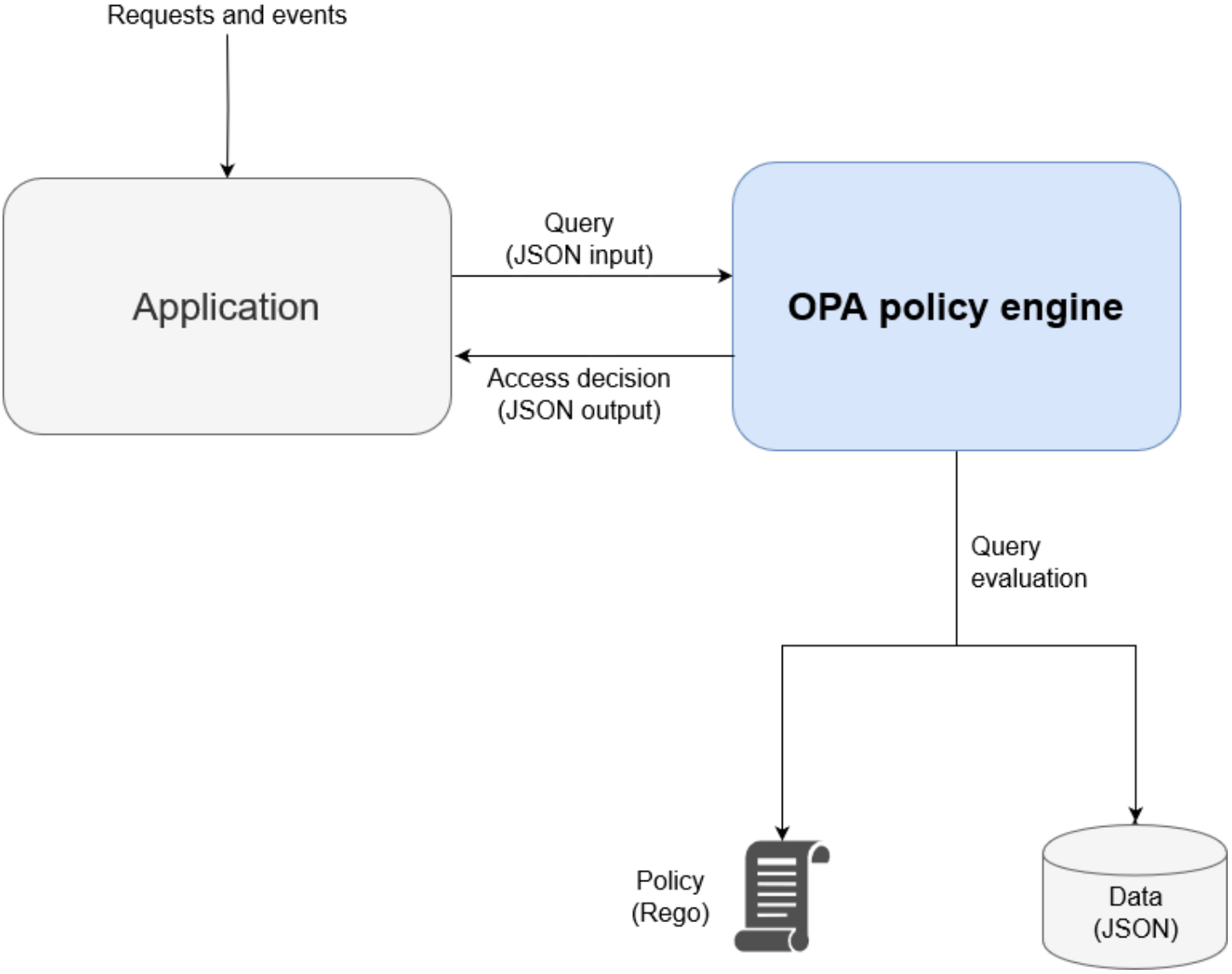
以下策略指定值为Role的类型admin可以编辑和查看数据和用户。此策略的ALLOW授权决策需要updateDataupdateUsers、viewData,或的操作viewUsers，并且还需要将资源与Data或类型相关联Users。ALLOW决定允许 UI 呈现所有四个按钮：updateDataButtonupdateUsersButton、viewDataButton、和viewUsersButton。

```
permit (
    principal in GuiApp::Role::"admin",
    action in [
        GuiApp::Action::"updateData",
        GuiApp::Action::"updateUsers",
        GuiApp::Action::"viewData",
        GuiApp::Action::"viewUsers"
       ],
    resource
```

```
)
when {
   resource in [GuiApp::Type::"Data", GuiApp::Type::"Users"]
};
```

使用 OPA 实现 PDP

开放政策代理 (OPA) 是一个开源的通用策略引擎。OPA 有很多用例，但与 PDP 实现相关的用例是它能够将授权逻辑与应用程序分离。这称为政策脱钩。OPA 在实施 PDP 时很有用，原因有很多。它使用一种名为 Rego 的高级声明性语言来起草政策和规则。这些策略和规则与应用程序分开存在，可以在没有任何特定于应用程序的逻辑的情况下做出授权决策。OPA 还公开了一个 RESTful API，使检索授权决策变得简单明了。要做出授权决定，应用程序会使用 JSON 输入查询 OPA，然后 OPA 根据指定的策略评估输入，以 JSON 格式返回访问决策。OPA 还能够导入可能与做出授权决定相关的外部数据。



与自定义策略引擎相比，OPA 有几个优点：

- OPA 及其对 Rego 的政策评估提供了一个灵活的、预先构建的策略引擎，它只需要插入策略和做出授权决策所需的任何数据。必须在自定义策略引擎解决方案中重新创建此策略评估逻辑。
- OPA 使用声明性语言编写策略，从而简化了授权逻辑。您可以独立于任何应用程序代码修改和管理这些策略和规则，无需具备应用程序开发技能。
- OPA 公开了一个 RESTful API，它简化了与策略执行点 (PEP) 的集成。
- OPA 为验证和解码 JSON 网络令牌 (JWT) 提供了内置支持。
- OPA 是一种公认的授权标准，这意味着如果您需要帮助或研究来解决特定问题，则可以获得大量文档和示例。
- 采用诸如 OPA 之类的授权标准，无论团队的应用程序使用何种编程语言，都可以在团队之间共享以 Rego 编写的策略。

OPA 不会自动提供以下两项内容：

- OPA 没有用于更新和管理策略的强大控制平面。OPA 确实提供了一些通过公开管理 API 来实现策略更新、监控和日志聚合的基本模式，但是与此 API 的集成必须由 OPA 用户处理。作为最佳实践，您应该使用持续集成和持续部署 (CI/CD) 管道来管理、修改和跟踪策略版本并管理 OPA 中的策略。
- 默认情况下，OPA 无法从外部来源检索数据。授权决策的外部数据源可以是保存用户属性的数据库。向OPA提供外部数据的方式有一定的灵活性——可以提前在本地缓存这些数据，也可以在请求授权决定时从API动态检索——但是获取这些信息不是OPA可以代表你做的事情。

Rego 概述

Rego 是一种通用策略语言，这意味着它适用于堆栈的任何层和任何域。Rego 的主要目的是接受 JSON/YAML 输入和数据，这些输入和数据经过评估，以便就基础设施资源、身份和运营做出基于策略的决策。Rego 使您能够编写有关堆栈或域中任何层的策略，而无需更改或扩展语言。以下是 Rego 可以做出的决策的一些示例：

- 此 API 请求是允许还是被拒绝？
- 此应用程序的备份服务器的主机名是什么？
- 提议的基础设施变更的风险评分是多少？
- 为了实现高可用性，应将此容器部署到哪些集群？
- 此微服务应使用哪些路由信息？

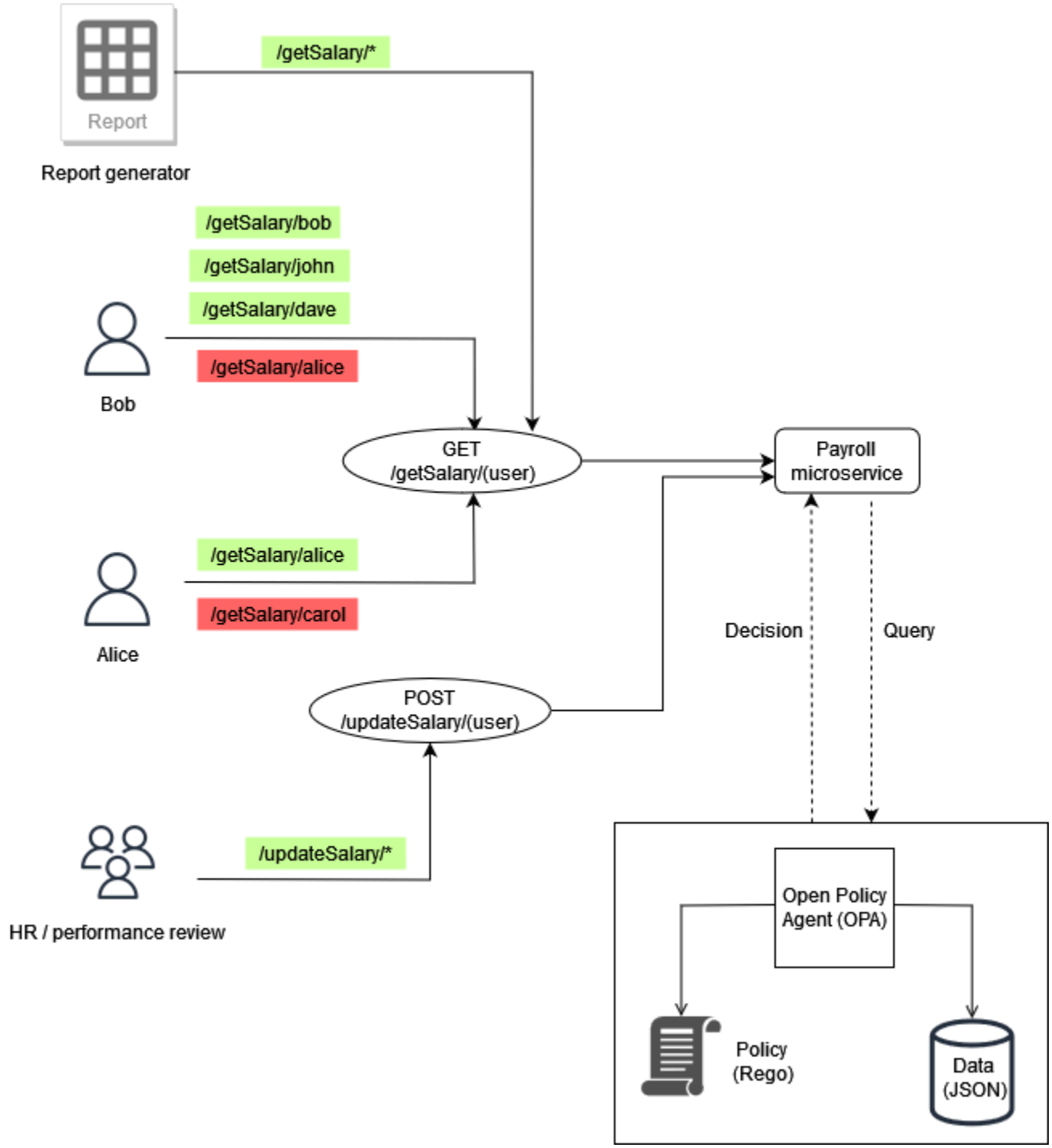
为了回答这些问题，Rego采用了关于如何做出这些决定的基本理念。在 Rego 中起草政策时的两个关键原则是：

- 每个资源、身份或操作都可以表示为 JSON 或 YAML 数据。
- 策略是应用于数据的逻辑。

Rego 通过定义有关如何评估 JSON/YAML 数据输入的逻辑，帮助软件系统做出授权决策。C、Java、Go 和 Python 等编程语言是解决这个问题的常用方法，但是 Rego 的设计重点是代表你的系统的数据和输入，以及使用这些信息做出政策决策的逻辑。

示例 1：带有 OPA 和 Rego 的基本 ABAC

本节介绍一种场景，其中使用 OPA 来决定允许哪些用户访问虚构的 Payroll 微服务中的信息。提供 Rego 代码片段是为了演示如何使用 Rego 来呈现访问控制决策。这些例子既不是详尽无遗的，也不是对Rego和OPA能力的全面探索。要更全面地了解 Rego，我们建议您查阅 OPA 网站上的 Rego 文档。



基本的 OPA 规则示例

在上图中，OPA 为工资单微服务强制执行的访问控制规则之一是：

员工可以读取自己的工资。

如果 Bob 尝试访问工资单微服务以查看自己的工资，工资单微服务可以将 API 调用重定向到 OPA RESTful API 以做出访问决定。工资单服务使用以下 JSON 输入查询 OPA 以获取决策：

```
{
    "user": "bob",
    "method": "GET",
    "path": ["getSalary", "bob"]
}
```

OPA 根据查询选择一个或多个策略。在这种情况下，以下用 Rego 编写的策略将评估 JSON 输入。

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

默认情况下，此策略拒绝访问。然后，它通过将查询中的输入绑定到全局变量input来评估该输入。点运算符用于此变量来访问变量的值。如果规则中的表达式也为真，Rego 规则将allow返回 true。Rego 规则验证输入method中的是否等于 GET。然后，它会验证列表中的第一个元素path是否存在，然后getSalary再将列表中的第二个元素赋给该变量user。最后，它/getSalary/bob通过检查发出请求的人是否与user变量匹配来检查正在访问的路径。user input.user该规则allow应用 if-then 逻辑来返回布尔值，如输出所示：

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

使用外部数据的部分规则

要演示其他 OPA 功能，您可以对正在实施的访问规则添加要求。假设您要在上图的上下文中强制执行此访问控制要求：

员工可以阅读任何向他们汇报的人的工资。

在此示例中，OPA 可以访问外部数据，这些数据可以导入以帮助做出访问决策：

```
"managers": {
```

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

您可以通过在 OPA 中创建部分规则来生成任意 JSON 响应，该规则返回一组值而不是固定响应。这是部分规则的示例：

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

此规则返回一组报告到值的所有用户`input.user`，在本例中为`bob`。规则中的`[_]`构造用于迭代集合的值。这是规则的输出：

```
{
    "direct_report": [
      "dave",
      "john"
    ]
}
```

检索此信息可以帮助确定用户是否是经理的直接下属。对于某些应用程序，返回动态 JSON 比返回简单的布尔响应更可取。

组合起来

最后一个访问要求比前两个要求更复杂，因为它结合了两个要求中指定的条件：

员工可以阅读自己的工资和向他们汇报的任何人的工资。

要满足此要求，您可以使用此 Rego 政策：

```
default allow = false

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

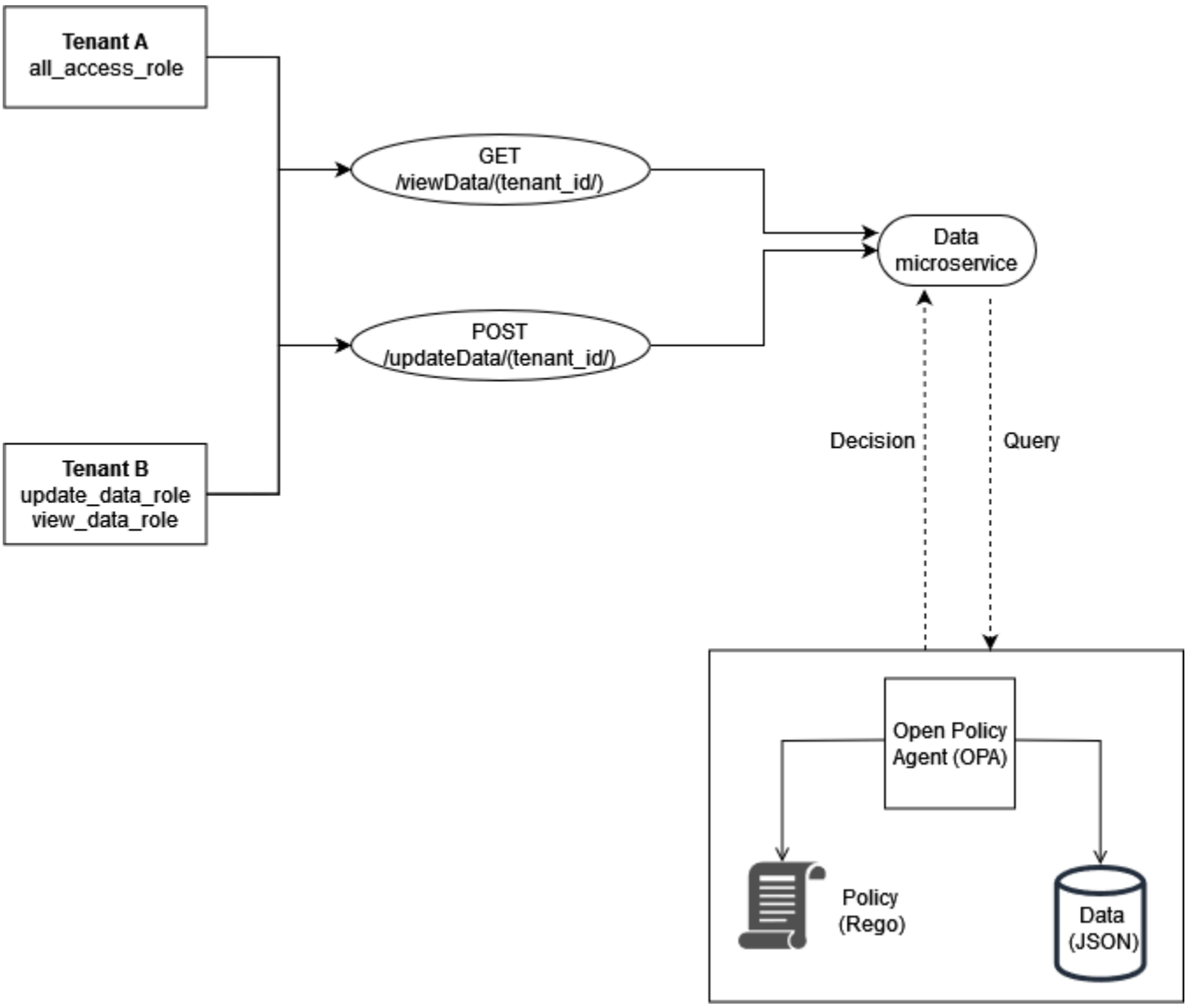
```
allow = true {
    input.method == "GET"
    input.path = ["getSalary", user]
    managers := data.managers[input.user][_]
    contains(managers, user)
}
```

如前所述，该政策的第一条规则允许任何试图查看自己的工资信息的用户进行访问。有两个同名的规则`allow`，在 Rego 中用作逻辑或运算符。第二条规则检索与之关联的所有直接下属的列表`input.user`（来自上图中的数据），并将此列表分配给变量。`managers`最后，该规则`input.user`通过验证其姓名是否包含在`managers`变量中，来检查试图查看其工资的用户是否是其直接下属。

本节中的示例非常基本，并未提供对 Rego 和 OPA 功能的完整或彻底探索。有关更多信息，请查看 OPA 文档，查看 OPA 自述文件，然后在 GitHub Rego P layground 中进行实验。

示例 2：使用 OPA 和 Rego 进行多租户访问控制和用户定义的 RBAC

此示例使用 OPA 和 Rego 来演示如何通过租户用户定义的自定义角色在多租户应用程序的 API 上实现访问控制。它还演示了如何根据租户限制访问权限。此模型显示 OPA 如何根据高级角色中提供的信息做出精细的权限决策。



租户的角色存储在用于为 OPA 做出访问决策的外部数据（RBAC 数据）中：

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

这些角色由租户用户定义时，应存储在外部数据源或身份提供者 (IdP) 中，在将租户定义的角色映射到权限和租户本身时，身份提供者可以充当真实来源。

此示例使用 OPA 中的两个策略来做出授权决策，并研究这些策略如何强制执行租户隔离。这些策略使用前面定义的 RBAC 数据。

```
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")
}
```

要显示此规则将如何运作，请考虑具有以下输入的 OPA 查询：

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

通过组合 RBAC 数据、OPA 策略和 OPA 查询输入，可以按如下方式做出此 API 调用的授权决定：

1. 来自的用户向Tenant A发出 API 调用/viewData/tenant_a。
2. 数据微服务接收调用并查询allowViewData规则，传递 OPA 查询输入示例中显示的输入。
3. OPA 使用 OPA 策略中的查询规则来评估所提供的输入。OPA 还使用 RBAC 数据中的数据来评估输入。OPA 会执行以下操作：
   a. 验证用于进行 API 调用的方法是否为GET。
   b. 验证请求的路径是否为。viewData
   c. 检查路径tenant_id中的是否等于与用户input.tenant_id关联的路径。这样可以确保保持租户隔离。另一个租户，即使角色相同，也无法获得进行此 API 调用的授权。
   d. 从角色的外部数据中提取角色权限列表并将其分配给变量。role_permissions此列表是通过使用与用户关联的租户定义角色来检索的 input.role.
   e. role_permissions检查它是否包含权限 viewData.
4. OPA 将以下决定返回给数据微服务：

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

此过程显示了 RBAC 和租户意识如何有助于通过 OPA 做出授权决定。为了进一步说明这一点，可以考虑使用以下查询输入/viewData/tenant_b进行 API 调用：

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

此规则将返回与 OPA 查询输入相同的输出，尽管它是针对具有不同角色的不同租户的。这是因为此调用是针对的，/tenant_b而 RBAC view_data_role 中的数据仍具有与之关联的viewData权限。要对强制执行相同类型的访问控制/updateData，可以使用类似的 OPA 规则：

