aws

多租戶 SaaS 授權和 API 存取控制：實作選項和最佳做法

AWS 規定指引

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS 規定指引: 多租戶 SaaS 授權和 API 存取控制：實作選項和最佳做法

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商標和商業外觀不得用於任何非 Amazon 的產品或服務，也不能以任何可能造成客戶混淆、任何貶低或使 Amazon 名譽受損的方式使用 Amazon 的商標和商業外觀。所有其他非 Amazon 擁有的商標均為其各自擁有者的財產，這些擁有者可能隸屬於 Amazon，或與 Amazon 有合作關係，或由 Amazon 贊助。

Table of Contents

多租戶 SaaS 授權和 API 存取控制：實作選項和最佳做法

虎斑病房，托馬斯·戴維斯，基定蘭德曼和托馬斯·里哈，Amazon Web Services（ ）AWS

2024 年 5 月 (文件歷史記錄)

授權和 API 存取控制是許多軟體應用程式的挑戰，尤其是對於多租戶軟體即服務 (SaaS) 應用程式而言。當您考慮到必須保護的微服務 API 擴散，以及來自不同租用戶、使用者特性和應用程式狀態的大量存取條件時，這種複雜性就很明顯。為了有效解決這些問題，解決方案必須對微服務、前端後端 (BFF) 層以及多租戶 SaaS 應用程式的其他元件所提供的許多 API 強制執行存取控制。這種方法必須伴隨著一種機制，該機制能夠根據許多因素和屬性做出複雜的訪問決策。

傳統上，API 訪問控制和授權是由應用程序代碼中的自定義邏輯處理的。這種方法容易出錯且不安全，因為可以訪問此代碼的開發人員可能會意外或故意更改授權邏輯，從而導致未經授權的訪問。稽核應用程式程式碼中的自訂邏輯所做的決定很困難，因為稽核人員必須將自己沉浸在自訂邏輯中，以判斷其在維護任何特定標準方面的有效性。此外，API 存取控制通常是不必要的，因為沒有那麼多的 API 來保護。應用程式設計的範式轉變為支援微服務和服務導向架構，增加了必須使用授權和存取控制形式的 API 數量。此外，在多租戶 SaaS 應用程式中維護以租戶為基礎的存取權限的需求會帶來額外的授權挑戰，以保留租用。本指南中概述的最佳做法提供了幾個好處：

- 授權邏輯可以集中化，並以不特定於任何程式設計語言的高階宣告式語言撰寫。
- 授權邏輯是從應用程式程式碼中抽取出來的，並可作為可重複模式套用至應用程式中的所有 API。
- 抽象可防止開發人員意外更改授權邏輯。
- 集成到 SaaS 應用程序是一致且簡單的。
- 抽象可防止需要為每個 API 端點編寫自定義授權邏輯。
- 稽核作業簡化，因為稽核人員不再需要檢閱程式碼來判斷權限。
- 本指南中概述的方法支援使用多重存取控制範例，視組織的需求而定。
- 這種授權和存取控制方法提供了一種簡單而直接的方法，可在 SaaS 應用程式中的 API 層維護租用戶資料隔離。
- 在授權方面，最佳做法可提供一致的方法來上線和離職租用戶。
- 這種方法提供了不同的授權部署模型（集區或筒倉），它們具有優點和缺點，如本指南中所述。

目標業務成果

此規範指引說明可重複的授權設計模式，以及可針對多租用戶 SaaS 應用程式實作的 API 存取控制。本指引適用於開發具有複雜授權需求或嚴格 API 存取控制需求之應用程式的任何團隊。架構詳細說明原則決策點 (PDP) 或原則引擎的建立，以及原則強制執行點 (PEP) 在 API 中的整合。討論建立 PDP 的兩個特定選項：將 Amazon 驗證許可與 Cedar SDK 搭配使用，以及使用開放原則代理程式 (OPA) 搭配 Rego 政策語言。本指南也討論根據屬性型存取控制 (ABAC) 模型或角色型存取控制 (RBAC) 模型，或兩種模型的組合，來做出存取決策。我們建議您使用本指南中提供的設計模式和概念，告知並標準化您在多租戶 SaaS 應用程式中實作授權和 API 存取控制的實作。此指引有助於實現以下業務成果：

- 多租戶 SaaS 應用程式的標準化 API 