```
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")
}
```

该规则在功能上与allowViewData规则相同，但它验证的是不同的路径和输入法。该规则仍可确保租户隔离，并检查租户定义的角色是否向 API 调用者授予权限。要了解如何强制执行此操作，请检查以下 API 调用的查询输入/updateData/tenant_b：

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

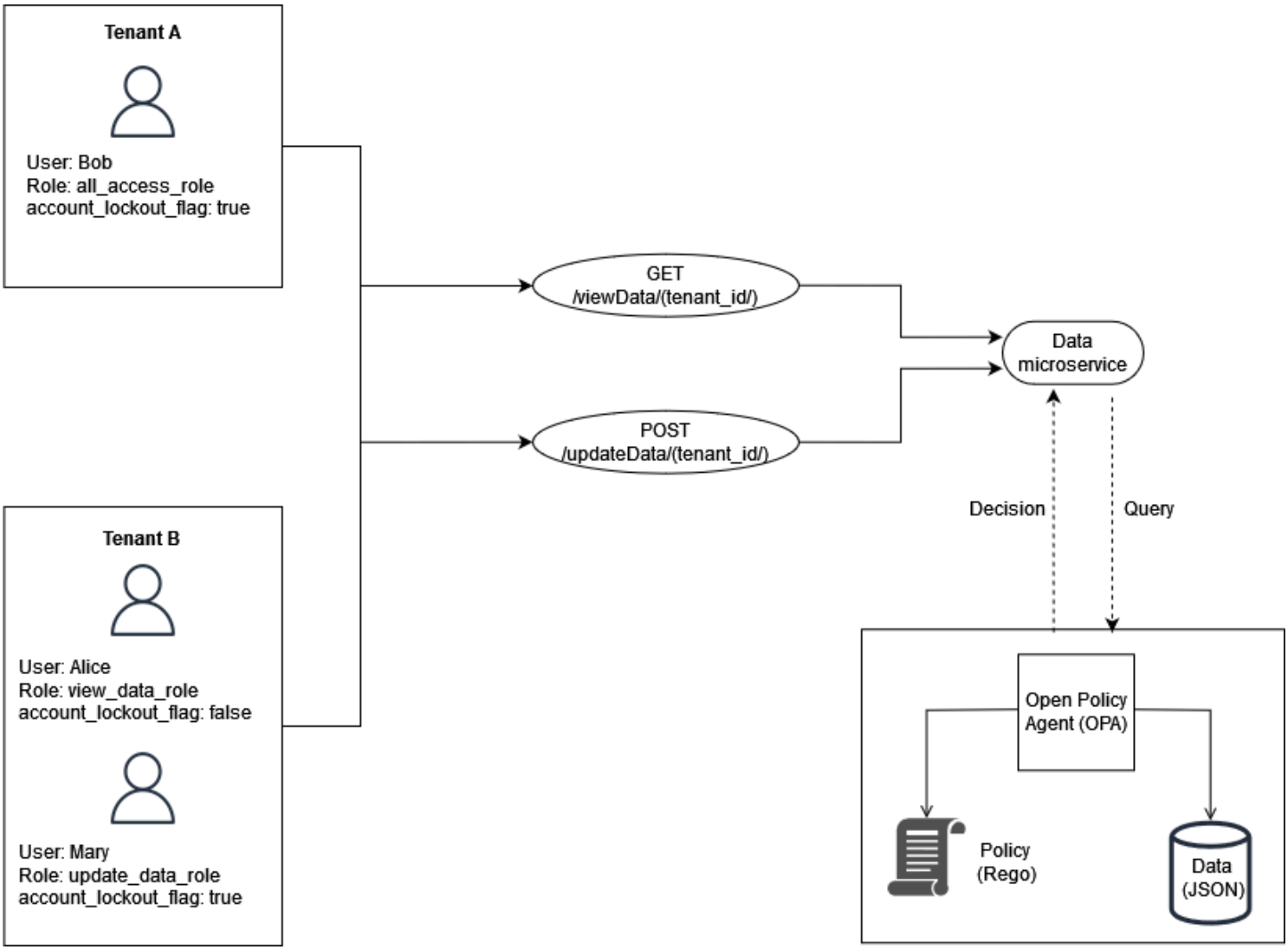
使用allowUpdateData规则评估此查询输入时，会返回以下授权决定：

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

此呼叫将不会获得授权。尽管 API 调用者关联了正确的，tenant_id并且正在使用经批准的方法调用 API，但还是租户view_data_role定义input.role的。view_data_role没有updateData权限；因此，对的调用/updateData是未经授权的。对于拥有. 的tenant_b用户来说，此调用本来是成功的update_data_role。

示例 3：使用 OPA 和 Rego 对 RBAC 和 ABAC 进行多租户访问控制

要增强上一节中的 RBAC 示例，您可以向用户添加属性。



此示例包含与上一个示例相同的角色，但添加了用户属性account_lockout_flag。这是用户特定的属性，与任何特定角色均无关联。您可以使用之前在本示例中使用的 RBAC 外部数据：

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

可以将account_lockout_flag用户属性作为用户 Bob 的 OPA 查询输入的一部分传递给数据服务：/viewData/tenant_a

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

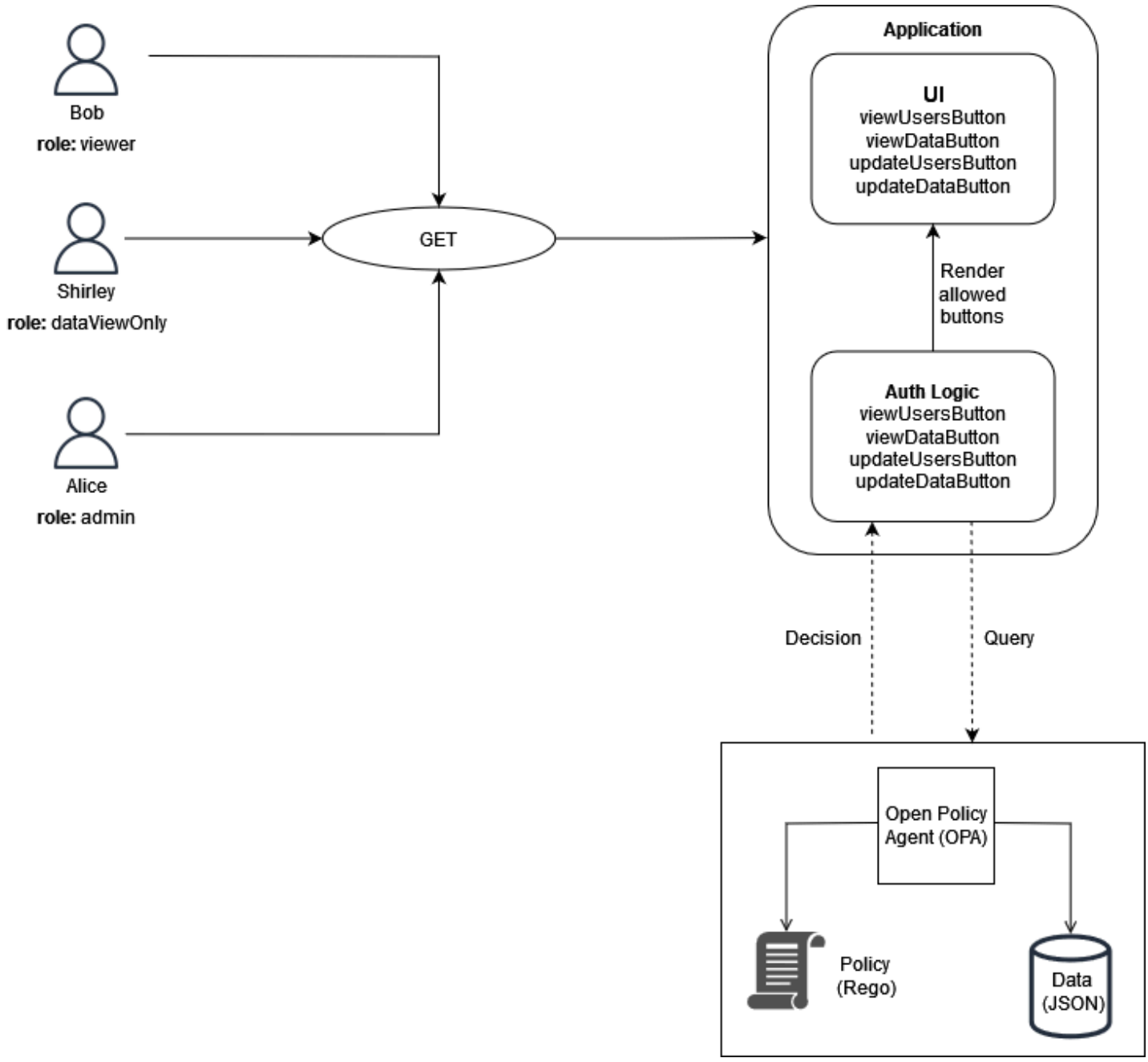
查询访问决策的规则与前面的示例类似，但包括另外一行用于检查该account_lockout_flag属性：

```
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"
}
```

此查询返回的授权决定为false。这是因为true适用于 Bob，而 Rego 规则allowViewData拒绝访问，尽管 Bob 的角色和租户account_lockout_flag attribute是正确的。

示例 4：使用 OPA 和 Rego 进行用户界面筛选

OPA 和 Rego 的灵活性支持筛选用户界面元素的能力。以下示例演示了 OPA 部分规则如何通过授权决定哪些元素应显示在 RBAC 的 UI 中。此方法是使用 OPA 筛选用户界面元素的众多不同方法之一。



在此示例中，单页 Web 应用程序有四个按钮。假设你想过滤 Bob、Shirley 和 Alice 的用户界面，这样他们只能看到与其角色相对应的按钮。当用户界面收到用户的请求时，它会查询 OPA 部分规则以确定应在界面中显示哪些按钮。当 Bob（具有角色`viewer`）向 UI 发出请求时，查询会将以下内容作为输入传递给 OPA：

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

OPA 使用为 RBAC 结构化的外部数据来做出访问决策：

```
{
    "roles": {
        "viewer": ["viewUsers", "viewData"],
        "dataViewOnly": ["viewData"],
        "admin": ["viewUsers", "viewData", "updateUsers", "updateData"]
    }
}
```

OPA 部分规则使用外部数据和输入来生成允许的操作列表：

```
user_permissions[permissions] {
    permissions := data.roles[input.role][_]
}
```

在部分规则中，OPA 使用作为查询一部分的`input.role`指定来确定应显示哪些按钮。Bob 拥有该角色`viewer`，外部数据指定查看者有两个权限：`viewUsers`和`viewData`。因此，此规则对 Bob（以及任何其他具有查看者角色的用户）的输出如下所示：

```
{
    "user_permissions": [
        "viewData",
        "viewUsers"
    ]
}
```

`dataViewOnly`扮演角色的 Shirley 的输出将包含一个权限按钮：`viewData`。拥有该`admin`角色的 Alice 的输出将包含所有这些权限。查询 OPA 时，这些响应会返回到用户界面。`user_permissions`然后，应用程序可以使用此响应来隐藏或显示`viewUsersButton``viewDataButton`、`updateUsersButton`、和`updateDataButton`。

使用自定义策略引擎

实现 PDP 的另一种方法是创建自定义策略引擎。此策略引擎的目标是将授权逻辑与应用程序分离。自定义策略引擎负责做出授权决策，类似于已验证权限或 OPA，以实现策略解钩。此解决方案与使用已

验证权限或 OPA 之间的主要区别在于，编写和评估策略的逻辑是为自定义策略引擎定制的。与引擎的任何交互都必须通过 API 或其他方法公开，以便授权决策能够到达应用程序。您可以使用任何编程语言编写自定义策略引擎，也可以使用其他机制进行策略评估，例如通用表达式语言 (CEL)。

实施 PEP

政策执行点 (PEP) 负责接收发送到政策决策点 (PDP) 进行评估的授权请求。PEP 可以是应用程序中必须保护数据和资源的任何地方，也可以是应用授权逻辑的地方。与 PDP 相比，PEP 相对简单。PEP 仅负责请求和评估授权决定，不需要任何授权逻辑。与 PDP 不同，PEP 不能集中在 SaaS 应用程序中。这是因为需要在整个应用程序及其接入点中实施授权和访问控制。PEP 可以应用于 API、微服务、前端后端 (BFF) 层或应用程序中需要或需要访问控制的任何点。在应用程序中普遍使用 PEP 可以确保在多个点经常独立地验证授权。

要实施 PEP，第一步是确定应在应用程序中何处实施访问控制。在决定应将 PEP 集成到应用程序中的位置时，请考虑以下原则：

如果应用程序公开了 API，则应对该API进行授权和访问控制。

这是因为在面向微服务或面向服务的架构中，API 充当不同应用程序功能之间的分隔符。将访问控制作为应用程序功能之间的逻辑检查点包含在内是有意义的。我们强烈建议您将 PEP 作为访问 SaaS 应用程序中每个 API 的先决条件。也可以在应用程序的其他位置集成授权。在单片应用程序中，可能需要将 PEP 集成到应用程序本身的逻辑中。没有一个地方应该包含 PEP，但可以考虑使用 API 原则作为起点。

请求授权决定

PEP 必须向 PDP 申请授权决定。请求可以采取多种形式。请求授权决策的最简单、最便捷的方法是向 PDP 公开的 RESTful API（OPA 或已验证权限）发送授权请求或查询。如果您使用的是经过验证的权限，也可以使用 AWS SDK 调用该IsAuthorized方法来检索授权决定。在这种模式下，PEP 的唯一功能是转发授权请求或查询所需的信息。这可以像将 API 收到的请求作为输入转发给 PDP 一样简单。还有其他创建 PEP 的方法。例如，您可以在本地将 OPA PDP 与以 Go 编程语言编写的应用程序作为库集成，而不必使用 API。

评估授权决定

PEP 需要包含评估授权决策结果的逻辑。当 PDP 作为 API 公开时，授权决策可能采用 JSON 格式并由 API 调用返回。PEP 必须评估此 JSON 代码，以确定所采取的操作是否获得授权。例如，如果 PDP 旨在提供布尔允许或拒绝授权决定，则 PEP 可能只需检查此值，然后返回 HTTP 状态码 200 表示允许，返回 HTTP 状态码 403 表示拒绝。这种将 PEP 作为访问 API 的先决条件的模式是一种易于实施且非常有效的模式，可以在 SaaS 应用程序中实现访问控制。在更复杂的场景中，PEP 可能负责评估 PDP 返回的任意 JSON 代码。撰写 PEP 时必须包含解释 PDP 返回的授权决策所必需的任何逻辑。

由于 PEP 可能在应用程序的许多不同位置实现，因此我们建议您将您的 PEP 代码打包为所选编程语言的可重用库或工件。这样，您的 PEP 就可以轻松地集成到应用程序中的任何位置，而只需极少的返工。

多租户 SaaS 架构的设计模型

实现 API 访问控制和授权的方法有很多。本指南重点介绍三种适用于多租户 SaaS 架构的设计模型。这些设计可作为实施政策决策点 (PDP) 和政策执行点 (PEP) 的高级参考，从而形成一个有凝聚力且无处不在的应用程序授权模型。

设计模型：

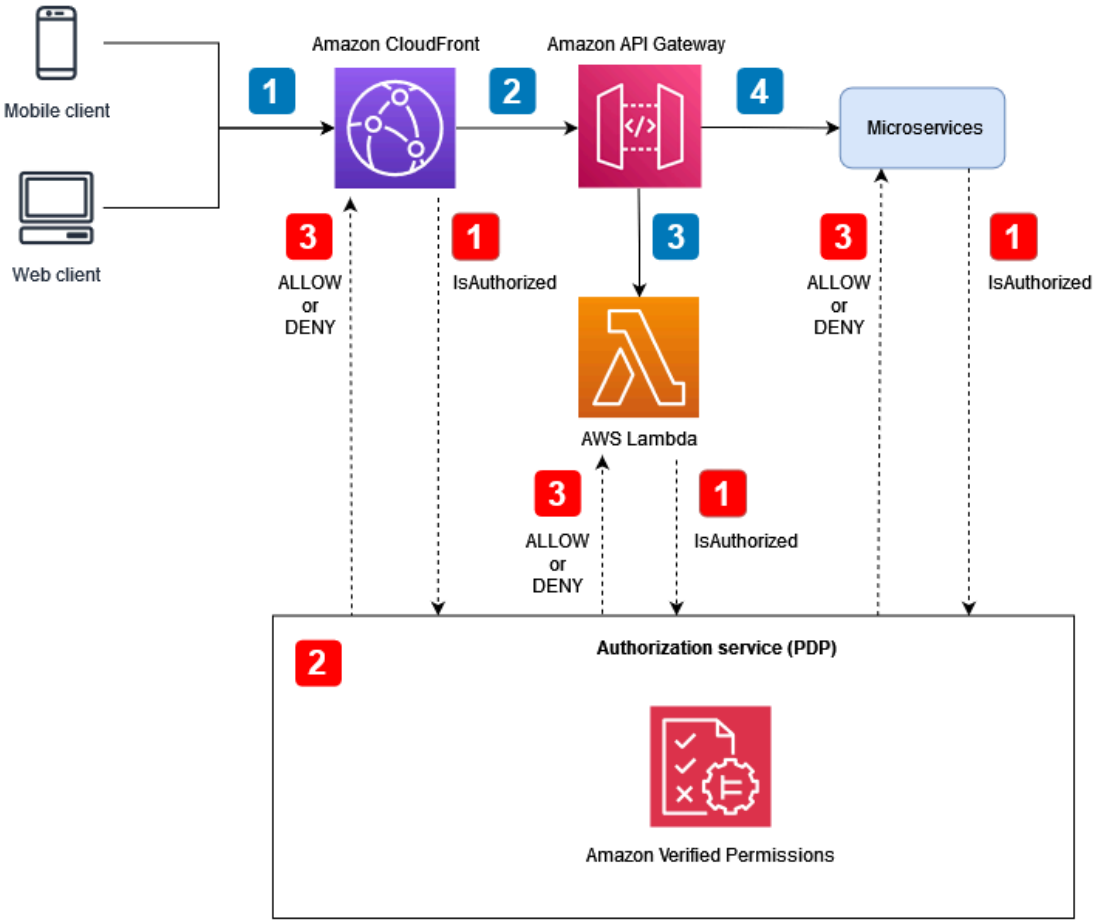
- 为 Amazon 验证权限设计模型
- OPA 的设计模型

为 Amazon 验证权限设计模型

在 API 上使用带有 PEP 的集中式 PDP

在 API 模型上带有策略执行点 (PEP) 的集中式政策决策点 (PDP) 遵循行业最佳实践，为 API 访问控制和授权创建有效且易于维护的系统。这种方法支持几个关键原则：

- 授权和 API 访问控制应用于应用程序中的多个点。
- 授权逻辑独立于应用程序。
- 访问控制决策是集中的。



应用程序流程（图中用蓝色编号标注说明）：

1. 拥有 JSON 网络令牌 (JWT) 的经过身份验证的用户会向亚马逊 CloudFront生成一个 HTTP 请求。
2. CloudFront 将请求转发到配置为 CloudFront来源的 Amazon API Gateway。
3. 调用 API Gateway 自定义授权器来验证 JWT。
4. 微服务会响应请求。

授权和 API 访问控制流程（图中用红色编号标注说明）：

1. PEP 调用授权服务并传递请求数据，包括任何 JWT。
2. 授权服务 (PDP)（在本例中为已验证权限）使用请求数据作为查询输入，并根据查询指定的相关策略对其进行评估。
3. 授权决定将返回给 PEP 并进行评估。

此模型使用集中式 PDP 来做出授权决策。PEP 是在不同时刻实施的，目的是向 PDP 提出授权请求。下图显示了如何在假设的多租户 SaaS 应用程序中实现此模型。

在此架构中，PEP 在服务终端节点上为亚马逊 CloudFront 和 Amazon API Gateway 以及每项微服务请求授权决策。授权决定由授权服务 Amazon 验证权限 (PDP) 做出。由于已验证权限是一项完全托管的服务，因此您不必管理底层基础架构。您可以使用 RESTful API 或 AWS SDK 与已验证的权限进行交互。

您也可以将此架构与自定义策略引擎一起使用。但是，从已验证权限中获得的任何优势都必须替换为自定义策略引擎提供的逻辑。

在 API 上使用 PEP 的集中式 PDP 为为 API 创建强大的授权系统提供了一个简单的选择。这简化了授权流程，还提供了一个可重复的接口 easy-to-use，用于为 API、微服务、前端后端 (BFF) 层或其他应用程序组件做出授权决策。

使用 Cedar SDK

Amazon Verified Permissions 使用 Cedar 语言来管理您的自定义应用程序中的精细权限。借助 Verified Permissions，您可以将 Cedar 策略存储在中心位置，利用毫秒级处理的低延迟，并审核不同应用程序的权限。您也可以选择将 Cedar SDK 直接集成到您的应用程序中，以便在不使用已验证权限的情况下提供授权决策。此选项需要额外的自定义应用程序开发来管理和存储您的用例的策略。但是，它可能是一个可行的替代方案，尤其是在由于互联网连接不一致而间歇性或无法访问已验证权限的情况下。

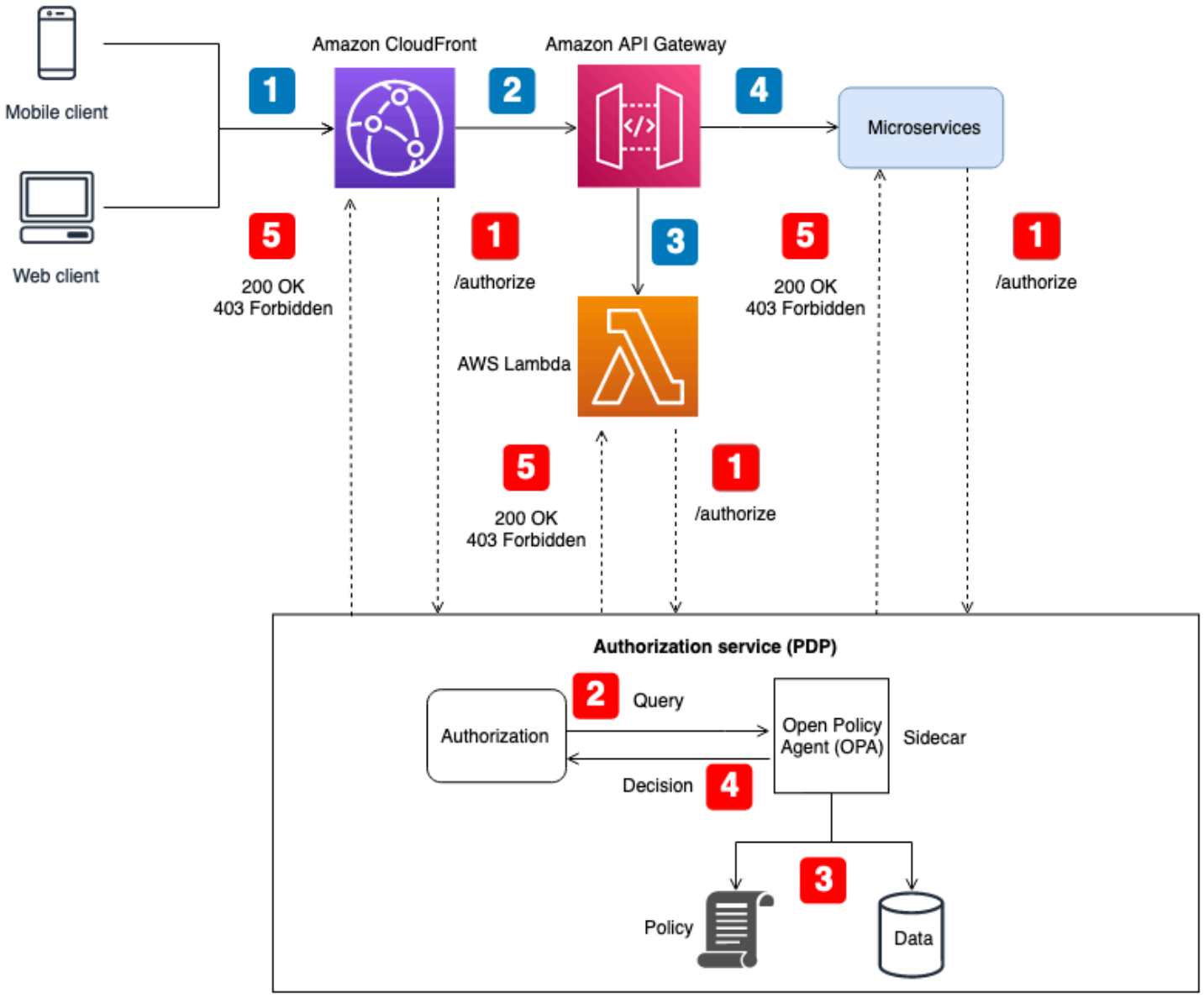
OPA 的设计模型

在 API 上使用带有 PEP 的集中式 PDP

在 API 模型上带有策略执行点 (PEP) 的集中式政策决策点 (PDP) 遵循行业最佳实践，为 API 访问控制和授权创建有效且易于维护的系统。这种方法支持几个关键原则：

- 授权和 API 访问控制应用于应用程序中的多个点。
- 授权逻辑独立于应用程序。
- 访问控制决策是集中的。

此模型使用集中式 PDP 来做出授权决策。所有 API 都实施了 PEP，用于向 PDP 提出授权请求。下图显示了如何在假设的多租户 SaaS 应用程序中实现此模型。



应用程序流程（图中用蓝色编号标注说明）：

1. 使用 JWT 的经过身份验证的用户会向 Amazon CloudFront 生成一个 HTTP 请求。
2. CloudFront 将请求转发到配置为 CloudFront来源的 Amazon API Gateway。
3. 调用 API Gateway 自定义授权器来验证 JWT。
4. 微服务会响应请求。

授权和 API 访问控制流程（图中用红色编号标注说明）：

1. PEP 调用授权服务并传递请求数据，包括任何 JWT。

2. 授权服务 (PDP) 获取请求数据并查询作为边车运行的 OPA 代理 REST API。请求数据用作查询的输入。
3. OPA 根据查询中指定的相关策略评估输入。如有必要，可以导入数据以做出授权决定。
4. OPA 将决策返回给授权服务。
5. 授权决定将返回给 PEP 并进行评估。

在此架构中，PEP 在服务终端节点上为亚马逊 CloudFront 和 Amazon API Gateway 以及每项微服务请求授权决策。授权决定由带有OPA边车的授权服务（PDP）做出。您可以将此授权服务作为容器或传统服务器实例进行操作。OPA 边车在本地公开其 RESTful API，因此只有授权服务才能访问该 API。授权服务公开了一个可供PEP使用的单独API。让授权服务充当 PEP 和 OPA 之间的中介，允许在 PEP 和 OPA 之间插入任何可能需要的转换逻辑，例如，当 PEP 的授权请求不符合 OPA 预期的查询输入时。

您也可以将此架构与自定义策略引擎一起使用。但是，从 OPA 获得的任何优势都必须替换为自定义策略引擎提供的逻辑。

在 API 上使用 PEP 的集中式 PDP 为为 API 创建强大的授权系统提供了一个简单的选择。它易于实现，还提供了一个可重复的接口 easy-to-use，用于为 API、微服务、前端后端 (BFF) 层或其他应用程序组件做出授权决策。但是，这种方法可能会在您的应用程序中造成过多的延迟，因为授权决策需要调用单独的 API。如果存在网络延迟问题，则可以考虑使用分布式 PDP。

在 API 上使用带有 PEP 的分布式 PDP

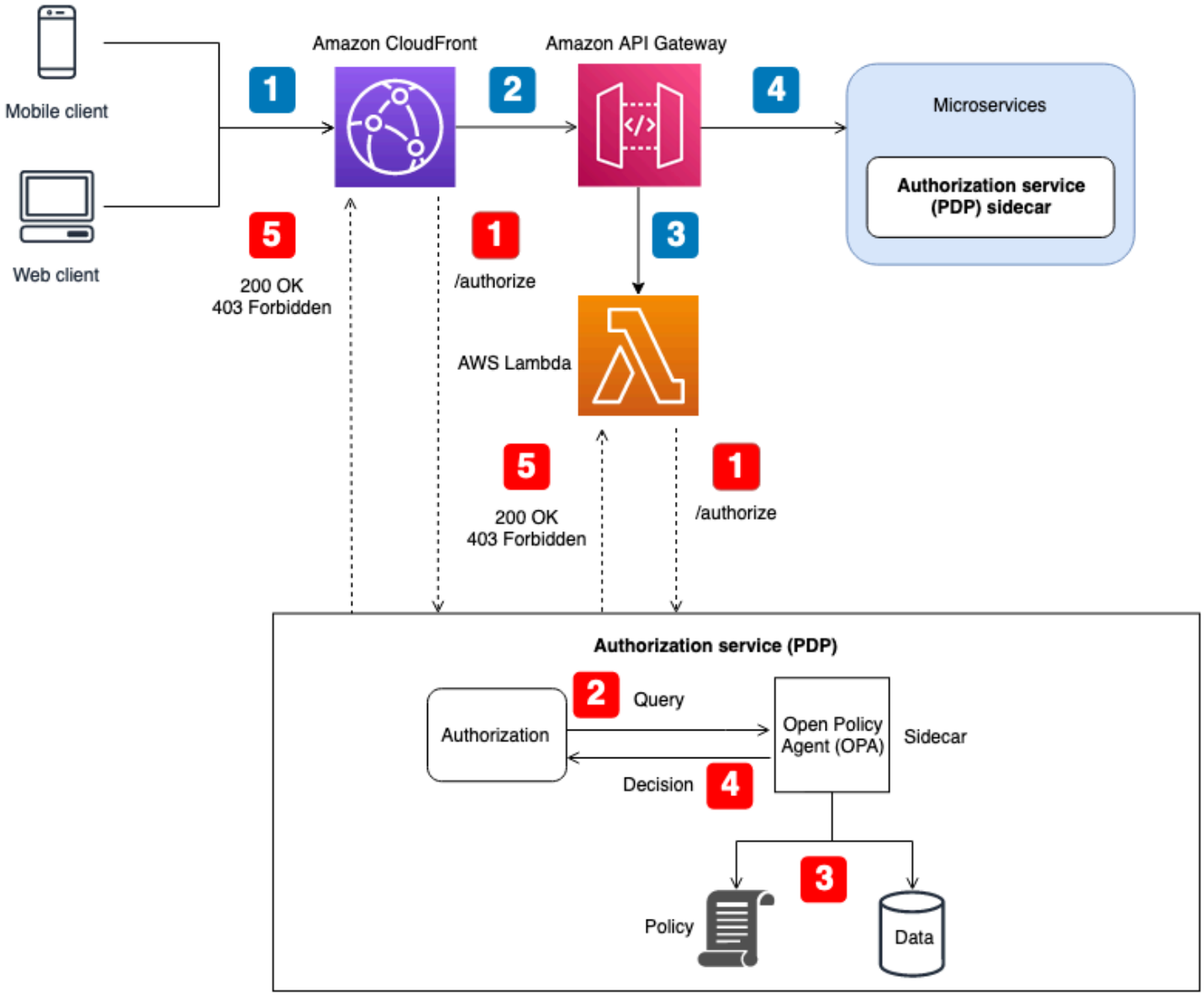
在 API 模型上带有策略执行点 (PEP) 的分布式策略决策点 (PDP) 遵循行业最佳实践，以创建有效的 API 访问控制和授权系统。与在 API 上使用 PEP 的集中式 PDP 模型一样，这种方法支持以下关键原则：

- 授权和 API 访问控制应用于应用程序中的多个点。
- 授权逻辑独立于应用程序。
- 访问控制决策是集中的。

您可能想知道，为什么在分发 PDP 时，访问控制决策是集中的。尽管 PDP 可能存在于应用程序的多个位置，但它必须使用相同的授权逻辑来做出访问控制决策。在输入相同的情况下，所有 PDP 都提供相同的访问控制决策。所有 API 都实施了 PEP，用于向 PDP 提出授权请求。下图显示了如何在假设的多租户 SaaS 应用程序中实现这种分布式模型。

在这种方法中，PDP 是在应用程序的多个位置实现的。对于具有可运行 OPA 并支持 PDP 的板载计算功能的应用程序组件，例如带边车的容器化服务或亚马逊弹性计算云 (Amazon EC2) 实例，PDP 决策可以直接集成到应用程序组件中，而不必对集中式 PDP 服务进行 RESTful API 调用。这样做的好处是可以减少集中式 PDP 模型中可能遇到的延迟，因为并非每个应用程序组件都必须进行额外的 API 调用才能获得授权决策。但是，对于不具备直接集成 PDP 的板载计算功能的应用程序组件（例如 Amazon 或 Amazon API CloudFront Gateway 服务），在此模型中仍然需要集中式 PDP。

下图显示了如何在假设的多租户 SaaS 应用程序中实现集中式 PDP 和分布式 PDP 的组合。



应用程序流程（图中用蓝色编号标注说明）：

1. 使用 JWT 的经过身份验证的用户会向 Amazon CloudFront 生成一个 HTTP 请求。

2. CloudFront 将请求转发到配置为 CloudFront来源的 Amazon API Gateway。
3. 调用 API Gateway 自定义授权器来验证 JWT。
4. 微服务会响应请求。

授权和 API 访问控制流程（图中用红色编号标注说明）：

1. PEP 调用授权服务并传递请求数据，包括任何 JWT。
2. 授权服务 (PDP) 获取请求数据并查询作为边车运行的 OPA 代理 REST API。请求数据用作查询的输入。
3. OPA 根据查询中指定的相关策略评估输入。如有必要，可以导入数据以做出授权决定。
4. OPA 将决策返回给授权服务。
5. 授权决定将返回给 PEP 并进行评估。

在此架构中，PEP 请求在服务端点 CloudFront 和 API Gateway 以及每项微服务的授权决策。微服务的授权决策由授权服务（PDP）做出，该服务与应用程序组件一起充当边车。这种模式适用于在容器或亚马逊弹性计算云 (Amazon EC2) 实例上运行的微服务（或服务）。API Gateway 等服务的授权决策仍 CloudFront 需要联系外部授权服务。无论如何，授权服务都会公开一个可供PEP使用的 API。让授权服务充当 PEP 和 OPA 之间的中介，允许在 PEP 和 OPA 之间插入任何可能需要的转换逻辑，例如，当 PEP 的授权请求不符合 OPA 预期的查询输入时。

您也可以将此架构与自定义策略引擎一起使用。但是，从 OPA 获得的任何优势都必须替换为自定义策略引擎提供的逻辑。

在 API 上包含 PEP 的分布式 PDP 提供了为 API 创建强大的授权系统的选项。它实现起来很简单，并提供了一个可重复的接口 easy-to-use，用于为 API、微服务、前端后端 (BFF) 层或其他应用程序组件做出授权决策。这种方法还具有减少集中式 PDP 模型中可能遇到的延迟的优点。

使用分布式 PDP 作为库

您也可以向以库或包形式提供的 PDP 请求授权决定，以便在应用程序中使用。OPA 可用作 Go 第三方库。对于其他编程语言，采用此模型通常意味着必须创建自定义策略引擎。

Amazon 已验证权限多租户设计注意事项

在多租户 SaaS 解决方案中使用 Amazon 验证权限来实施授权时，需要考虑多种设计选项。在探索这些选项之前，让我们澄清一下多租户 SaaS 环境中隔离和授权之间的区别。隔离租户可以防止入站和出站数据泄露给错误的租户。授权可确保用户拥有访问租户的权限。

在已验证的权限中，策略存储在策略存储中。如Verified Permissions 文档中所述，您可以通过为每个租户使用单独的策略存储来隔离租户的策略，也可以通过为所有租户使用单个策略存储来允许租户共享策略。本节讨论了这两种隔离策略的优缺点，并描述了如何使用分层部署模型来部署它们。有关其他背景信息，请参阅 “已验证权限” 文档。

尽管本节中讨论的标准侧重于已验证的权限，但一般概念植根于隔离思维方式及其提供的指导。SaaS 应用程序必须始终将租户隔离视为其设计的一部分，这种一般的隔离原则延伸到在 SaaS 应用程序中包括经过验证的权限。本节还引用了核心的SaaS隔离模型，例如孤立的SaaS模型和池化的SaaS 模型。有关更多信息，请参阅 Well-Architecte AWS d 框架 SaaS 镜头中的核心隔离概念。

设计多租户 SaaS 解决方案时的关键考虑因素是租户隔离和租户入职。租户隔离会影响安全、隐私、弹性和性能。租户入职会影响您的运营流程，因为它与运营开销和可观察性有关。经历SaaS旅程或实施多租户解决方案的组织必须始终优先考虑SaaS应用程序如何处理租赁。尽管 SaaS 解决方案可能倾向于特定的隔离模型，但不一定要求整个 SaaS 解决方案保持一致。例如，您为应用程序的前端组件选择的隔离模型可能与您为微服务或授权服务选择的隔离模型不同。

设计注意事项：

- 租户入职和用户租户注册
- 每租户策略存储
- 一个共享的多租户策略存储
- 分层部署模型

租户入职和用户租户注册

SaaS 应用程序遵守 SaaS 身份的概念，并遵循将用户身份绑定到租户身份的一般最佳实践。绑定涉及将租户标识符存储为身份提供者中用户的声明或属性。这就将向租户映射身份的责任从每个应用程序转移到用户注册流程。然后，每位经过身份验证的用户都将拥有正确的租户身份，作为 JSON Web 令牌 (JWT) 的一部分。

同样，为授权请求选择正确的策略存储不应由应用程序逻辑决定。要确定特定授权请求应使用哪个策略存储，请维护用户到策略存储或租户与策略存储的映射。这些映射通常保存在您的应用程序引用的数据

存储中，例如 Amazon DynamoDB 或亚马逊关系数据库服务 (Amazon RDS)。您也可以通过身份提供商 (IdP) 中的数据来提供或补充这些映射。然后，租户、用户和策略存储之间的关系通常通过包含授权请求所需的所有关系的 JWT 提供给用户。

此示例显示了属于租户`TenantA`并使用带有策略存储 ID 的策略存储进行授权的用户 `Alice` JWT `ps-43214321` 的显示方式。

```
{
   "sub":"1234567890",
   "name":"Alice",
   "tenant":"TenantA",
   "policyStoreId":"ps-43214321"
}
```

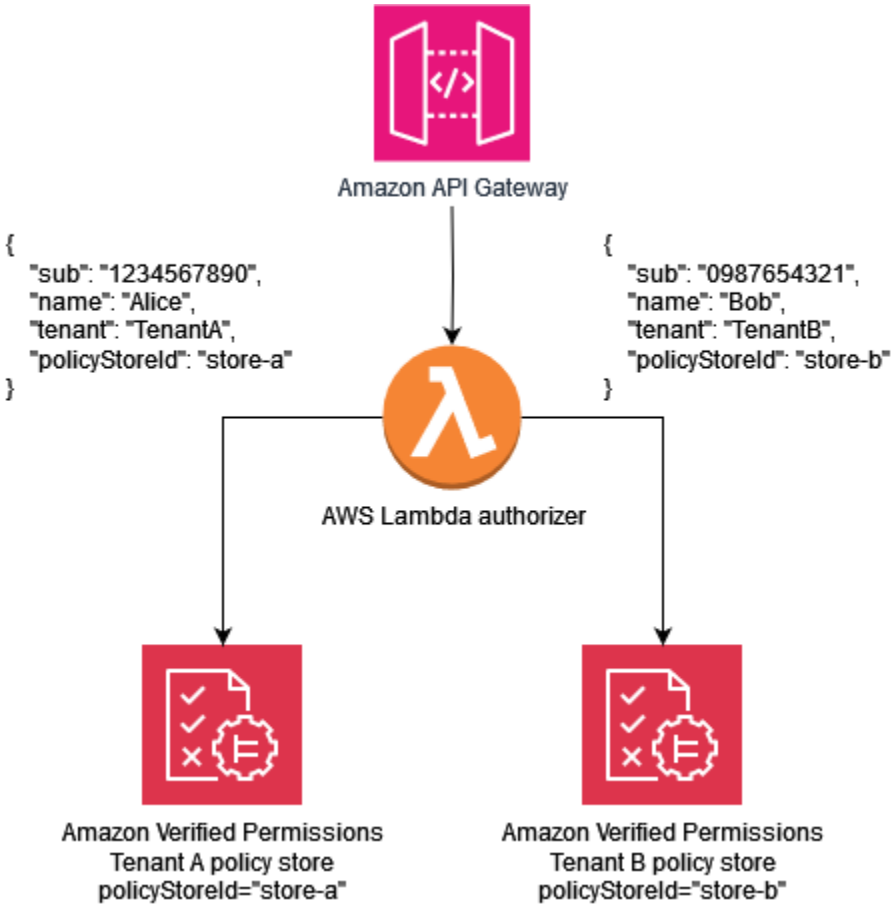
每租户策略存储

Amazon Verified Permissions 中的每租户策略存储设计模型将 SaaS 应用程序中的每个租户与其自己的策略存储区相关联。此模型类似于 SaaS 孤岛隔离模型。两种模式都要求创建租户特定的基础设施，并且具有相似的优缺点。这种方法的主要好处是基础设施强制的租户隔离，支持每个租户的独特授权模式，消除了邻居的噪音担忧，以及缩小了策略更新或部署失败的影响范围。这种方法的缺点包括更复杂的租户入职流程、部署和运营。如果解决方案对每个租户都有唯一的策略，则推荐使用按租户策略存储的方法。

如果您的 SaaS 应用程序需要，按租户策略存储模型可以提供一种高度孤立的租户隔离方法。您也可以将此模型与池隔离配合使用，但是您的 Verified Permissions 实现不会共享更广泛的池隔离模型的标准优势，例如简化的管理和操作。

在每租户策略存储中，租户隔离是通过在用户注册过程中将租户的策略存储标识符映射到用户的 SaaS 身份来实现的，如前所述。这种方法将租户的策略存储与用户主体紧密地联系在一起，并提供了一种在整个 SaaS 解决方案中共享映射的一致方式。您可以将 SaaS 应用程序作为 IdP 的一部分或外部数据源（例如 DynamoDB）进行维护，从而将其提供到 SaaS 应用程序的映射。这还可以确保委托人是租户的一部分，并且使用租户的策略存储。

此示例说明了如何将包含用户`policyStoreId`和`tenant`的 JWT 从 API 端点传递到授权方中的策略评估点， AWS Lambda 授权方将请求路由到正确的策略存储。



以下示例策略说明了每租户策略商店的设计范例。Alice属于的用户还TenantA.会policyStoreIdstore-a映射到的租户身份，Alice,并强制使用正确的策略存储。这样可以确保使用TenantA的策略。

> **Note**
>
> 每租户策略存储模型隔离了租户的策略。授权会强制执行允许用户对其数据执行的操作。任何使用此模型的假设应用程序所涉及的资源都应使用其他隔离机制进行隔离，如Well-Architected Fr amework，SaaS Lens AWS 文档中所定义。

在此策略中，Alice有权查看所有资源的数据。

```
permit (
    principal == MultiTenantApp::User::"Alice",
    action == MultiTenantApp::Action::"viewData",
    resource
);
```

要提出授权请求并使用已验证权限策略开始评估，您需要提供与映射到租户的唯一 ID 相对应的策略存储 ID `store-a`。

```
{
   "policyStoreId":"store-a",
   "principal":{
      "entityType":"MultiTenantApp::User",
      "entityId":"Alice"
   },
   "action":{
      "actionType":"MultiTenantApp::Action",
      "actionId":"viewData"
   },
   "resource":{
      "entityType":"MultiTenantApp::Data",
      "entityId":"my_example_data"
   },
   "entities":{
      "entityList":[
         [
            {
               "identifier":{
                  "entityType":"MultiTenantApp::User",
                  "entityId":"Alice"
               },
               "attributes":{},
               "parents":[]
            },
            {
               "identifier":{
                  "entityType":"MultiTenantApp::Data",
                  "entityId":"my_example_data"
               },
               "attributes":{},
               "parents":[]
            }
         ]
      ]
   }
}
```

该用户Bob属于租户 B， policyStoreId`store-b`并且还映射到的租户身份Bob，这会强制使用正确的策略存储。这样可以确保使用租户 B 的策略。

在此策略中，Bob有权自定义所有资源的数据。在此示例中，customizeData可能是仅针对租户 B 的操作，因此该策略对租户 B 来说是唯一的。每租户策略存储模型本质上支持基于每个租户的自定义策略。

```
permit (
    principal == MultiTenantApp::User::"Bob",
    action == MultiTenantApp::Action::"customizeData",
    resource
);
```

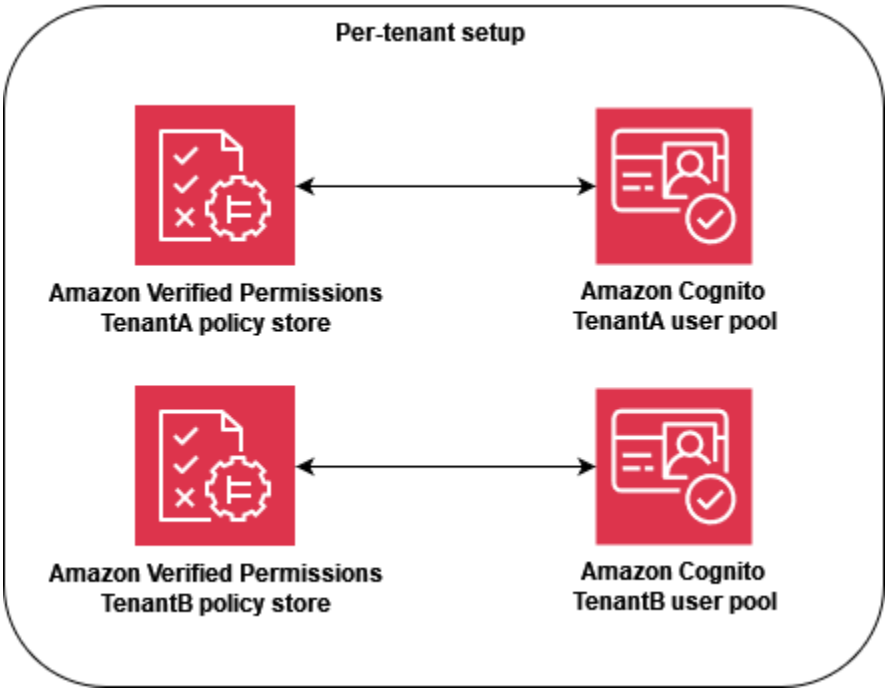
要提出授权请求并使用已验证权限策略开始评估，您需要提供与映射到租户的唯一 ID 相对应的策略存储 ID store-b。

```
{
   "policyStoreId":"store-b",
   "principal":{
      "entityType":"MultiTenantApp::User",
      "entityId":"Bob"
   },
   "action":{
      "actionType":"MultiTenantApp::Action",
      "actionId":"customizeData"
   },
   "resource":{
      "entityType":"MultiTenantApp::Data",
      "entityId":"my_example_data"
   },
   "entities":{
      "entityList":[
         [
            {
               "identifier":{
                  "entityType":"MultiTenantApp::User",
                  "entityId":"Bob"
               },
               "attributes":{},
               "parents":[]
            },
            {
               "identifier":{
                  "entityType":"MultiTenantApp::Data",
                  "entityId":"my_example_data"
```

```
                },
                "attributes":{},
                "parents":[]
            }
        ]
    ]
  }
}
```

使用已验证的权限，可以将 IdP 与策略存储集成，但不是必需的。这种集成允许策略将身份存储中的委托人明确引用为策略的主体。有关如何作为已验证权限的 IdP 与 Amazon Cognito 集成的更多信息，请参阅已验证权限文档和 Amazon Cognito 文档。

将策略存储与 IdP 集成时，每个策略存储只能使用一个身份源。例如，如果您选择将已验证权限与 Amazon Cognito 集成，则必须镜像用于隔离已验证权限策略存储和 Amazon Cognito 用户池的租户策略。策略存储库和用户池也必须位于同一位置 AWS 账户。



在操作层面上，每租户策略存储具有审计优势，因为您可以轻松地为每个租户AWS CloudTrail 单独查询已记录的活动。但是，我们仍然建议您将每个租户维度上的其他自定义指标记录到 Amazon CloudWatch。

按租户策略存储方法还需要密切关注两个已验证的权限配额，以确保它们不会干扰 SaaS 解决方案的运行。这些配额是每个账户每个区域的策略存储量和每个账户每个区域的每秒IsAuthorized 请求数。您可以申请提高两个配额。

有关如何实现按租户策略存储模型的更详细示例，请参阅 AWS 博客文章 SaaS 访问控制使用每租户策略存储的 Amazon Verified Permis sions。

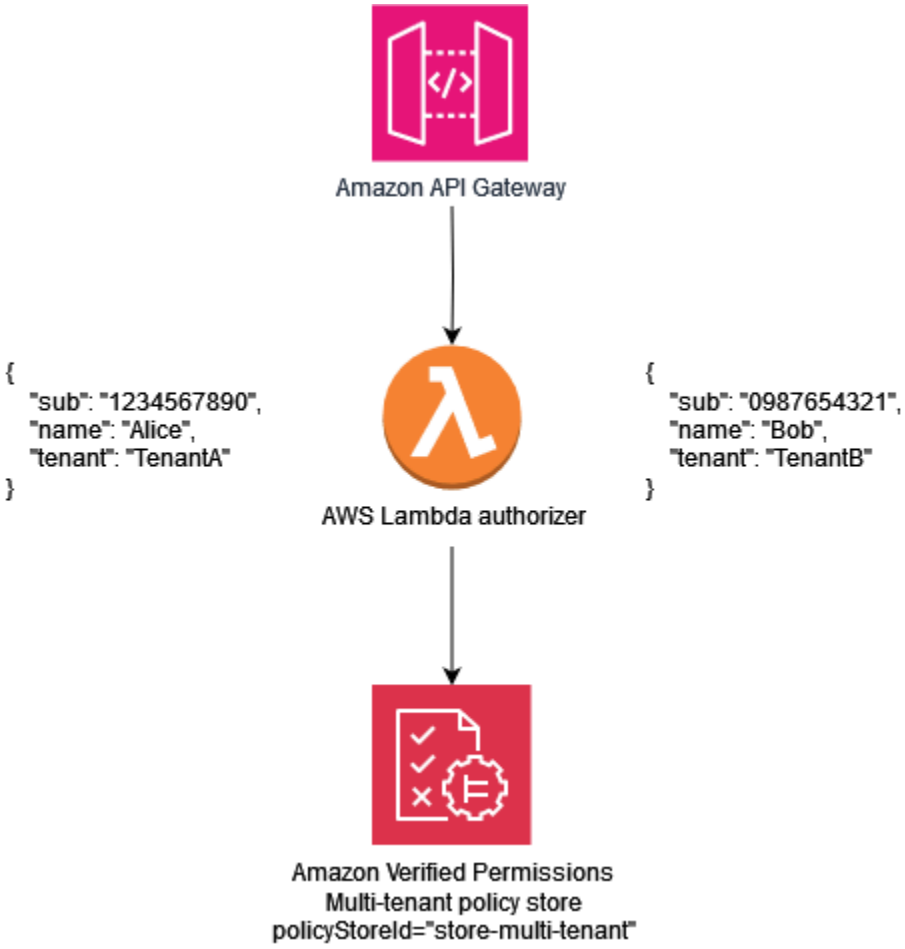
一个共享的多租户策略存储

一个共享的多租户策略存储设计模型在 Amazon 中为所有租户使用 SaaS 解决方案中所有租户的单个多租户策略存储经过验证的权限。这种方法的主要好处是简化了管理和操作，特别是因为在租户入职期间，您不必创建额外的策略存储库。这种方法的缺点包括策略更新或部署中的任何失败或错误所造成的影响范围扩大，以及更容易受到噪音邻居效应的影响。此外，如果您的解决方案要求每个租户都有独特的策略，我们不建议使用这种方法。在这种情况下，请改用每租户策略存储模型，以保证使用正确租户的策略。

一个共享的多租户策略存储方法类似于 SaaS 池化隔离模型。如果您的 SaaS 应用程序需要，它可以提供一种共享的租户隔离方法。如果您的 SaaS 解决方案对其微服务应用孤立隔离，则也可以使用此模型。选择模型时，应独立评估租户数据隔离要求和 SaaS 应用程序所需的已验证权限策略结构。

如前所述，为了在整个 SaaS 解决方案中采用一致的方式共享租户标识符，最好在用户注册期间将标识符映射到用户的 SaaS 身份。您可以将 SaaS 应用程序作为 IdP 的一部分或外部数据源（例如 DynamoDB）进行维护，从而将其提供给 SaaS 应用程序。我们还建议您将共享策略存储 ID 映射到用户。尽管 ID 不用作租户隔离的一部分，但这是一种很好的做法，因为它有助于将来的更改。

以下示例显示了 API 端点如何为属于不同租户但与策略存储 ID 共享策略存储以`store-multi-tenant`进行授权的用户`Alice`和`Bob`发送 JWT。由于所有租户共享一个策略存储，因此您无需在令牌或数据库中维护策略存储 ID。由于所有租户共享一个策略存储 ID，因此您可以将该 ID 作为环境变量提供，您的应用程序可以使用该变量来调用策略存储。



以下示例策略说明了一个共享的多租户策略设计范例。在此策略中`MultiTenantApp::User`，`MultiTenantApp::RoleAdmin`拥有父项的委托人有权查看所有资源的数据。

```
permit (
    principal in MultiTenantApp::Role::"Admin",
    action == MultiTenantApp::Action::"viewData",
    resource
);
```

由于使用的是单个策略存储，因此已验证权限策略存储必须确保与委托人关联的租赁属性与与资源关联的租赁属性相匹配。这可以通过在策略存储中加入以下策略来实现，以确保所有在资源和委托人上没有匹配租赁属性的授权请求都被拒绝。

```
forbid(
    principal,
    action,
```

```
    resource
)
unless {
    resource.Tenant == principal.Tenant
};
```

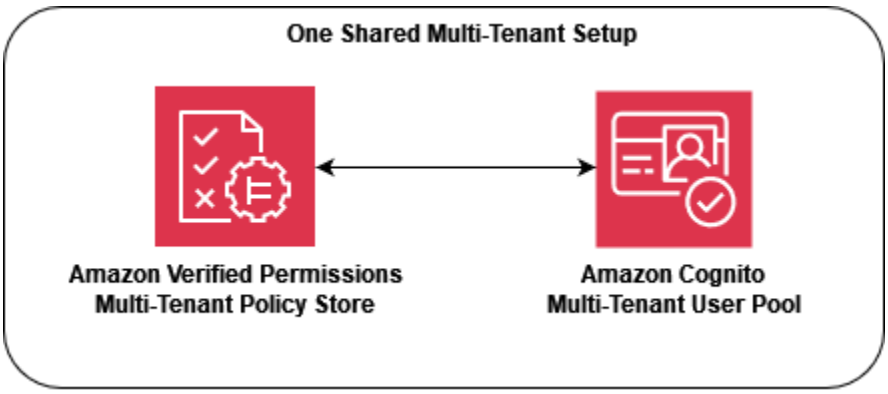
对于使用一个共享的多租户策略存储模型的授权请求，策略存储 ID 是共享策略存储的标识符。在以下请求中 UserAlice，允许访问是因为她有 ofAdmin，并且与资源和委托人关联的Tenant属性都是TenantA。Role

```
{
   "policyStoreId":"store-multi-tenant",
   "principal":{
      "entityType":"MultiTenantApp::User",
      "entityId":"Alice"
   },
   "action":{
      "actionType":"MultiTenantApp::Action",
      "actionId":"viewData"
   },
   "resource":{
      "entityType":"MultiTenantApp::Data",
      "entityId":"my_example_data"
   },
   "entities":{
      "entityList":[
         {
            "identifier":{
               "entityType":"MultiTenantApp::User",
               "entityId":"Alice"
            },
            "attributes": {
                {
                    "Tenant": {
                        "entityIdentifier": {
                            "entityType":"MultitenantApp::Tenant",
                            "entityId":"TenantA"
                        }
                    }
                }
            },
            "parents":[
               {
```

```
                "entityType":"MultiTenantApp::Role",
                "entityId":"Admin"
            }
         ]
      },
      {
         "identifier":{
            "entityType":"MultiTenantApp::Data",
            "entityId":"my_example_data"
         },
         "attributes": {
             {
                 "Tenant": {
                     "entityIdentifier": {
                         "entityType":"MultitenantApp::Tenant",
                         "entityId":"TenantA"
                     }
                 }
             }
         },
         "parents":[]
      }
   ]
}
}
```

使用已验证的权限，可以将 IdP 与策略存储集成，但不是必需的。这种集成允许策略将身份存储中的委托人明确引用为策略的主体。有关如何作为已验证权限的 IdP 与 Amazon Cognito 集成的更多信息，请参阅已验证权限文档和 Amazon Cognito 文档。

将策略存储与 IdP 集成时，每个策略存储只能使用一个身份源。例如，如果您选择将已验证权限与 Amazon Cognito 集成，则必须镜像用于隔离已验证权限策略存储和 Amazon Cognito 用户池的租户策略。策略存储库和用户池也必须位于同一位置 AWS 账户。



从运营和审计的角度来看，一个共享的多租户策略存储模式有一个缺点，因为记录的活动 AWS CloudTrail需要更多的复杂查询才能筛选出租户上的单个活动，因为每个记录的 CloudTrail 呼叫都使用相同的策略存储。在这种情况下，将每个租户维度上的其他自定义指标记录 CloudWatch 到 Amazon 会很有帮助，这样可以确保适当的可观察性和审计能力水平。

单一共享的多租户策略存储方法还需要密切关注已验证的权限配额，以确保它们不会干扰 SaaS 解决方案的运行。特别是，我们建议您监控每个区域每个账户的每秒IsAuthorized 请求量配额，以确保不超过其限制。您可以申请增加此配额。

分层部署模型

通过创建分层部署模型，您可以将高优先级 “企业级” 租户与可能更多的 “标准层” 客户隔离开来。在此模型中，您可以针对每个层级单独推出部署到策略存储库中的策略的任何更改，这样可以将每个层级的客户与其级别之外所做的更改隔离开来。在分层部署模型中，策略存储通常是作为每个层级的初始基础架构配置的一部分创建的，而不是在租户加入时进行部署。

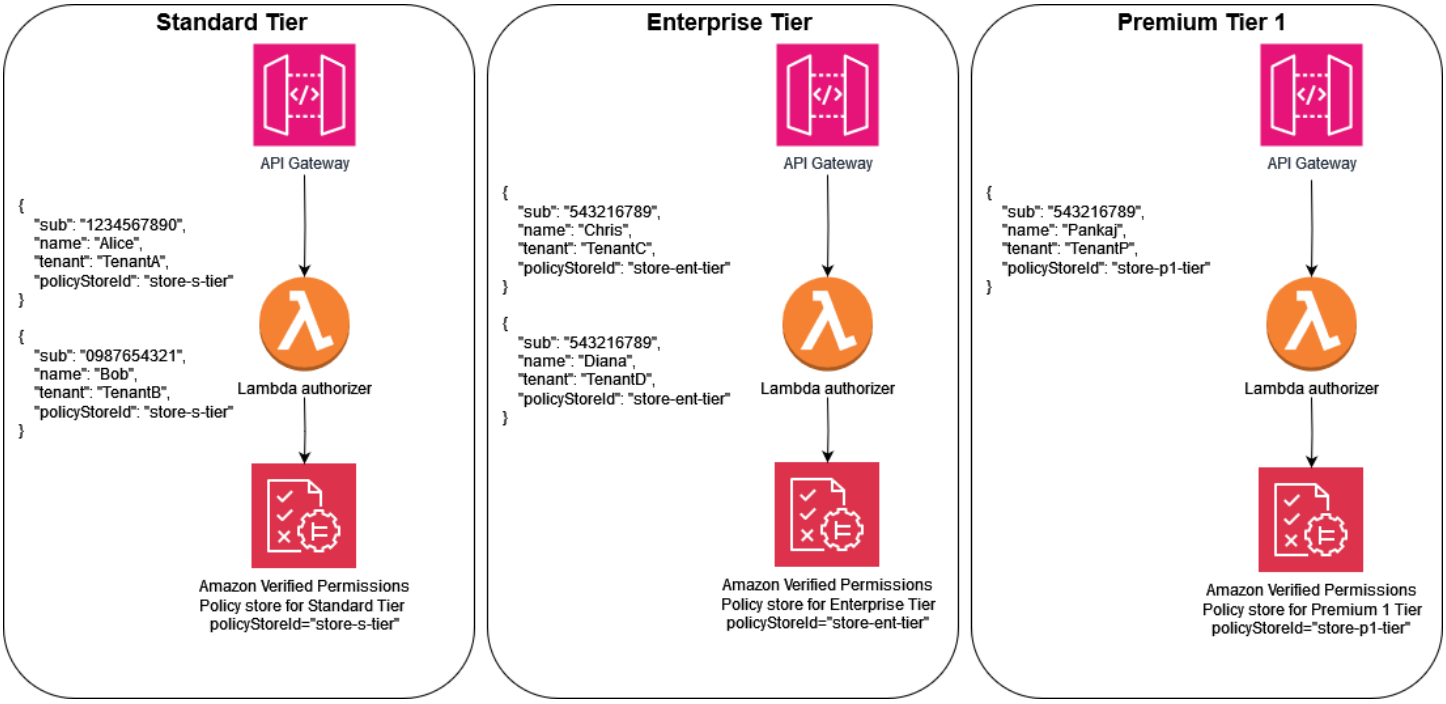
如果您的解决方案主要使用池化隔离模型，则可能需要额外的隔离或自定义。例如，您可以创建一个 “高级层”，其中每个租户都将获得自己的租户层基础架构，通过部署只有一个租户的池化实例来创建孤立的模型。这可以采取完全分离的 “高级租户A” 和 “高级租户B” 基础架构的形式，包括保单存储。这种方法为最高级别的客户提供了孤立的隔离模型。

在分层部署模型中，每个策略存储都应遵循相同的隔离模型，尽管它是分开部署的。由于正在使用多个策略存储，因此您需要在整个 SaaS 解决方案中强制采用一致的方式共享与租户关联的策略存储标识符。与每租户策略存储模型一样，最好在用户注册期间将租户标识符映射到用户的 SaaS 身份。

下图显示了三个层：Standard TierEnterprise Tier、和Premium Tier 1。每个层都单独部署在自己的基础架构中，并在该层内使用一个共享策略存储。标准和企业层包含多个租户。TenantATenantB并且在企业层中Standard Tier，TenantCTenantD并且在企业层中。

Premium Tier 1仅包含TenantP，因此您可以像解决方案具有完全孤立的隔离模型一样为高级租户提供服务，并提供自定义策略等功能。为新的高级级别客户提供入职将导致Premium Tier 2基础设施的创建。

> **Note**
>
> 高级层中的应用程序、部署和租户注册与标准层和企业层相同。唯一的区别是，高级层级入职工作流程始于配置新的层级基础架构。



OPA 多租户设计注意事项

Open Policy Agent (OPA) 是一项灵活的服务，可以应用于需要应用程序做出策略和授权决策的许多用例。在多租户 SaaS 应用程序中使用 OPA 需要考虑独特的标准，以确保租户隔离等关键的 SaaS 最佳实践仍然是 OPA 实施的一部分。这些标准包括 OPA 部署模式、租户隔离和 OPA 文档模型以及租户入职。这些因素都会影响 OPA 的最佳设计，因为它与多租户应用程序有关。

尽管本节的讨论侧重于OPA，但一般概念植根于孤立思维及其提供的指导。SaaS 应用程序必须始终将租户隔离视为其设计的一部分，这种一般的隔离原则延伸到将 OPA 包含在 SaaS 应用程序中。如果使用得当，OPA 可以成为如何在 SaaS 应用程序中强制隔离的关键部分。本节还引用了核心的SaaS隔离模型，例如孤立的SaaS模型和池化的SaaS 模型。有关更多信息，请参阅 Well-Architected Framework Saa AWS S 镜头中的核心隔离概念。

设计注意事项：

- 比较集中式和分布式部署模式
- 使用 OPA 文档模型进行租户隔离
- 租户入职

比较集中式和分布式部署模式

您可以采用集中式或分布式部署模式部署 OPA，多租户应用程序的理想方法取决于用例。有关这些模式的示例，请参阅本指南前面的 “在 API 上使用带有 PEP 的集中式 PDP 和在 API 上使用分布式 PDP 和 PE P 部分。由于 OPA 可以作为守护程序部署在操作系统或容器中，因此可以通过多种方式实现以支持多租户应用程序。

在集中部署模式中，OPA 作为容器或守护程序部署，其 RESTful API 可供应用程序中的其他服务使用。当一项服务需要 OPA 做出决策时，会调用中央 OPA RESTful API 来做出此决定。这种方法易于部署和维护，因为只有一个部署 OPA。这种方法的缺点是，它没有提供任何机制来维持租户数据的分离。由于 OPA 只有一个部署，因此 OPA 决策中使用的所有租户数据，包括 OPA 引用的任何外部数据，都必须可用于 OPA。您可以使用这种方法保持租户数据隔离，但必须通过 OPA 的策略和文档结构或对外部数据的访问来强制执行。集中部署模式还需要更高的延迟，因为每个授权决策都必须对其他服务进行 RESTful API 调用。

在分布式部署模式中，OPA 作为容器或守护程序与多租户应用程序的服务一起部署。它可以作为 sidecar 容器部署，也可以部署为在操作系统上本地运行的守护程序。要从 OPA 中检索决策，该服务会对本地 OPA 部署进行 RESTful API 调用。（由于 OPA 可以作为 Go 包部署，因此您可以原生使用 Go 来检索决策，而不必使用 RESTful API 调用。）与集中式部署模式不同，分布式模式需要付出更

大的努力来部署、维护和更新，因为它存在于应用程序的多个区域。分布式部署模式的一个好处是能够保持租户数据的隔离，特别是对于使用孤立的 SaaS 模型的应用程序。租户特定的数据可以在特定于该租户的 OPA 部署中隔离，因为分布式模型中的 OPA 是与租户一起部署的。此外，分布式部署模式的延迟要比集中式部署模式低得多，因为每个授权决策都可以在本地做出。

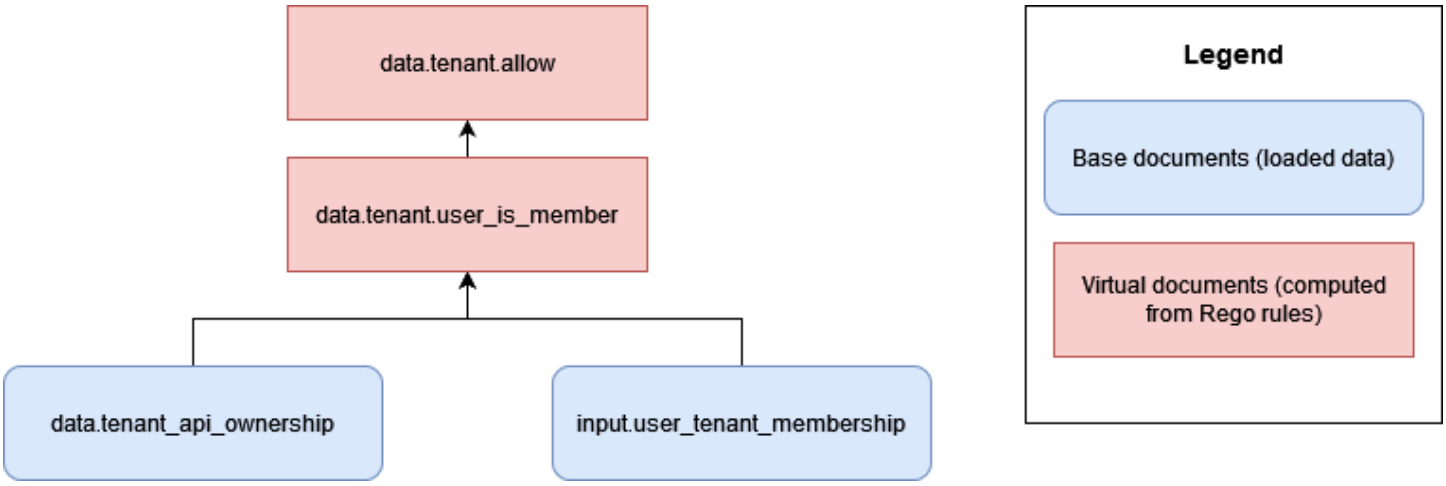
在多租户应用程序中选择 OPA 部署模式时，请务必评估对您的应用程序最重要的标准。如果您的多租户应用程序对延迟很敏感，则分布式部署模式可以提供更好的性能，但会牺牲更复杂的部署和维护。尽管您可以通过 DevOps 和自动化来管理其中的一些复杂性，但与集中式部署模式相比，它仍然需要付出额外的努力。

如果您的多租户应用程序使用孤立的 SaaS 模型，则可以使用分布式 OPA 部署模式来模仿租户数据隔离的孤立方法。这是因为当 OPA 与每个租户特定的应用程序服务一起运行时，您可以自定义每个 OPA 部署，使其仅包含与该租户关联的数据。在集中式 OPA 部署模式中孤立 OPA 数据是不可能的。如果您将集中部署模式或分布式模式与池化 SaaS 模型结合使用，则必须在 OPA 文档模型中维护租户数据隔离。

使用 OPA 文档模型进行租户隔离

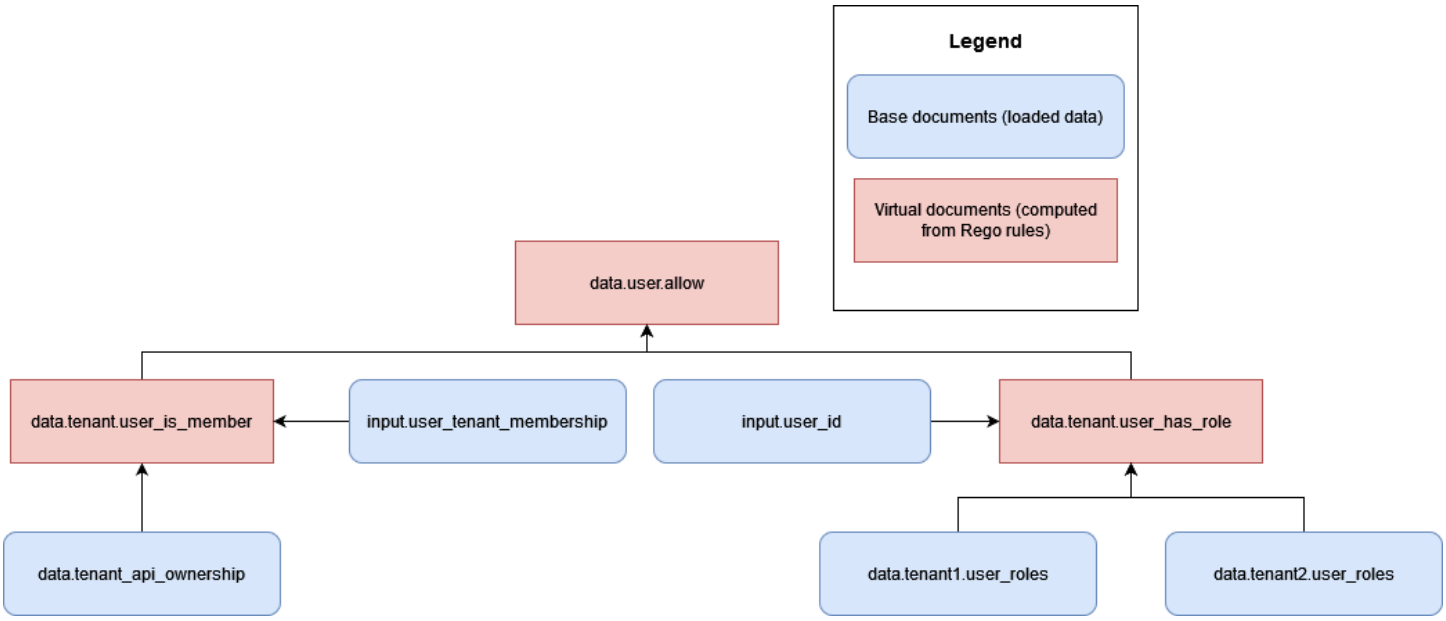
OPA 使用文件来做出决定。这些文档可能包含租户特定的数据，因此您必须考虑如何保持租户数据隔离。OPA 文档由基础文档和虚拟文档组成。基础文档包含来自外部世界的数据。这包括直接提供给 OPA 的数据、有关 OPA 请求的数据以及可能作为输入传递给 OPA 的数据。虚拟文档由策略计算，包括 OPA 政策和规则。有关更多信息，请参阅 OPA 文档。

要在 OPA 中为多租户应用程序设计文档模型，必须首先考虑在 OPA 中做出决策所需的基础文档类型。如果这些基础文档包含租户特定的数据，则必须采取措施确保这些数据不会意外暴露给跨租户访问。幸运的是，在许多情况下，在OPA中做出决策不需要租户特定的数据。以下示例显示了一个假设的 OPA 文档模型，该模型允许根据哪个租户拥有 API 以及用户是否为租户成员来访问 API，如输入文档所示。



在这种方法中，除了有关哪些租户拥有 API 的信息外，OPA 无法访问任何租户特定的数据。在这种情况下，不必担心 OPA 会促进跨租户访问，因为 OPA 用于做出访问决策的唯一信息是用户与租户的关联以及租户与 API 的关联。您可以将这种类型的 OPA 文档模型应用于孤立的 SaaS 模型，因为每个租户都拥有独立资源的所有权。

但是，在许多 RBAC 授权方法中，有可能跨租户泄露信息。在以下示例中，假设的 OPA 文档模型允许根据用户是否为租户成员以及该用户是否具有访问 API 的正确角色来访问 API。



这种模式引入了跨租户访问的风险，因为现在`data.tenant2.user_roles`必须允许 OPA 访问`data.tenant1.user_roles`和中的多个租户的角色和权限才能做出授权决定。为了保持租户隔离和角色映射的隐私，这些数据不应存在于 OPA 中。RBAC 数据应位于外部数据源（例如数据库）中。此外，不应使用 OPA 将预定义的角色映射到特定权限，因为这会使租户难以定义自己的角色和权限。它还会使您的授权逻辑变得僵化，需要不断更新。有关如何将 RBAC 数据安全地纳入 OPA 决策过程的指导，请参阅本指南后面的租户隔离和数据隐私建议部分。

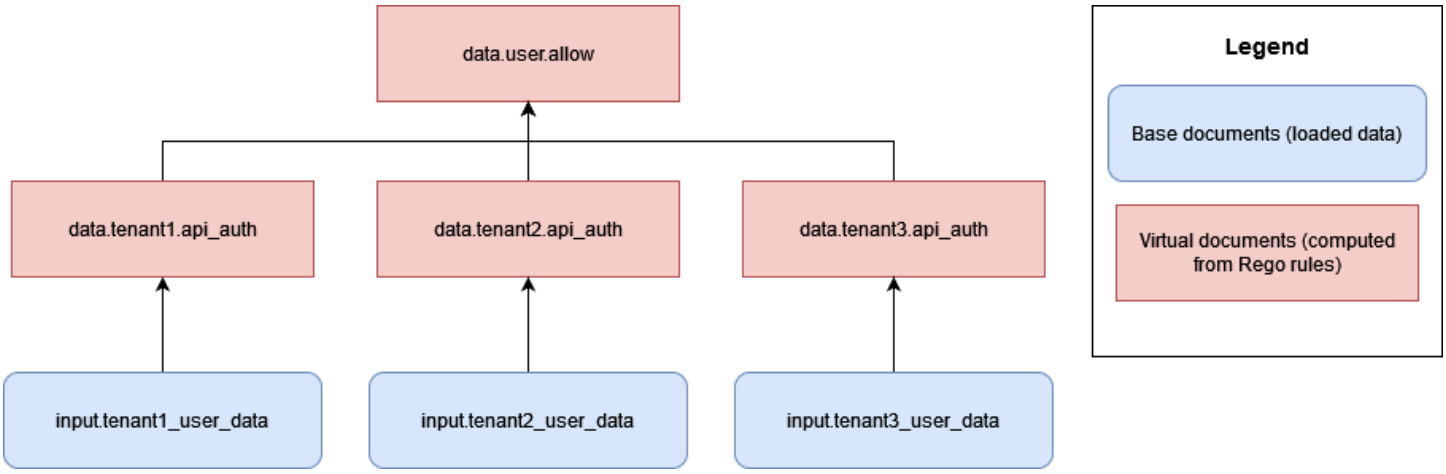
通过不将任何租户特定的数据存储为异步基础文档，您可以轻松地在 OPA 中保持租户隔离。异步基础文档是存储在内存中的数据，可以在 OPA 中定期更新。其他基础文档（例如 OPA 输入）是同步传递的，并且仅在决策时才可用。例如，将租户特定的数据作为 OPA 输入的一部分提供给查询并不构成违反租户隔离的行为，因为这些数据只能在决策过程中同步获得。

租户入职

OPA 文件的结构必须允许租户在不引入繁琐要求的情况下直接入职。您可以使用文件包在 OPA 文档模型层次结构中组织虚拟文档，这些文件包可以包含许多规则。在为多租户应用程序规划 OPA 文档模

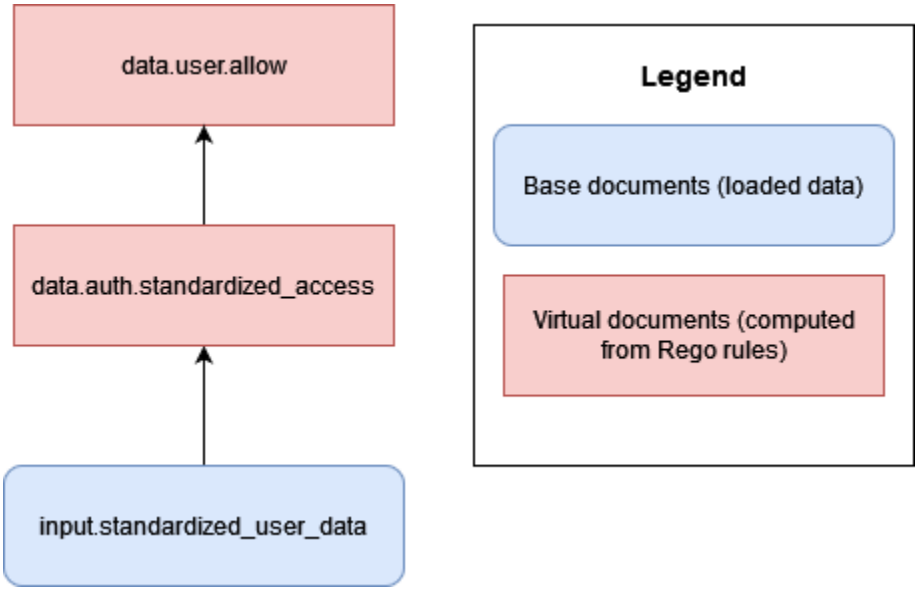
型时，请首先确定 OPA 需要哪些数据才能做出决策。您可以提供数据作为输入，将其预加载到 OPA 中，或者在决策时或定期从外部数据源提供。有关在 OPA 中使用外部数据的更多信息，请参阅本指南后面的 “在 OP A 中检索 PDP 的外部数据” 一节。

确定在 OPA 中做出决策所需的数据后，请考虑如何实施以包形式组织的 OPA 规则，以便使用这些数据做出决策。例如，在孤立的 SaaS 模型中，每个租户可能对授权决策的制定方式有独特的要求，您可以实施特定于租户的 OPA 一揽子规则。



这种方法的缺点是，您必须为添加到 SaaS 应用程序的每个租户添加一组针对每个租户的新的 OPA 规则。这既繁琐又难以扩展，但可能不可避免，具体取决于租户的要求。

或者，在池化 SaaS 模型中，如果所有租户都根据相同的规则做出授权决策并使用相同的数据结构，则可以使用具有普遍适用规则的标准 OPA 包，以便更轻松地加入租户并扩展 OPA 的实施。



在可能的情况下，我们建议您使用通用的 OPA 规则和软件包（或虚拟文档），根据每个租户提供的标准化数据做出决策。这种方法使得 OPA 易于扩展，因为您只需更改为每个租户提供给 OPA 的数据，

而不更改 OPA 通过其规则提供决策的方式。只有当个别租户需要独特的决策或必须向OPA提供与其他租户不同的数据时，才需要引入 rules-per-tenant 模型。

DevOps、监控、记录和检索 PDP 的数据

在这个提议的授权模式中，策略集中在授权服务中。这种集中化是经过深思熟虑的，因为本指南中讨论的设计模型的目标之一是实现策略脱钩，或者从应用程序中的其他组件中删除授权逻辑。Amazon Verified Permissions 和开放政策代理 (OPA) 都提供了在需要更改授权逻辑时更新策略的机制。

对于已验证权限， AWS 软件开发工具包提供了以编程方式更新策略的机制（参见 Amazon 已验证权限 API 参考指南）。使用 SDK，您可以按需推送新政策。此外，由于 Verified Permissions 是一项托管服务，因此您无需管理、配置或维护控制平面或代理即可执行更新。但是，我们建议您使用持续集成和持续部署 (CI/CD) 管道来管理使用软件开发工具包的已验证权限策略存储的部署和策略更新。 AWS

通过验证的权限，可以轻松访问可观察性功能。可以将其配置为记录对亚马逊 CloudWatch 日志组 AWS CloudTrail、Amazon Simple Storage Service (Amazon S3) 存储桶或 Amazon Data Firehose 传输流的所有访问尝试，从而能够快速响应安全事件和审计请求。此外，您可以通过监控已验证权限服务的运行状况 AWS Health Dashboard。由于 Verified Permissions 是一项托管服务 AWS，因此其运行状况由维护，您可以使用其他 AWS 托管服务来配置可观察性功能。

就 OPA 而言，REST API 提供了以编程方式更新策略的方法。您可以将 API 配置为从既定位置提取新版本的策略包或按需推送策略。此外，OPA 还提供基本的发现服务，通过该服务，可以动态配置新代理，并由分发发现包的控制平面进行集中管理。（OPA 的控制平面必须由 OPA 操作员设置和配置。）无论策略引擎是已验证权限、OPA 还是其他解决方案，我们都建议您创建强大的 CI/CD 管道来进行版本控制、验证和更新策略。

对于 OPA，控制平面还提供监控和审计选项。您可以将包含 OPA 授权决策的日志导出到远程 HTTP 服务器以进行日志聚合。这些决策日志对于审计非常有用。

如果您正在考虑采用一种将访问控制决策与应用程序分开的授权模型，请确保您的授权服务具有有效的监控、日志记录和 CI/CD 管理功能，用于加入新 PDP 或更新策略。

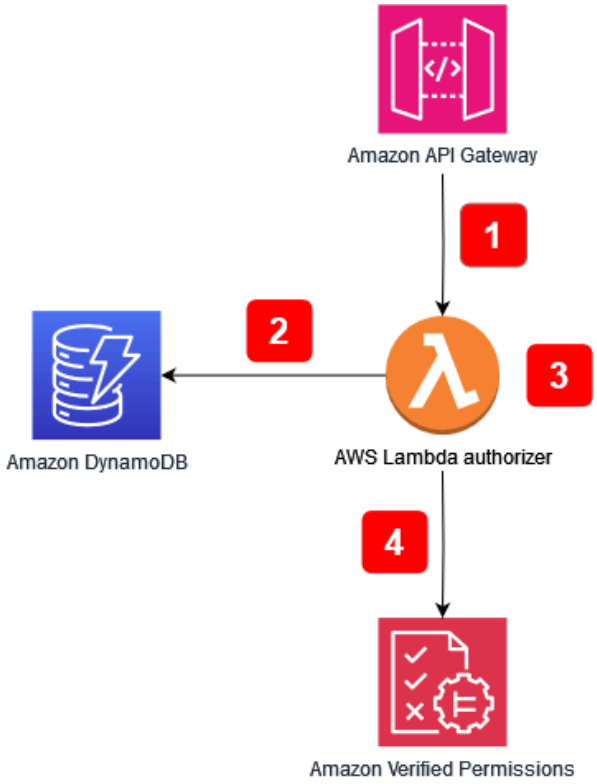
主题

在 Amazon 已验证权限中检索 PDP 的外部数据

Amazon Verified Permissions 不支持检索 PDP 的外部数据，但它可以将用户提供的数据存储为其架构的一部分。与 OPA 一样，如果授权决策的所有数据都可以作为授权请求的一部分或作为请求的一部分

传递的 JSON Web Token (JWT) 的一部分提供，则无需进行其他配置。但是，作为调用 “已验证权限” 的应用程序授权服务的一部分，您可以通过授权请求向已验证权限提供来自外部来源的额外数据。例如，应用程序的授权方服务可以向外部来源（例如 DynamoDB 或 Amazon RDS）查询数据，然后这些服务可以将外部提供的数据作为授权请求的一部分。

下图显示了如何检索其他数据并将其合并到已验证权限授权请求中的示例。可能需要使用此方法来检索 RBAC 角色映射之类的数据，检索与资源或委托人相关的其他属性，或者在数据位于应用程序的不同部分且无法通过身份提供者 (IdP) 令牌提供的情况下。



申请流程：

1. 应用程序接收到对 Amazon API Gateway 的 API 调用，并将该调用转发给 AWS Lambda 授权方。
2. Lambda 授权机构调用 Amazon DynamoDB 来检索有关提出请求的委托人的其他数据。
3. Lambda 授权方将其他数据合并到向已验证权限发出的授权请求中。
4. Lambda 授权者向已验证的权限发出授权请求并收到授权决定。

该图表包括 Amazon API Gateway 的一项名为 Lambda 授权者的功能。尽管此功能可能不适用于其他服务或应用程序提供的 API，但您可以跨多个用例复制使用授权方提取其他数据以整合到已验证权限授权请求中的常规模型。

在 OPA 中检索 PDP 的外部数据

对于 OPA，如果授权决策所需的所有数据都可以作为输入提供，或者作为查询组成部分传递的 JSON Web Token (JWT) 的一部分提供，则无需进行其他配置。（将 JWT 和 SaaS 上下文数据作为查询输入的一部分传递给 OPA 相对简单。）OPA 可以通过所谓的重载输入方法接受任意 JSON 输入。如果 PDP 需要的数据超出了可以作为输入或 JWT 包含的范围，OPA 会提供多种检索这些数据的选项。其中包括捆绑、推送数据（复制）和动态数据检索。

OPA 捆绑销售

OPA 捆绑功能支持以下外部数据检索流程：

1. 政策执行点 (PEP) 要求做出授权决定。
2. OPA 下载新的政策包，包括外部数据。
3. 捆绑服务从数据源复制数据。

当您使用捆绑功能时，OPA 会定期从集中式捆绑服务中下载策略和数据包。（OPA 不提供捆绑服务的实现和设置。）从捆绑服务中提取的所有策略和外部数据都存储在内存中。如果外部数据大小太大而无法存储在内存中，或者数据变化过于频繁，则此选项将不起作用。

有关捆绑功能的更多信息，请参阅 OPA 文档。

OPA 复制（推送数据）

OPA 复制方法支持以下外部数据检索流程：

1. PEP 要求做出授权决定。
2. 数据复制器将数据推送到 OPA。
3. 数据复制器复制来自数据源的数据。

在捆绑方法的这种替代方案中，数据被推送到OPA，而不是由OPA定期提取。（OPA 不提供复制器的实现和设置。）推送方法与捆绑方法具有相同的数据大小限制，因为 OPA 将所有数据存储在内存中。推送选项的主要优点是，您可以使用增量更新 OPA 中的数据，而不必每次都替换所有外部数据。这使得推送选项更适合频繁变化的数据集。

有关复制选项的更多信息，请参阅 OPA 文档。

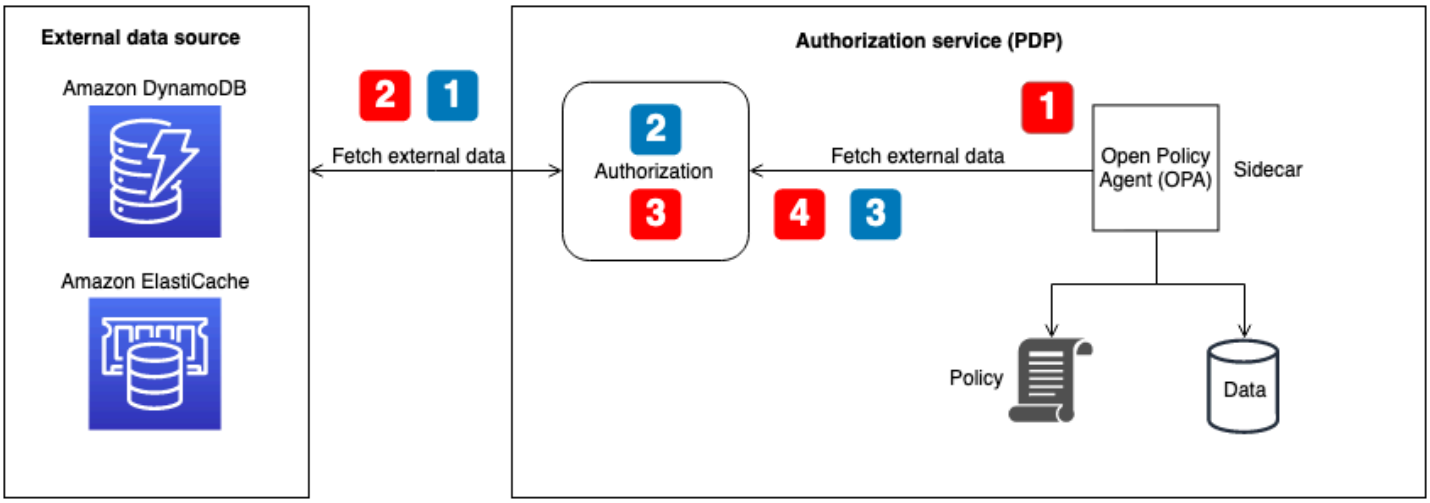
OPA 动态数据检索

如果要检索的外部数据太大而无法缓存在OPA的存储器中，则可以在评估授权决策期间从外部来源动态提取数据。使用这种方法时，数据始终是最新的。这种方法有两个缺点：网络延迟和可访问性。目前，OPA 只能通过 HTTP 请求在运行时检索数据。如果转到外部数据源的调用无法以 HTTP 响应的形式返回数据，则它们需要自定义 API 或其他机制才能向 OPA 提供这些数据。由于 OPA 只能通过 HTTP 请求检索数据，而且检索数据的速度至关重要，因此我们建议您尽可能使用 AWS 服务 诸如 Amazon DynamoDB 之类的设备来保存外部数据。

有关拉取方法的更多信息，请参阅 OPA 文档。

使用授权服务与 OPA 一起实施

当您使用捆绑、复制或动态拉取方法获取外部数据时，我们建议授权服务为这种交互提供便利。这是因为授权服务可以检索外部数据并将其转换为 JSON，以便 OPA 做出授权决策。下图显示了授权服务如何使用这三种外部数据检索方法发挥作用。



检索 OPA 流程的外部数据 — 在决策时进行捆绑数据或动态数据检索（图中用红色编号标注说明）：

1. OPA 调用授权服务的本地 API 端点，该服务被配置为捆绑端点或在授权决策期间用于动态数据检索的端点。
2. 授权服务查询或调用外部数据源来检索外部数据。（对于捆绑终端节点，此数据还应包含 OPA 策略和规则。 Bundle 更新会替换 OPA 缓存中的所有内容（包括数据和策略）。）
3. 授权服务对返回的数据执行任何必要的转换，以将其转换为预期的 JSON 输入。
4. 数据将返回到 OPA。它被缓存在内存中用于捆绑包配置，并立即用于动态授权决策。

检索 OPA 流程的外部数据 — replicator（图中用蓝色编号的标注说明）：

1. 复制器（授权服务的一部分）调用外部数据源并检索要在 OPA 中更新的任何数据。这可能包括策略、规则和外部数据。此调用可以按设定的节奏进行，也可以响应外部源中的数据更新。
2. 授权服务对返回的数据执行任何必要的转换，以将其转换为预期的 JSON 输入。
3. 授权服务调用 OPA 并将数据缓存在内存中。授权服务可以有选择地更新数据、策略和规则。

关于租户隔离和数据隐私的建议

上一节提供了几种使用具有 OPA 和 Amazon 验证权限的外部数据来帮助做出授权决策的方法。在可能的情况下，我们建议您使用重载输入方法将 SaaS 上下文数据传递给 OPA 以做出授权决策，而不是将数据存储在 OPA 的内存中。此用例不适用于 AWS Cloud Map，因为它不支持在服务中存储外部数据。

在基于角色的访问控制 (RBAC) 或 RBAC 和基于属性的访问控制 (ABAC) 混合模型中，仅通过授权请求或查询提供的数据可能不足，因为必须引用角色和权限才能做出授权决策。为了保持租户隔离和角色映射的隐私，这些数据不应存在于 OPA 中。RBAC 数据应位于外部数据源（例如数据库）中，或者应作为 IdP 在 JWT 中的声明的一部分传递。在 Verified Permissions 中，RBAC 数据可以作为每租户策略存储模型中的策略和架构的一部分进行维护，因为每个租户都有自己的逻辑上分离的策略存储。但是，在一个共享的多租户策略存储模型中，为了保持租户隔离，角色映射数据不应位于已验证权限中。

此外，不应使用 OPA 和已验证权限将预定义的角色映射到特定权限，因为这会使租户难以定义自己的角色和权限。它还会使您的授权逻辑变得僵化，需要不断更新。本指南的例外情况是 Verified Permissions 中的每租户策略存储模型，因为该模型允许每个租户拥有自己的策略，这些策略可以根据每个租户进行独立评估。

Amazon Verified Permissions

只有在架构中，经过验证的权限才能存储潜在的私有 RBAC 数据。这在每租户策略存储模式中是可以接受的，因为每个租户都有自己的逻辑上独立的策略存储。但是，在一个共享的多租户策略存储模式中，它可能会损害租户隔离。如果需要这些数据才能做出授权决定，则应从 DynamoDB 或 Amazon RDS 等外部来源检索这些数据，并将其合并到已验证的权限授权请求中。

OPA

OPA 用于维护 RBAC 数据的隐私和租户隔离的安全方法包括使用动态数据检索或复制来获取外部数据。这是因为您可以使用上图所示的授权服务，仅提供租户特定或用户特定的外部数据，以便做出授权决策。例如，当用户登录时，您可以使用复制器向 OPA 缓存提供 RBAC 数据或权限矩阵，并根据输入

数据中提供的用户引用数据。您可以对动态提取的数据使用类似的方法，仅检索相关数据以做出授权决策。此外，在动态数据检索方法中，这些数据不必缓存在OPA中。在维护租户隔离方面，捆绑方法不如动态检索方法有效，因为它会更新 OPA 缓存中的所有内容，并且无法处理精确的更新。捆绑模型仍然是更新 OPA 政策和非 RBAC 数据的好方法。

最佳实践

本节列出了本指南中的一些高级要点。有关每点的详细讨论，请点击相应章节的链接。

选择适用于您的应用程序的访问控制模型

本指南讨论了几种访问控制模型。根据您的应用程序和业务需求，您应该选择适合自己的型号。考虑如何使用这些模型来满足您的访问控制需求，以及您的访问控制需求可能如何演变，这需要对所选方法进行更改。

实施 PDP

政策决策点 (PDP) 可以描述为策略或规则引擎。此组件负责应用策略或规则，并返回有关是否允许特定访问的决定。PDP 允许将应用程序代码中的授权逻辑转移到单独的系统中。这可以简化应用程序代码。它还提供了一个 easy-to-use 等性接口，用于为 API、微服务、前端后端 (BFF) 层或任何其他应用程序组件做出授权决策。PDP 可用于在整个应用程序中一致地强制执行租赁要求。

为应用程序中的每个 API 实现 PEP

实施策略实施点 (PEP) 需要确定应在应用程序中何处实施访问控制。首先，在应用程序中找到可以整合 PEP 的点。在决定在哪里添加 PEP 时，请考虑以下原则：

如果应用程序公开了 API，则应对该API进行授权和访问控制。

考虑使用 Amazon 验证权限或 OPA 作为您的 PDP 的策略引擎

与自定义策略引擎相比，Amazon 验证权限具有优势。它是一项可扩展、精细的权限管理和授权服务，适用于您构建的应用程序。它支持使用高级声明性开源语言 Cedar 编写策略。因此，与实施自己的解决方案相比，使用已验证权限实现策略引擎所需的开发工作更少。此外，经过验证的权限是完全托管的，因此您不必管理底层基础架构。

与自定义策略引擎相比，开放策略代理 (OPA) 具有优势。OPA 及其对 Rego 的策略评估提供了一个灵活的预建策略引擎，支持使用高级声明性语言编写策略。这使得实施策略引擎所需的工作量大大低于构建自己的解决方案。此外，OPA 正在迅速成为一种备受支持的授权标准。

为 OPA 实现控制平面 DevOps，用于监控和记录

由于 OPA 不提供通过源代码控制更新和跟踪授权逻辑更改的方法，因此我们建议您实现控制平面来执行这些功能。这将允许更轻松地将更新分发给 OPA 代理，特别是在 OPA 在分布式系统中运行时，这将减轻使用 OPA 的管理负担。此外，控制平面还可用于收集日志以进行聚合和监控 OPA 代理的状态。

在“已验证权限”中配置日志记录和可观察性功能

通过验证的权限，可以轻松访问可观察性功能。您可以将该服务配置为记录所有访问尝试 AWS CloudTrail、Amazon CloudWatch 日志组、S3 存储桶或 Amazon Data Firehose 传输流，从而能够快速响应安全事件和审计请求。此外，您可以通过监控服务的运行状况 AWS Health Dashboard。由于 Verified Permissions 是一项托管服务 AWS，因此其运行状况由维护，您可以使用其他 AWS 托管服务来配置其可观察性功能。

使用 CI/CD 管道在已验证权限中配置和更新策略存储和策略

Verified Permissions 是一项托管服务，因此您无需管理、配置或维护控制平面或代理即可执行更新。但是，我们仍然建议您使用持续集成和持续部署 (CI/CD) 管道来管理已验证权限策略存储的部署以及使用软件开发工具包的策略更新。 AWS当您更改已验证权限资源时，这项工作可以省去手动操作并降低操作员出错的可能性。

确定授权决策是否需要外部数据，然后选择适合该数据的模型

如果 PDP 可以仅基于 JSON Web Token (JWT) 中包含的数据做出授权决策，则通常无需导入外部数据来帮助做出这些决策。如果您使用已验证权限或 OPA 作为 PDP，它也可以接受作为请求的一部分传递的其他输入，即使这些数据未包含在 JWT 中。对于已验证的权限，您可以为其他数据使用上下文参数。对于 OPA，您可以使用 JSON 数据作为重载输入。如果您使用 JWT，则上下文或重载输入法通常比在其他来源中维护外部数据容易得多。如果需要更复杂的外部数据来做出授权决策，OPA 提供了几种检索外部数据的模型，而且 Verified Permissions 可以通过使用授权服务引用外部来源来补充其授权请求中的数据。

常见问题解答

本节提供了有关在多租户 SaaS 应用程序中实施 API 访问控制和授权的常见问题的答案。

问：授权和身份验证有什么区别？

答：身份验证是验证用户身份的过程。授权向用户授予访问特定资源的权限。

问：SaaS 应用程序中的授权和租户隔离有什么区别？

答：租户隔离是指在SaaS系统中使用的明确机制，用于确保每个租户的资源，即使在共享基础架构上运行也是如此，也是如此。多租户授权是指对入站操作进行授权，并防止这些操作在错误的租户上实施。假设的用户可以经过身份验证和授权，但可能仍能访问其他租户的资源。身份验证和授权并不一定会阻止这种访问，但是要实现这一目标，需要租户隔离。有关这两个概念的更多信息，请参阅《AWS SaaS 架构基础知识》白皮书中的租户隔离讨论。

问：为什么我需要考虑对我的 SaaS 应用程序进行租户隔离？

答：SaaS 应用程序有多个租户。租户可以是客户组织或使用该 SaaS 应用程序的任何外部实体。根据应用程序的设计方式，这意味着租户可能正在访问共享的 API、数据库或其他资源。重要的是要保持租户隔离（即严格控制资源访问权限的结构，并阻止任何访问其他租户资源的尝试），以防止一个租户的用户访问另一个租户的私人信息。SaaS 应用程序通常旨在确保在整个应用程序中保持租户隔离，并且租户只能访问自己的资源。

问：为什么我需要访问控制模型？

答：访问控制模型用于创建一种一致的方法来确定如何授予对应用程序中资源的访问权限。这可以通过向与业务逻辑紧密一致的用户分配角色来完成，也可以基于其他上下文属性，例如一天中的时间或用户是否满足预定义的条件。访问控制模型构成了应用程序在做出授权决策以确定用户权限时使用的基本逻辑。

问：我的应用程序是否需要 API 访问控制？

答：是的。API 应始终验证调用方是否具有适当的访问权限。无处不在的 API 访问控制还可确保仅根据租户授予访问权限，从而保持适当的隔离。

问：为什么建议使用策略引擎或 PDP 进行授权？

答：策略决策点 (PDP) 允许将应用程序代码中的授权逻辑转移到单独的系统中。这可以简化应用程序代码。它还提供了一个 easy-to-use 等性接口，用于为 API、微服务、前端后端 (BFF) 层或任何其他应用程序组件做出授权决策。

问：什么是 PEP？

答：政策执行点 (PEP) 负责接收发送给 PDP 进行评估的授权请求。PEP 可以是应用程序中必须保护数据和资源的任何地方，也可以是应用授权逻辑的地方。与 PDP 相比，PEP 相对简单。PEP 仅负责请求和评估授权决定，不需要将任何授权逻辑纳入其中。

问：我应该如何在 Amazon 验证权限和 OP A 之间做出选择？

答：要在已验证权限和开放政策代理 (OPA) 之间进行选择，请始终牢记您的用例和您的独特要求。Verified Permissions 提供了一种完全托管的方式，用于定义细粒度权限、审核跨应用程序的权限以及集中管理应用程序的策略管理系统，同时通过毫秒级处理满足应用程序延迟要求。OPA 是一个开源的通用策略引擎，还可以帮助您在应用程序堆栈中统一策略。为了运行 OPA，您需要将其托管在您的 AWS 环境中，通常使用容器或 AWS Lambda 函数。

已验证权限使用开源 Cedar 策略语言，而 OPA 使用 Rego。因此，熟悉其中一种语言可能会影响您选择该解决方案。但是，我们建议您阅读这两种语言的相关知识，然后从您想要解决的问题中进行回顾，以找到最适合您的用例的解决方案。

问：除了已验证权限和 OP A 之外，还有其他开源替代方案吗？

答：有一些开源系统类似于已验证权限和开放策略代理 (OPA)，例如通用表达语言 (CEL)。本指南重点介绍作为可扩展权限管理和细粒度授权服务的已验证权限，以及 OPA，后者已被广泛采用、记录在案，可适应许多不同类型的应用程序和授权要求。

问：我需要编写授权服务才能使用 OPA，还是可以直接与 OPA 交互？

答：您可以直接与 OPA 互动。本指南中的授权服务是指将授权决策请求转换为 OPA 查询（反之亦然）的服务。如果您的应用程序可以直接查询和接受 OPA 响应，则无需引入这种额外的复杂性。

问：如何监控我的 OPA 代理的正常运行时间和审计目的？

答：OPA 提供日志记录和基本的正常运行时间监控，尽管其默认配置可能不足以用于企业部署。有关更多信息，请参阅本DevOps指南前面的 “监控和记录” 部分。

问：如何监控已验证的权限以实现正常运行时间和审计目的？

答：已验证权限是一项 AWS 托管服务，可以通过监控其可用性 AWS Health Dashboard。此外，经过验证的权限还可以登录亚马逊 CloudWatch 日志 AWS CloudTrail、亚马逊 S3 和亚马逊 Data Firehose。

问：我可以使用哪些操作系统和 AWS 服务来运行 OPA？

答：你可以在 macOS、Windows 和 Linux 上运行 O PA。OPA 代理可以在亚马逊弹性计算云 (Amazon EC2) 代理以及容器化服务上配置，例如亚马逊弹性容器服务 (Amazon ECS) 和亚马逊弹性 Kubernetes 服务 (Amazon EKS)。

问：我可以使用哪些操作系统和 AWS服务来运行已验证的权限？

答：已验证权限是一项 AWS 托管服务，由运营 AWS。除了能够向服务发出授权请求外，无需进行其他配置、安装或托管即可使用经过验证的权限。

问：我可以运行 OPA 吗？ AWS Lambda

答：你可以在 Lambda 上以 Go 库的形式运行 OPA。有关如何为 API Gateway Lambda 授权方执行此操作的信息，请参阅 AWS 博客文章使用开放策略代理创建自定义 Lambd a 授权方。

问：我应该如何在分布式 PDP 和集中式 PDP 方法之间做出选择？

答：这取决于您的应用程序。它很可能是根据分布式和集中式 PDP 模型之间的延迟差异来确定的。我们建议您构建概念验证并测试应用程序的性能，以验证您的解决方案。

问：除了 API 之外，我还能将 OPA 用于 OPA 吗？

答：是的。OPA 文档提供了 Kubernetes、En vo y、Docker、K afka、SSH 和 s udo 以及 Terraform 的示例。此外，OPA 可以使用 Rego 部分规则对查询返回任意 JSON 响应。根据查询的不同，OPA 可用于通过 JSON 响应回答许多问题。

问：除了 API 之外，我还能将经过验证的权限用于用例吗？

答：是的。已验证的权限可以为其收到的任何授权请求提供`ALLOW`或`DENY`响应。已验证的权限可以为任何需要授权决策的应用程序或服务提供授权响应。

问：我能否使用 IAM 策略语言在已验证权限中创建策略？

答：不是。 您必须使用 Cedar 政策语言来撰写政策。Cedar 旨在支持客户应用程序资源的权限管理，而 AWS Identity and Access Management (IAM) 策略语言则演变为支持 AWS 资源的访问控制。

后续步骤

通过采用与语言无关的标准化方法来做出授权决策，可以克服多租户 SaaS 应用程序授权和 API 访问控制的复杂性。这些方法包括政策决策点 (PDP) 和政策执行点 (PDP)，它们以灵活和普遍的方式强制执行访问权限。可将多种访问控制方法，例如基于角色的访问控制（RBAC）、基于属性的访问权限控制（ABAC）或二者的组合，合并到内聚的访问控制策略中。从应用程序中移除授权逻辑，可以消除在应用程序代码中包含临时解决方案以处理访问控制的开销。本指南中讨论的实施和最佳实践旨在为在多租户 SaaS 应用程序中实施授权和 API 访问控制的方法提供信息和标准化。您可以将本指南用作收集信息并为应用程序设计强大的访问控制和授权系统的第一步。接下来的步骤：

- 查看授权和租户隔离需求，并为应用程序选择访问控制模型。
- 使用 Amazon 验证权限或开放策略代理 (OPA)，或者编写自己的自定义策略引擎，构建用于测试的概念验证。
- 确定应用程序中应实施 PEP 的 API 和位置。

资源

参考

- Amazon 已验证权限文档（AWS 文档）
- 如何使用 Amazon 验证权限进行授权（AWS 博客文章）
- 使用亚马逊验证权限为 ASP.NET 核心应用程序实施自定义授权策略提供程序（AWS 博客文章）
- 使用 Amazon 验证权限通过 PBAC 管理角色和权利（AWS 博客文章）
- 使用基于每租户策略存储的 Amazon 验证权限进行 SaaS 访问控制（AWS 博客文章）
- OPA 官方文档
- 为什么企业必须拥抱最近毕业的CNCF项目——开放政策代理（Janakiram MSV 在《福布斯》上发表的文章，2021年2月8日）
- 使用开放策略代理创建自定义 Lambda 授权方（AWS 博客文章）
- 通过 Open Policy Agent 使用 AWS Cloud Development Kit 实现策略即代码（AWS 博客文章）
- 云治理和合规性 AWS 以政策即代码（AWS 博客文章）
- 在 Amazon EKS 上使用开放政策代理（AWS 博客文章）
- 使用开放政策代理、Amazon 和 AWS Lambda（AWS 博客文章）对 Amazon EventBridge ECS 进行合规即代码
- Kubernetes 基于策略的对策 — 第 1 部分（博客文章）AWS
- 使用 API Gateway Lambda 授权方（文档）AWS

工具

- Cedar Play ground（用于在浏览器中测试 Cedar）
- 雪松 Github
- 雪松语言参考
- Rego Playgro und（用于在浏览器中测试 Rego）
- OPA 存储库 GitHub

合作伙伴

- Identity and Access Management

- 应用程序安全合作伙伴
- 云治理合作伙伴
- 安全与合规合作伙伴
- 安全运营和自动化合作伙伴
- 安全工程合作伙伴

文档历史记录

下表介绍了本指南的一些重要更改。如果您希望收到有关未来更新的通知，可以订阅 RSS 源。

变更	说明	日期
添加了 Amazon 验证权限的详细信息和示例	添加了有关使用 Amazon 验证权限实施 PDP 的详细讨论、示例和代码。新章节包括： • 使用 Amazon 验证权限实施 PDP • 为 Amazon 验证权限设计模型 • Amazon 已验证权限多租户设计注意事项 • 在 Amazon 已验证权限中检索 PDP 的外部数据	2024 年 5 月 28 日
澄清信息	阐明了在 API 设计模型上使用 PEP 的分布式 PDP。	2024 年 1 月 10 日
添加了有关新 AWS 服务的简要信息	添加了有关 Amazon 验证权限的信息，该权限提供的功能和优势与 OPA 相同。	2023 年 5 月 22 日
—	初次发布	2021 年 8 月 17 日

AWS 规范性指导词汇表

以下是 AWS 规范性指导提供的策略、指南和模式中的常用术语。若要推荐词条，请使用术语表末尾的提供反馈链接。

数字

7 R

将应用程序迁移到云中的 7 种常见迁移策略。这些策略以 Gartner 于 2011 年确定的 5 R 为基础，包括以下内容：

- 重构/重新架构 - 充分利用云原生功能来提高敏捷性、性能和可扩展性，以迁移应用程序并修改其架构。这通常涉及到移植操作系统和数据库。示例：将您的本地 Oracle 数据库迁移到兼容 Amazon Aurora PostgreSQL 的版本。
- 更换平台 - 将应用程序迁移到云中，并进行一定程度的优化，以利用云功能。示例：在中将您的本地 Oracle 数据库迁移到适用于 Oracle 的亚马逊关系数据库服务 (Amazon RDS) AWS Cloud。
- 重新购买 - 转换到其他产品，通常是从传统许可转向 SaaS 模式。示例：将您的客户关系管理 (CRM) 系统迁移到 Salesforce.com。
- 更换主机（直接迁移）- 将应用程序迁移到云中，无需进行任何更改即可利用云功能。示例：在中的 EC2 实例上将您的本地 Oracle 数据库迁移到 Oracle AWS Cloud。
- 重新定位（虚拟机监控器级直接迁移）：将基础设施迁移到云中，无需购买新硬件、重写应用程序或修改现有操作。您可以将服务器从本地平台迁移到同一平台的云服务。示例：将Microsoft Hyper-V应用程序迁移到 AWS。
- 保留（重访）- 将应用程序保留在源环境中。其中可能包括需要进行重大重构的应用程序，并且您希望将工作推迟到以后，以及您希望保留的遗留应用程序，因为迁移它们没有商业上的理由。
- 停用 - 停用或删除源环境中不再需要的应用程序。

A

ABAC

请参阅基于属性的访问控制。

抽象服务

参见托管服务。

酸

参见原子性、一致性、隔离性、耐久性。

主动-主动迁移

一种数据库迁移方法，在这种方法中，源数据库和目标数据库保持同步（通过使用双向复制工具或双写操作），两个数据库都在迁移期间处理来自连接应用程序的事务。这种方法支持小批量、可控的迁移，而不需要一次性割接。与主动-被动迁移相比，它更灵活，但需要更多的工作。

主动-被动迁移

一种数据库迁移方法，在这种方法中，源数据库和目标数据库保持同步，但在将数据复制到目标数据库时，只有源数据库处理来自连接应用程序的事务。目标数据库在迁移期间不接受任何事务。

聚合函数

一个 SQL 函数，它对一组行进行操作并计算该组的单个返回值。聚合函数的示例包括SUM和MAX。

AI

参见人工智能。

AIOps

参见人工智能操作。

匿名化

永久删除数据集中个人信息的过程。匿名化可以帮助保护个人隐私。匿名化数据不再被视为个人数据。

反模式

一种用于解决反复出现的问题的常用解决方案，而在这类问题中，此解决方案适得其反、无效或不如替代方案有效。

应用程序控制

一种安全方法，仅允许使用经批准的应用程序，以帮助保护系统免受恶意软件的侵害。

应用程序组合

有关组织使用的每个应用程序的详细信息的集合，包括构建和维护该应用程序的成本及其业务价值。这些信息是产品组合发现和分析过程的关键，有助于识别需要进行迁移、现代化和优化的应用程序并确定其优先级。

人工智能（AI）

计算机科学领域致力于使用计算技术执行通常与人类相关的认知功能，例如学习、解决问题和识别模式。有关更多信息，请参阅什么是人工智能？

人工智能运营（AIOps）

使用机器学习技术解决运营问题、减少运营事故和人为干预以及提高服务质量的过程。有关如何在 AWS 迁移策略中使用 AIOps 的更多信息，请参阅运营集成指南。

非对称加密

一种加密算法，使用一对密钥，一个公钥用于加密，一个私钥用于解密。您可以共享公钥，因为它不用于解密，但对私钥的访问应受到严格限制。

原子性、一致性、隔离性、持久性（ACID）

一组软件属性，即使在出现错误、电源故障或其他问题的情况下，也能保证数据库的数据有效性和操作可靠性。

基于属性的访问权限控制（ABAC）

根据用户属性（如部门、工作角色和团队名称）创建精细访问权限的做法。有关更多信息，请参阅 AWS Identity and Access Management (I AM) 文档 AWS中的 AB AC。

权威数据源

存储主要数据版本的位置，被认为是最可靠的信息源。您可以将数据从权威数据源复制到其他位置，以便处理或修改数据，例如对数据进行匿名化、编辑或假名化。

可用区

中的一个不同位置 AWS 区域 ，不受其他可用区域故障的影响，并向同一区域中的其他可用区提供低成本、低延迟的网络连接。

AWS 云采用框架 (AWS CAF)

该框架包含指导方针和最佳实践 AWS ，可帮助组织制定高效且有效的计划，以成功迁移到云端。AWS CAF将指导分为六个重点领域，称为视角：业务、人员、治理、平台、安全和运营。业务、人员和治理角度侧重于业务技能和流程；平台、安全和运营角度侧重于技术技能和流程。例如，人员角度针对的是负责人力资源（HR）、人员配置职能和人员管理的利益相关者。从这个角度来看，AWS CAF 为人员发展、培训和沟通提供了指导，以帮助组织为成功采用云做好准备。有关更多信息，请参阅 AWS CAF 网站和 AWS CAF 白皮书。

AWS 工作负载资格框架 (AWS WQF)

一种评估数据库迁移工作负载、推荐迁移策略和提供工作估算的工具。 AWS WQF 包含在 AWS Schema Conversion Tool (AWS SCT) 中。它用来分析数据库架构和代码对象、应用程序代码、依赖关系和性能特征，并提供评测报告。

B

坏机器人

旨在破坏个人或组织或对其造成伤害的机器人。

BCP

参见业务连续性计划。

行为图

一段时间内资源行为和交互的统一交互式视图。您可以使用 Amazon Detective 的行为图来检查失败的登录尝试、可疑的 API 调用和类似的操作。有关更多信息，请参阅 Detective 文档中的行为图中的数据。

大端序系统

一个先存储最高有效字节的系统。另请参见字节顺序。

二进制分类

一种预测二进制结果（两个可能的类别之一）的过程。例如，您的 ML 模型可能需要预测诸如“该电子邮件是否为垃圾邮件？”或“这个产品是书还是汽车？”之类的问题

bloom 筛选条件

一种概率性、内存高效的数据结构，用于测试元素是否为集合的成员。

蓝/绿部署

一种部署策略，您可以创建两个独立但完全相同的环境。在一个环境中运行当前的应用程序版本（蓝色），在另一个环境中运行新的应用程序版本（绿色）。此策略可帮助您在影响最小的情况下快速回滚。

自动程序

一种通过互联网运行自动任务并模拟人类活动或互动的软件应用程序。有些机器人是有用或有益的，例如在互联网上索引信息的网络爬虫。其他一些被称为恶意机器人的机器人旨在破坏个人或组织或对其造成伤害。

僵尸网络

被恶意软件感染并受单方（称为机器人牧民或机器人操作员）控制的机器人网络。僵尸网络是最著名的扩展机器人及其影响力的机制。

分支

代码存储库的一个包含区域。在存储库中创建的第一个分支是主分支。您可以从现有分支创建新分支，然后在新分支中开发功能或修复错误。为构建功能而创建的分支通常称为功能分支。当功能可以发布时，将功能分支合并回主分支。有关更多信息，请参阅关于分支（GitHub 文档）。

破碎的玻璃通道

在特殊情况下，通过批准的流程，用户 AWS 账户 可以快速访问他们通常没有访问权限的内容。有关更多信息，请参阅 Well -Architected 指南中的 “实施破碎玻璃程序” 指示 AWS 器。

棕地策略

您环境中的现有基础设施。在为系统架构采用棕地策略时，您需要围绕当前系统和基础设施的限制来设计架构。如果您正在扩展现有基础设施，则可以将棕地策略和全新策略混合。

缓冲区缓存

存储最常访问的数据的内存区域。

业务能力

企业如何创造价值（例如，销售、客户服务或营销）。微服务架构和开发决策可以由业务能力驱动。有关更多信息，请参阅在 AWS上运行容器化微服务白皮书中的围绕业务能力进行组织部分。

业务连续性计划（BCP）

一项计划，旨在应对大规模迁移等破坏性事件对运营的潜在影响，并使企业能够快速恢复运营。

C

CAF

参见AWS 云采用框架。

金丝雀部署

向最终用户缓慢而渐进地发布版本。当你有信心时，你可以部署新版本并全部替换当前版本。

CCoE

参见云卓越中心。

CDC

参见变更数据捕获。

更改数据捕获（CDC）

跟踪数据来源（如数据库表）的更改并记录有关更改的元数据的过程。您可以将 CDC 用于各种目的，例如审计或复制目标系统中的更改以保持同步。

混沌工程

故意引入故障或破坏性事件来测试系统的弹性。您可以使用 AWS Fault Injection Service (AWS FIS) 来执行实验，对您的 AWS 工作负载施加压力并评估其响应。

CI/CD

查看持续集成和持续交付。

分类

一种有助于生成预测的分类流程。分类问题的 ML 模型预测离散值。离散值始终彼此不同。例如，一个模型可能需要评估图像中是否有汽车。

客户端加密

在目标 AWS 服务 收到数据之前，对数据进行本地加密。

云卓越中心（CCoE）

一个多学科团队，负责推动整个组织的云采用工作，包括开发云最佳实践、调动资源、制定迁移时间表、领导组织完成大规模转型。有关更多信息，请参阅 AWS Cloud 企业战略博客上的 CCoE 帖子。

云计算

通常用于远程数据存储和 IoT 设备管理的云技术。云计算通常与边缘计算技术相关。

云运营模型

在 IT 组织中，一种用于构建、完善和优化一个或多个云环境的运营模型。有关更多信息，请参阅构建您的云运营模型。

云采用阶段

组织迁移到以下阶段时通常会经历四个阶段 AWS Cloud：

- 项目 - 出于概念验证和学习目的，开展一些与云相关的项目
- 基础 - 进行基础投资以扩大云采用率（例如，创建登录区、定义 CCoE、建立运营模型）

- 迁移 - 迁移单个应用程序
- 重塑 - 优化产品和服务，在云中创新

Stephen Orban在 AWS Cloud 企业战略博客的博客文章《云优先之旅和采用阶段》中定义了这些阶段。有关它们与 AWS 迁移策略的关系的信息，请参阅迁移准备指南。

CMDB

参见配置管理数据库。

代码存储库

通过版本控制过程存储和更新源代码和其他资产（如文档、示例和脚本）的位置。常见的云存储库包括 GitHub 或 AWS CodeCommit。每个版本的代码都称为一个分支。在微服务结构中，每个存储库都专门用于一个功能。单个 CI/CD 管道可以使用多个存储库。

冷缓存

一种空的、填充不足或包含过时或不相关数据的缓冲区缓存。这会影响性能，因为数据库实例必须从主内存或磁盘读取，这比从缓冲区缓存读取要慢。

冷数据

很少访问的数据，且通常是历史数据。查询此类数据时，通常可以接受慢速查询。将这些数据转移到性能较低且成本更低的存储层或类别可以降低成本。

计算机视觉 (CV)

人工智能领域，使用机器学习来分析和提取数字图像和视频等视觉格式中的信息。例如，AWS Panorama 提供将 CV 添加到本地摄像机网络的设备，而 Amazon 则为 CV SageMaker 提供图像处理算法。

配置偏差

对于工作负载，配置会从预期状态发生变化。这可能会导致工作负载变得不合规，而且通常是渐进的，不是故意的。

配置管理数据库（CMDB）

一种存储库，用于存储和管理有关数据库及其 IT 环境的信息，包括硬件和软件组件及其配置。您通常在迁移的产品组合发现和分析阶段使用来自 CMDB 的数据。

合规性包

一系列 AWS Config 规则和补救措施，您可以汇编这些规则和补救措施，以自定义合规性和安全性检查。您可以使用 YAML 模板将一致性包作为单个实体部署在 AWS 账户 和区域或整个组织中。有关更多信息，请参阅 AWS Config 文档中的一致性包。

持续集成和持续交付（CI/CD）

自动执行软件发布过程的源代码、构建、测试、暂存和生产阶段的过程。CI/CD 通常被描述为管道。CI/CD 可以帮助您实现流程自动化、提高工作效率、改善代码质量并加快交付速度。有关更多信息，请参阅持续交付的优势。CD 也可以表示持续部署。有关更多信息，请参阅持续交付与持续部署。

CV

参见计算机视觉。

D

静态数据

网络中静止的数据，例如存储中的数据。

数据分类

根据网络中数据的关键性和敏感性对其进行识别和分类的过程。它是任何网络安全风险管理策略的关键组成部分，因为它可以帮助您确定对数据的适当保护和保留控制。数据分类是 Well-Architecte AWS d Framework 中安全支柱的一个组成部分。有关详细信息，请参阅数据分类。

数据漂移

生产数据与用来训练机器学习模型的数据之间的有意义差异，或者输入数据随时间推移的有意义变化。数据漂移可能降低机器学习模型预测的整体质量、准确性和公平性。

传输中数据

在网络中主动移动的数据，例如在网络资源之间移动的数据。

数据网格

一种架构框架，可提供分布式、去中心化的数据所有权以及集中式管理和治理。

数据最少化

仅收集并处理绝对必要数据的原则。在中进行数据最小化 AWS Cloud 可以降低隐私风险、成本和分析碳足迹。

数据边界

AWS 环境中的一组预防性防护措施，可帮助确保只有可信身份才能访问来自预期网络的可信资源。有关更多信息，请参阅在上构建数据边界。 AWS

数据预处理

将原始数据转换为 ML 模型易于解析的格式。预处理数据可能意味着删除某些列或行，并处理缺失、不一致或重复的值。

数据溯源

在数据的整个生命周期跟踪其来源和历史的过程，例如数据如何生成、传输和存储。

数据主体

正在收集和处理其数据的个人。

数据仓库

一种支持商业智能（例如分析）的数据管理系统。数据仓库通常包含大量历史数据，通常用于查询和分析。

数据库定义语言（DDL）

在数据库中创建或修改表和对象结构的语句或命令。

数据库操作语言（DML）

在数据库中修改（插入、更新和删除）信息的语句或命令。

DDL

参见数据库定义语言。

深度融合

组合多个深度学习模型进行预测。您可以使用深度融合来获得更准确的预测或估算预测中的不确定性。

深度学习

一个 ML 子字段使用多层人工神经网络来识别输入数据和感兴趣的目标变量之间的映射。

defense-in-depth

一种信息安全方法，经过深思熟虑，在整个计算机网络中分层实施一系列安全机制和控制措施，以保护网络及其中数据的机密性、完整性和可用性。当你采用这种策略时 AWS，你会在 AWS Organizations 结构的不同层面添加多个控件来帮助保护资源。例如，一种 defense-in-depth 方法可以结合多因素身份验证、网络分段和加密。

委托管理员

在中 AWS Organizations，兼容的服务可以注册 AWS 成员帐户来管理组织的帐户并管理该服务的权限。此账户被称为该服务的委托管理员。有关更多信息和兼容服务列表，请参阅 AWS Organizations 文档中使用 AWS Organizations的服务。

部署

使应用程序、新功能或代码修复在目标环境中可用的过程。部署涉及在代码库中实现更改，然后在应用程序的环境中构建和运行该代码库。

开发环境

参见环境。

侦测性控制

一种安全控制，在事件发生后进行检测、记录日志和发出警报。这些控制是第二道防线，提醒您注意绕过现有预防性控制的安全事件。有关更多信息，请参阅在 AWS上实施安全控制中的侦测性控制。

开发价值流映射 (DVSM)

用于识别对软件开发生命周期中的速度和质量产生不利影响的限制因素并确定其优先级的流程。DVSM 扩展了最初为精益生产实践设计的价值流映射流程。其重点关注在软件开发过程中创造和转移价值所需的步骤和团队。

数字孪生

真实世界系统的虚拟再现，如建筑物、工厂、工业设备或生产线。数字孪生支持预测性维护、远程监控和生产优化。

维度表

在星型架构中，一种较小的表，其中包含事实表中定量数据的数据属性。维度表属性通常是文本字段或行为类似于文本的离散数字。这些属性通常用于查询约束、筛选和结果集标注。

灾难

阻止工作负载或系统在其主要部署位置实现其业务目标的事件。这些事件可能是自然灾害、技术故障或人为操作的结果，例如无意的配置错误或恶意软件攻击。

灾难恢复 (DR)

您用来最大限度地减少灾难造成的停机时间和数据丢失的策略和流程。有关更多信息，请参阅 Well-Architected Fr ame AWS work 中的 “工作负载灾难恢复：云端 AWS 恢复”。

DML

参见数据库操作语言。

领域驱动设计

一种开发复杂软件系统的方法，通过将其组件连接到每个组件所服务的不断发展的领域或核心业务目标。Eric Evans 在其著作领域驱动设计：软件核心复杂性应对之道（Boston: Addison-Wesley Professional, 2003）中介绍了这一概念。有关如何将领域驱动设计与 strangler fig 模式结合使用的信息，请参阅使用容器和 Amazon API Gateway 逐步将原有的 Microsoft ASP.NET（ASMX）Web 服务现代化。

DR

参见灾难恢复。

漂移检测

跟踪与基准配置的偏差。例如，您可以使用 AWS CloudFormation 来检测系统资源中的偏差，也可以使用 AWS Control Tower 来检测着陆区中可能影响监管要求合规性的变化。

DVSM

参见开发价值流映射。

E

EDA

参见探索性数据分析。

边缘计算

该技术可提高位于 IoT 网络边缘的智能设备的计算能力。与云计算相比，边缘计算可以减少通信延迟并缩短响应时间。

加密

一种将人类可读的纯文本数据转换为密文的计算过程。

加密密钥

由加密算法生成的随机位的加密字符串。密钥的长度可能有所不同，而且每个密钥都设计为不可预测且唯一。

字节顺序

字节在计算机内存中的存储顺序。大端序系统先存储最高有效字节。小端序系统先存储最低有效字节。

端点

参见服务端点。

端点服务

一种可以在虚拟私有云（VPC）中托管，与其他用户共享的服务。您可以使用其他 AWS 账户 或 AWS Identity and Access Management (IAM) 委托人创建终端节点服务， AWS PrivateLink 并向其授予权限。这些账户或主体可通过创建接口 VPC 端点来私密地连接到您的端点服务。有关更多信息，请参阅 Amazon Virtual Private Cloud（Amazon VPC）文档中的创建端点服务。

企业资源规划 (ERP)

一种自动化和管理企业关键业务流程（例如会计、MES 和项目管理）的系统。

信封加密

用另一个加密密钥对加密密钥进行加密的过程。有关更多信息，请参阅 AWS Key Management Service (AWS KMS) 文档中的信封加密。

environment

正在运行的应用程序的实例。以下是云计算中常见的环境类型：

- 开发环境 — 正在运行的应用程序的实例，只有负责维护应用程序的核心团队才能使用。开发环境用于测试更改，然后再将其提升到上层环境。这类环境有时称为测试环境。
- 下层环境 — 应用程序的所有开发环境，比如用于初始构建和测试的环境。
- 生产环境 — 最终用户可以访问的正在运行的应用程序的实例。在 CI/CD 管道中，生产环境是最后一个部署环境。
- 上层环境 — 除核心开发团队以外的用户可以访问的所有环境。这可能包括生产环境、预生产环境和用户验收测试环境。

epic

在敏捷方法学中，有助于组织工作和确定优先级的功能类别。epics 提供了对需求和实施任务的总体描述。例如， AWS CAF 安全史诗包括身份和访问管理、侦探控制、基础设施安全、数据保护和事件响应。有关 AWS 迁移策略中 epics 的更多信息，请参阅计划实施指南。

ERP

参见企业资源规划。

探索性数据分析（EDA）

分析数据集以了解其主要特征的过程。您收集或汇总数据，并进行初步调查，以发现模式、检测异常并检查假定情况。EDA 通过计算汇总统计数据和创建数据可视化得以执行。

F

事实表

星形架构中的中心表。它存储有关业务运营的定量数据。通常，事实表包含两种类型的列：包含度量的列和包含维度表外键的列。

失败得很快

一种使用频繁和增量测试来缩短开发生命周期的理念。这是敏捷方法的关键部分。

故障隔离边界

在中 AWS Cloud，诸如可用区 AWS 区域、控制平面或数据平面之类的边界，它限制了故障的影响并有助于提高工作负载的弹性。有关更多信息，请参阅AWS 故障隔离边界。

功能分支

参见分支。

特征

您用来进行预测的输入数据。例如，在制造环境中，特征可能是定期从生产线捕获的图像。

特征重要性

特征对于模型预测的重要性。这通常表示为数值分数，可以通过各种技术进行计算，例如 Shapley 加法解释（SHAP）和积分梯度。有关更多信息，请参阅机器学习模型的可解释性：AWS。

功能转换

为 ML 流程优化数据，包括使用其他来源丰富数据、扩展值或从单个数据字段中提取多组信息。这使得 ML 模型能从数据中获益。例如，如果您将“2021-05-27 00:15:37”日期分解为“2021”、“五月”、“星期四”和“15”，则可以帮助学习与不同数据成分相关的算法学习精细模式。

FGAC

请参阅精细的访问控制。

精细访问控制 (FGAC)

使用多个条件允许或拒绝访问请求。

快闪迁移

一种数据库迁移方法，它使用连续的数据复制，通过更改数据捕获在尽可能短的时间内迁移数据，而不是使用分阶段的方法。目标是将停机时间降至最低。

G

地理封锁

请参阅地理限制。

地理限制（地理阻止）

在 Amazon 中 CloudFront，一种阻止特定国家/地区的用户访问内容分发的选项。您可以使用允许列表或阻止列表来指定已批准和已禁止的国家/地区。有关更多信息，请参阅 CloudFront 文档中的限制内容的地理分布。

GitFlow 工作流程

一种方法，在这种方法中，下层和上层环境在源代码存储库中使用不同的分支。Gitflow 工作流程被认为是传统的，而基于主干的工作流程是现代的首选方法。

全新策略

在新环境中缺少现有基础设施。在对系统架构采用全新策略时，您可以选择所有新技术，而不受对现有基础设施（也称为棕地）兼容性的限制。如果您正在扩展现有基础设施，则可以将棕地策略和全新策略混合。

防护机制

一种高级规则，用于跨组织单位（OU）管理资源、策略和合规性。预防性防护机制会执行策略以确保符合合规性标准。它们是使用服务控制策略和 IAM 权限边界实现的。侦测性防护机制会检测策略违规和合规性问题，并生成警报以进行修复。它们通过使用 AWS Config、、Amazon、 AWS Security Hub GuardDuty AWS Trusted Advisor、Amazon Inspector 和自定义 AWS Lambda 支票来实现。

H

HA

参见高可用性。

异构数据库迁移

将源数据库迁移到使用不同数据库引擎的目标数据库（例如，从 Oracle 迁移到 Amazon Aurora）。异构迁移通常是重新架构工作的一部分，而转换架构可能是一项复杂的任务。AWS 提供了 AWS SCT 来帮助实现架构转换。

高可用性 (HA)

在遇到挑战或灾难时，工作负载无需干预即可连续运行的能力。HA 系统旨在自动进行故障转移、持续提供良好性能，并以最小的性能影响处理不同负载和故障。

历史数据库现代化

一种用于实现运营技术 (OT) 系统现代化和升级以更好满足制造业需求的方法。历史数据库是一种用于收集和存储工厂中各种来源数据的数据库。

同构数据库迁移

将源数据库迁移到共享同一数据库引擎的目标数据库（例如，从 Microsoft SQL Server 迁移到 Amazon RDS for SQL Server）。同构迁移通常是更换主机或更换平台工作的一部分。您可以使用本机数据库实用程序来迁移架构。

热数据

经常访问的数据，例如实时数据或近期的转化数据。这些数据通常需要高性能存储层或存储类别才能提供快速的查询响应。

修补程序

针对生产环境中关键问题的紧急修复。由于其紧迫性，修补程序通常是在典型的 DevOps 发布工作流程之外进行的。

hypercare 周期

割接之后，迁移团队立即管理和监控云中迁移的应用程序以解决任何问题的时间段。通常，这个周期持续 1-4 天。在 hypercare 周期结束时，迁移团队通常会将应用程序的责任移交给云运营团队。

I

IaC

参见基础架构即代码。

基于身份的策略

附加到一个或多个 IAM 委托人的策略，用于定义他们在 AWS Cloud 环境中的权限。

空闲应用程序

90 天内平均 CPU 和内存使用率在 5% 到 20% 之间的应用程序。在迁移项目中，通常会停用这些应用程序或将其保留在本地。

IIoT

参见工业物联网。

不可变的基础架构

一种为生产工作负载部署新基础架构，而不是更新、修补或修改现有基础架构的模型。不可变基础架构本质上比可变基础架构更一致、更可靠、更可预测。有关更多信息，请参阅 Well-Architected Framework 中的使用不可变基础架构 AWS 部署最佳实践。

入站（入口）VPC

在 AWS 多账户架构中，一种接受、检查和路由来自应用程序外部的网络连接的 VPC。AWS 安全参考架构建议使用入站、出站和检查 VPC 设置网络账户，保护应用程序与广泛的互联网之间的双向接口。

增量迁移

一种割接策略，在这种策略中，您可以将应用程序分成小部分进行迁移，而不是一次性完整割接。例如，您最初可能只将几个微服务或用户迁移到新系统。在确认一切正常后，您可以逐步迁移其他微服务或用户，直到停用遗留系统。这种策略降低了大规模迁移带来的风险。

工业 4.0

该术语由克劳斯·施瓦布（Klaus Schwab）于2016年推出，指的是通过连接、实时数据、自动化、分析和人工智能/机器学习的进步实现制造流程的现代化。

基础设施

应用程序环境中包含的所有资源和资产。

基础设施即代码（IaC）

通过一组配置文件预置和管理应用程序基础设施的过程。IaC 旨在帮助您集中管理基础设施、实现资源标准化和快速扩展，使新环境具有可重复性、可靠性和一致性。

工业物联网（IIoT）

在工业领域使用联网的传感器和设备，例如制造业、能源、汽车、医疗保健、生命科学和农业。有关更多信息，请参阅制定工业物联网（IIoT）数字化转型策略。

检查 VPC

在 AWS 多账户架构中，一种集中式 VPC，用于管理 VPC（相同或不同 AWS 区域）、互联网和本地网络之间的网络流量检查。AWS 安全参考架构建议使用入站、出站和检查 VPC 设置网络账户，保护应用程序与广泛的互联网之间的双向接口。

物联网（IoT）

由带有嵌入式传感器或处理器的连接物理对象组成的网络，这些传感器或处理器通过互联网或本地通信网络与其他设备和系统进行通信。有关更多信息，请参阅什么是 IoT？

可解释性

它是机器学习模型的一种特征，描述了人类可以理解模型的预测如何取决于其输入的程度。有关更多信息，请参阅使用 AWS 实现机器学习模型的可解释性。

IoT

参见物联网。

IT 信息库（ITIL）

提供 IT 服务并使这些服务符合业务要求的一套最佳实践。ITIL 是 ITSM 的基础。

IT 服务管理（ITSM）

为组织设计、实施、管理和支持 IT 服务的相关活动。有关将云运营与 ITSM 工具集成的信息，请参阅运营集成指南。

ITIL

请参阅 IT 信息库。

ITSM

请参阅 IT 服务管理。

L

基于标签的访问控制 (LBAC)

强制访问控制 (MAC) 的一种实施方式，其中明确为用户和数据本身分配了安全标签值。用户安全标签和数据安全标签之间的交集决定了用户可以看到哪些行和列。

登录区

landing zone 是一个架构精良的多账户 AWS 环境，具有可扩展性和安全性。这是一个起点，您的组织可以从这里放心地在安全和基础设施环境中快速启动和部署工作负载和应用程序。有关登录区的更多信息，请参阅设置安全且可扩展的多账户 AWS 环境。

大规模迁移

迁移 300 台或更多服务器。

LBAC

参见基于标签的访问控制。

最低权限

授予执行任务所需的最低权限的最佳安全实践。有关更多信息，请参阅 IAM 文档中的应用最低权限许可。

直接迁移

见 7 R。

小端序系统

一个先存储最低有效字节的系统。另请参见字节顺序。

下层环境

参见环境。

M

机器学习（ML）

一种使用算法和技术进行模式识别和学习的人工智能。ML 对记录的数据（例如物联网（IoT）数据）进行分析和学习，以生成基于模式的统计模型。有关更多信息，请参阅机器学习。

主分支

参见分支。

恶意软件

旨在危害计算机安全或隐私的软件。恶意软件可能会破坏计算机系统、泄露敏感信息或获得未经授权的访问。恶意软件的示例包括病毒、蠕虫、勒索软件、特洛伊木马、间谍软件和键盘记录器。

托管服务

AWS 服务 它 AWS 运行基础设施层、操作系统和平台，您可以访问端点来存储和检索数据。亚马逊简单存储服务 (Amazon S3) Service 和 Amazon DynamoDB 就是托管服务的示例。这些服务也称为抽象服务。

制造执行系统 (MES)

一种软件系统，用于跟踪、监控、记录和控制在车间将原材料转化为成品的生产过程。

MAP

参见迁移加速计划。

机制

一个完整的过程，在此过程中，您可以创建工具，推动工具的采用，然后检查结果以进行调整。机制是一种在运行过程中自我增强和改进的循环。有关更多信息，请参阅在 Well-Architect AWS ed 框架中构建机制。

成员账户

AWS 账户 除属于组织中的管理账户之外的所有账户 AWS Organizations。一个账户一次只能是一个组织的成员。

MES

参见制造执行系统。

消息队列遥测传输 (MQTT)

一种基于发布/订阅模式的轻量级 machine-to-machine (M2M) 通信协议，适用于资源受限的物联网设备。

微服务

一种小型独立服务，通过明确定义的 API 进行通信，通常由小型独立团队拥有。例如，保险系统可能包括映射到业务能力（如销售或营销）或子域（如购买、理赔或分析）的微服务。微服务的好处包括敏捷、灵活扩展、易于部署、可重复使用的代码和恢复能力。有关更多信息，请参阅使用 AWS 无服务器服务集成微服务。

微服务架构

一种使用独立组件构建应用程序的方法，这些组件将每个应用程序进程作为微服务运行。这些微服务使用轻量级 API 通过明确定义的接口进行通信。该架构中的每个微服务都可以更新、部署和扩展，以满足对应用程序特定功能的需求。有关更多信息，请参阅在上实现微服务。 AWS

迁移加速计划（MAP）

AWS 该计划提供咨询支持、培训和服务，以帮助组织为迁移到云奠定坚实的运营基础，并帮助抵消迁移的初始成本。MAP 提供了一种以系统的方式执行遗留迁移的迁移方法，以及一套用于自动执行和加速常见迁移场景的工具。

大规模迁移

将大部分应用程序组合分波迁移到云中的过程，在每一波中以更快的速度迁移更多应用程序。本阶段使用从早期阶段获得的最佳实践和经验教训，实施由团队、工具和流程组成的迁移工厂，通过自动化和敏捷交付简化工作负载的迁移。这是 AWS 迁移策略的第三阶段。

迁移工厂

跨职能团队，通过自动化、敏捷的方法简化工作负载迁移。迁移工厂团队通常包括运营、业务分析师和所有者、迁移工程师、开发 DevOps 人员和冲刺专业人员。20% 到 50% 的企业应用程序组合由可通过工厂方法优化的重复模式组成。有关更多信息，请参阅本内容集中有关迁移工厂的讨论和云迁移工厂指南。

迁移元数据

有关完成迁移所需的应用程序和服务器的信息。每种迁移模式都需要一套不同的迁移元数据。迁移元数据的示例包括目标子网、安全组和 AWS 账户。

迁移模式

一种可重复的迁移任务，详细列出了迁移策略、迁移目标以及所使用的迁移应用程序或服务。示例：使用 AWS 应用程序迁移服务重新托管向 Amazon EC2 的迁移。

迁移组合评测（MPA）

一种在线工具，可提供信息，用于验证迁移到的业务案例。 AWS Cloud MPA 提供了详细的组合评测（服务器规模调整、定价、TCO 比较、迁移成本分析）以及迁移计划（应用程序数据分析和数据收集、应用程序分组、迁移优先级排序和波次规划）。所有 AWS 顾问和 APN 合作伙伴顾问均可免费使用 MPA 工具（需要登录）。

迁移准备情况评测（MRA）

使用 AWS CAF 深入了解组织的云就绪状态、确定优势和劣势以及制定行动计划以缩小已发现差距的过程。有关更多信息，请参阅迁移准备指南。MRA 是 AWS 迁移策略的第一阶段。

迁移策略

用于将工作负载迁移到的方法 AWS Cloud。有关更多信息，请参阅此词汇表中的 7 R 条目和动员组织以加快大规模迁移。

ML

参见机器学习。

现代化

将过时的（原有的或单体）应用程序及其基础设施转变为云中敏捷、弹性和高度可用的系统，以降低成本、提高效率和利用创新。有关更多信息，请参阅中的应用程序现代化策略。 AWS Cloud

现代化准备情况评估

一种评估方式，有助于确定组织应用程序的现代化准备情况；确定收益、风险和依赖关系；确定组织能够在多大程度上支持这些应用程序的未来状态。评估结果是目标架构的蓝图、详细说明现代化进程发展阶段和里程碑的路线图以及解决已发现差距的行动计划。有关更多信息，请参阅中的评估应用程序的现代化准备情况 AWS Cloud。

单体应用程序（单体式）

作为具有紧密耦合进程的单个服务运行的应用程序。单体应用程序有几个缺点。如果某个应用程序功能的需求激增，则必须扩展整个架构。随着代码库的增长，添加或改进单体应用程序的功能也会变得更加复杂。若要解决这些问题，可以使用微服务架构。有关更多信息，请参阅将单体分解为微服务。

MPA

参见迁移组合评估。

MQTT

请参阅消息队列遥测传输。

多分类器

一种帮助为多个类别生成预测（预测两个以上结果之一）的过程。例如，ML 模型可能会询问“这个产品是书、汽车还是手机？”或“此客户最感兴趣什么类别的产品？”

可变基础架构

一种用于更新和修改现有生产工作负载基础架构的模型。为了提高一致性、可靠性和可预测性，Well-Architect AWS ed Framework 建议使用不可变基础设施作为最佳实践。

O

OAC

请参阅源站访问控制。

OAI

参见源访问身份。

OCM

参见组织变更管理。

离线迁移

一种迁移方法，在这种方法中，源工作负载会在迁移过程中停止运行。这种方法会延长停机时间，通常用于小型非关键工作负载。

OI

参见运营集成。

OLA

参见运营层协议。

在线迁移

一种迁移方法，在这种方法中，源工作负载无需离线即可复制到目标系统。在迁移过程中，连接工作负载的应用程序可以继续运行。这种方法的停机时间为零或最短，通常用于关键生产工作负载。

OPC-UA

参见开放流程通信-统一架构。

开放流程通信-统一架构 (OPC-UA)

一种用于工业自动化的 machine-to-machine （M2M）通信协议。OPC-UA 提供了数据加密、身份验证和授权方案的互操作性标准。

运营级别协议（OLA）

一项协议，阐明了 IT 职能部门承诺相互交付的内容，以支持服务水平协议（SLA）。

运营准备情况审查 (ORR)

一份问题清单和相关的最佳实践，可帮助您理解、评估、预防或缩小事件和可能的故障的范围。有关更多信息，请参阅 Well-Architecte AWS d Frame work 中的运营准备情况评估 (ORR)。

操作技术 (OT)

与物理环境配合使用以控制工业运营、设备和基础设施的硬件和软件系统。在制造业中，OT 和信息技术 (IT) 系统的集成是工业 4.0 转型的重点。

运营整合（OI）

在云中实现运营现代化的过程，包括就绪计划、自动化和集成。有关更多信息，请参阅运营整合指南。

组织跟踪

由 AWS CloudTrail 此创建的跟踪记录组织 AWS 账户 中所有人的所有事件 AWS Organizations。该跟踪是在每个 AWS 账户 中创建的，属于组织的一部分，并跟踪每个账户的活动。有关更多信息，请参阅 CloudTrail文档中的为组织创建跟踪。

组织变革管理（OCM）

一个从人员、文化和领导力角度管理重大、颠覆性业务转型的框架。OCM 通过加快变革采用、解决过渡问题以及推动文化和组织变革，帮助组织为新系统和战略做好准备和过渡。在 AWS 迁移策略中，该框架被称为人员加速，因为云采用项目需要变更的速度。有关更多信息，请参阅 OCM 指南。

来源访问控制（OAC）

在中 CloudFront，一个增强的选项，用于限制访问以保护您的亚马逊简单存储服务 (Amazon S3) 内容。OAC 全部支持所有 S3 存储桶 AWS 区域、使用 AWS KMS (SSE-KMS) 进行服务器端加密，以及对 S3 存储桶的动态PUT和DELETE请求。

来源访问身份（OAI）

在中 CloudFront，一个用于限制访问权限以保护您的 Amazon S3 内容的选项。当您使用 OAI 时，CloudFront 会创建一个 Amazon S3 可以对其进行身份验证的委托人。经过身份验证的委托人只能通过特定 CloudFront 分配访问 S3 存储桶中的内容。另请参阅 OAC，其中提供了更精细和增强的访问控制。

或者

参见运营准备情况审查。

OT

参见运营技术。

出站（出口）VPC

在 AWS 多账户架构中，一种处理从应用程序内部启动的网络连接的 VPC。AWS 安全参考架构建议使用入站、出站和检查 VPC 设置网络账户，保护应用程序与广泛的互联网之间的双向接口。

P

权限边界

附加到 IAM 主体的 IAM 管理策略，用于设置用户或角色可以拥有的最大权限。有关更多信息，请参阅 IAM 文档中的权限边界。

个人身份信息 (PII)

直接查看其他相关数据或与之配对时可用于合理推断个人身份的信息。PII 的示例包括姓名、地址和联系信息。

PII

查看个人身份信息。

playbook

一套预定义的步骤，用于捕获与迁移相关的工作，例如在云中交付核心运营功能。playbook 可以采用脚本、自动化运行手册的形式，也可以是操作现代化环境所需的流程或步骤的摘要。

PLC

参见可编程逻辑控制器。

PLM

参见产品生命周期管理。

策略

一个对象，可以在中定义权限（参见基于身份的策略）、指定访问条件（参见基于资源的策略）或定义组织中所有账户的最大权限 AWS Organizations （参见服务控制策略）。

多语言持久性

根据数据访问模式和其他要求，独立选择微服务的数据存储技术。如果您的微服务采用相同的数据存储技术，它们可能会遇到实现难题或性能不佳。如果微服务使用最适合其需求的数据存储，则可以更轻松地实现微服务，并获得更好的性能和可扩展性。有关更多信息，请参阅在微服务中实现数据持久性。

组合评测

一个发现、分析和确定应用程序组合优先级以规划迁移的过程。有关更多信息，请参阅评估迁移准备情况。

谓词

返回`true`或的查询条件`false`，通常位于子`WHERE`句中。

谓词下推

一种数据库查询优化技术，可在传输前筛选查询中的数据。这减少了必须从关系数据库检索和处理的数据量，并提高了查询性能。

预防性控制

一种安全控制，旨在防止事件发生。这些控制是第一道防线，帮助防止未经授权的访问或对网络的意外更改。有关更多信息，请参阅在 AWS上实施安全控制中的预防性控制。

主体

中 AWS 可以执行操作和访问资源的实体。此实体通常是 IAM 角色的根用户或用户。 AWS 账户有关更多信息，请参阅 IAM 文档中角色术语和概念中的主体。

隐私设计

一种贯穿整个工程化过程考虑隐私的系统工程方法。

私有托管区

私有托管区就是一个容器，其中包含的信息说明您希望 Amazon Route 53 如何响应一个或多个 VPC 中的某个域及其子域的 DNS 查询。有关更多信息，请参阅 Route 53 文档中的私有托管区的使用。

主动控制

一种安全控制措施，旨在防止部署不合规的资源。这些控件会在资源置备之前对其进行扫描。如果资源与控件不兼容，则不会对其进行配置。有关更多信息，请参阅 AWS Control Tower 文档中的控制参考指南，并参见在上实施安全控制中的主动控制 AWS。

产品生命周期管理 (PLM)

在产品的整个生命周期中，从设计、开发和上市，到成长和成熟，再到衰落和移除，对产品进行数据和流程的管理。

生产环境

参见环境。

可编程逻辑控制器 (PLC)

在制造业中，一种高度可靠、适应性强的计算机，用于监控机器并实现制造过程自动化。

假名化

用占位符值替换数据集中个人标识符的过程。假名化可以帮助保护个人隐私。假名化数据仍被视为个人数据。

发布/订阅（发布/订阅）

一种支持微服务间异步通信的模式，以提高可扩展性和响应能力。例如，在基于微服务的 MES 中，微服务可以将事件消息发布到其他微服务可以订阅的频道。系统可以在不更改发布服务的情况下添加新的微服务。

Q

查询计划

一系列步骤，例如指令，用于访问 SQL 关系数据库系统中的数据。

查询计划回归

当数据库服务优化程序选择的最佳计划不如数据库环境发生特定变化之前时。这可能是由统计数据、约束、环境设置、查询参数绑定更改和数据库引擎更新造成的。

R

RACI 矩阵

参见 “负责任、负责、咨询、知情”（RACI）。

勒索软件

一种恶意软件，旨在阻止对计算机系统或数据的访问，直到付款为止。

RASCI 矩阵

参见 “负责任、负责、咨询、知情”（RACI）。

RCAC

请参阅行和列访问控制。

只读副本

用于只读目的的数据库副本。您可以将查询路由到只读副本，以减轻主数据库的负载。

重新架构师

见 7 R。

恢复点目标 (RPO)

自上一个数据恢复点以来可接受的最长时间。这决定了从上一个恢复点到服务中断之间可接受的数据丢失情况。

恢复时间目标 (RTO)

服务中断和服务恢复之间可接受的最大延迟。

重构

见 7 R。

区域

地理区域内的 AWS 资源集合。每一个 AWS 区域 都相互隔离，彼此独立，以提供容错、稳定性和弹性。有关更多信息，请参阅指定 AWS 区域 您的账户可以使用的账户。

回归

一种预测数值的 ML 技术。例如，要解决“这套房子的售价是多少？”的问题 ML 模型可以使用线性回归模型，根据房屋的已知事实（如建筑面积）来预测房屋的销售价格。

重新托管

见 7 R。

版本

在部署过程中，推动生产环境变更的行为。

搬迁

见 7 R。

更换平台

见 7 R。

回购

见 7 R。

故障恢复能力

应用程序抵御中断或从中断中恢复的能力。在中规划弹性时，高可用性和灾难恢复是常见的考虑因素。 AWS Cloud有关更多信息，请参阅AWS Cloud 弹性。

基于资源的策略

一种附加到资源的策略，例如 AmazonS3 存储桶、端点或加密密钥。此类策略指定了允许哪些主体访问、支持的操作以及必须满足的任何其他条件。

责任、问责、咨询和知情（RACI）矩阵

定义参与迁移活动和云运营的所有各方的角色和责任的矩阵。矩阵名称源自矩阵中定义的责任类型：负责 (R)、问责 (A)、咨询 (C) 和知情 (I)。支持 (S) 类型是可选的。如果包括支持，则该矩阵称为 RASCI 矩阵，如果将其排除在外，则称为 RACI 矩阵。

响应性控制

一种安全控制，旨在推动对不良事件或偏离安全基线的情况进行修复。有关更多信息，请参阅在AWS上实施安全控制中的响应性控制。

保留

见 7 R。

退休

见 7 R。

旋转

定期更新密钥以使攻击者更难访问凭据的过程。

行列访问控制 (RCAC)

使用已定义访问规则的基本、灵活的 SQL 表达式。RCAC 由行权限和列掩码组成。

RPO

参见恢复点目标。

RTO

参见恢复时间目标。

运行手册

执行特定任务所需的一套手动或自动程序。它们通常是为了简化重复性操作或高错误率的程序而设计的。

S

SAML 2.0

许多身份提供商 (IdPs) 使用的开放标准。此功能支持联合单点登录 (SSO)，因此用户无需在 IAM 中为组织中的所有人创建用户即可登录 AWS Management Console 或调用 AWS API 操作。有关基于 SAML 2.0 的联合身份验证的更多信息，请参阅 IAM 文档中的关于基于 SAML 2.0 的联合身份验证。

SCADA

参见监督控制和数据采集。

SCP

参见服务控制政策。

secret

在中 AWS Secrets Manager，您以加密形式存储的机密或受限信息，例如密码或用户凭证。它由密钥值及其元数据组成。密钥值可以是二进制、单个字符串或多个字符串。有关更多信息，请参阅 Secret s Manager 密钥中有什么？ 在 Secrets Manager 文档中。

安全控制

一种技术或管理防护机制，可防止、检测或降低威胁行为体利用安全漏洞的能力。安全控制主要有四种类型：预防性、侦测、响应式和主动式。

安全加固

缩小攻击面，使其更能抵御攻击的过程。这可能包括删除不再需要的资源、实施授予最低权限的最佳安全实践或停用配置文件中不必要的功能等操作。

安全信息和事件管理（SIEM）系统

结合了安全信息管理（SIM）和安全事件管理（SEM）系统的工具和服务。SIEM 系统会收集、监控和分析来自服务器、网络、设备和其他来源的数据，以检测威胁和安全漏洞，并生成警报。

安全响应自动化

一种预定义和编程的操作，旨在自动响应或修复安全事件。这些自动化可作为侦探或响应式安全控制措施，帮助您实施 AWS 安全最佳实践。自动响应操作的示例包括修改 VPC 安全组、修补 Amazon EC2 实例或轮换证书。

服务器端加密

在目的地对数据进行加密，由接收数据 AWS 服务 的人加密。

服务控制策略（SCP）

一种策略，用于集中控制 AWS Organizations的组织中所有账户的权限。SCP 为管理员可以委托给用户或角色的操作定义了防护机制或设定了限制。您可以将 SCP 用作允许列表或拒绝列表，指定允许或禁止哪些服务或操作。有关更多信息，请参阅 AWS Organizations 文档中的服务控制策略。

服务端点

的入口点的 URL AWS 服务。您可以使用端点，通过编程方式连接到目标服务。有关更多信息，请参阅 AWS 一般参考 中的 AWS 服务 端点。

服务水平协议（SLA）

一份协议，阐明了 IT 团队承诺向客户交付的内容，比如服务正常运行时间和性能。

服务级别指示器 (SLI)

对服务性能方面的衡量，例如其错误率、可用性或吞吐量。

服务级别目标 (SLO)

代表服务运行状况的目标指标，由服务级别指标衡量。

责任共担模式

描述您在云安全与合规方面共同承担 AWS 的责任的模型。 AWS 负责云的安全，而您则负责云中的安全。有关更多信息，请参阅责任共担模式。

暹粒

参见安全信息和事件管理系统。

单点故障 (SPOF)

应用程序的单个关键组件出现故障，可能会中断系统。

SLA

参见服务级别协议。

SLI

参见服务级别指标。

SLO

参见服务级别目标。

split-and-seed 模型

一种扩展和加速现代化项目的模式。随着新功能和产品发布的定义，核心团队会拆分以创建新的产品团队。这有助于扩展组织的能力和服务，提高开发人员的工作效率，支持快速创新。有关更多信息，请参阅中的分阶段实现应用程序现代化的方法。 AWS Cloud

恶作剧

参见单点故障。

星型架构

一种数据库组织结构，它使用一个大型事实表来存储交易数据或测量数据，并使用一个或多个较小的维度表来存储数据属性。此结构专为在数据仓库中使用或用于商业智能目的而设计。

strangler fig 模式

一种通过逐步重写和替换系统功能直至可以停用原有的系统来实现单体系统现代化的方法。这种模式用无花果藤作为类比，这种藤蔓成长为一棵树，最终战胜并取代了宿主。该模式是由 Martin Fowler 提出的，作为重写单体系统时管理风险的一种方法。有关如何应用此模式的示例，请参阅使用容器和 Amazon API Gateway 逐步将原有的 Microsoft ASP.NET（ASMX）Web 服务现代化。

子网

您的 VPC 内的一个 IP 地址范围。子网必须位于单个可用区中。

监控和数据采集 (SCADA)

在制造业中，一种使用硬件和软件来监控有形资产和生产操作的系统。

对称加密

一种加密算法，它使用相同的密钥来加密和解密数据。

综合测试

以模拟用户交互的方式测试系统，以检测潜在问题或监控性能。你可以使用 Amazon S CloudWatch ynthetics 来创建这些测试。

T

标签

键值对，充当用于组织资源的元数据。 AWS 标签可帮助您管理、识别、组织、搜索和筛选资源。有关更多信息，请参阅标记您的 AWS 资源。

目标变量

您在监督式 ML 中尝试预测的值。这也被称为结果变量。例如，在制造环境中，目标变量可能是产品缺陷。

任务列表

一种通过运行手册用于跟踪进度的工具。任务列表包含运行手册的概述和要完成的常规任务列表。对于每项常规任务，它包括预计所需时间、所有者和进度。

测试环境

参见环境。

训练

为您的 ML 模型提供学习数据。训练数据必须包含正确答案。学习算法在训练数据中查找将输入数据属性映射到目标（您希望预测的答案）的模式。然后输出捕获这些模式的 ML 模型。然后，您可以使用 ML 模型对不知道目标的新数据进行预测。

中转网关

中转网关是网络中转中心，您可用它来互连 VPC 和本地网络。有关更多信息，请参阅 AWS Transit Gateway 文档中的什么是公交网关。

基于中继的工作流程

一种方法，开发人员在功能分支中本地构建和测试功能，然后将这些更改合并到主分支中。然后，按顺序将主分支构建到开发、预生产和生产环境。

可信访问权限

向您指定的服务授予权限，该服务可以代表您在其账户中执行任务。 AWS Organizations 当需要服务相关的角色时，受信任的服务会在每个账户中创建一个角色，为您执行管理任务。有关更多信息，请参阅 AWS Organizations 文档中的AWS Organizations 与其他 AWS 服务一起使用。

优化

更改训练过程的各个方面，以提高 ML 模型的准确性。例如，您可以通过生成标签集、添加标签，并在不同的设置下多次重复这些步骤来优化模型，从而训练 ML 模型。

双披萨团队

一个小 DevOps 团队，你可以用两个披萨来喂食。双披萨团队的规模可确保在软件开发过程中充分协作。

U

不确定性

这一概念指的是不精确、不完整或未知的信息，这些信息可能会破坏预测式 ML 模型的可靠性。不确定性有两种类型：认知不确定性是由有限的、不完整的数据造成的，而偶然不确定性是由数据中固有的噪声和随机性导致的。有关更多信息，请参阅量化深度学习系统中的不确定性指南。

无差别任务

也称为繁重工作，即创建和运行应用程序所必需的工作，但不能为最终用户提供直接价值或竞争优势。无差别任务的示例包括采购、维护和容量规划。

上层环境

参见环境。

V

vacuum 操作

一种数据库维护操作，包括在增量更新后进行清理，以回收存储空间并提高性能。

版本控制

跟踪更改的过程和工具，例如存储库中源代码的更改。

VPC 对等连接

两个 VPC 之间的连接，允许您使用私有 IP 地址路由流量。有关更多信息，请参阅 Amazon VPC 文档中的什么是 VPC 对等连接。

漏洞

损害系统安全的软件缺陷或硬件缺陷。

W

热缓存

一种包含经常访问的当前相关数据的缓冲区缓存。数据库实例可以从缓冲区缓存读取，这比从主内存或磁盘读取要快。

暖数据

不常访问的数据。查询此类数据时，通常可以接受中速查询。

窗口函数

一个 SQL 函数，用于对一组以某种方式与当前记录相关的行进行计算。窗口函数对于处理任务很有用，例如计算移动平均线或根据当前行的相对位置访问行的值。

工作负载

一系列资源和代码，它们可以提供商业价值，如面向客户的应用程序或后端过程。

工作流

迁移项目中负责一组特定任务的职能小组。每个工作流都是独立的，但支持项目中的其他工作流。例如，组合工作流负责确定应用程序的优先级、波次规划和收集迁移元数据。组合工作流将这些资产交付给迁移工作流，然后迁移服务器和应用程序。

蠕虫

参见一次写入，多读。

WQF

请参阅 AWS 工作负载资格框架。

一次写入，多次读取 (WORM)

一种存储模型，它可以一次写入数据并防止数据被删除或修改。授权用户可以根据需要多次读取数据，但他们无法对其进行更改。这种数据存储基础架构被认为是不可变的。

Z

零日漏洞利用

一种利用未修补漏洞的攻击，通常是恶意软件。

零日漏洞

生产系统中不可避免的缺陷或漏洞。威胁主体可能利用这种类型的漏洞攻击系统。开发人员经常因攻击而意识到该漏洞。

僵尸应用程序

平均 CPU 和内存使用率低于 5% 的应用程序。在迁移项目中，通常会停用这些应用程序。

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。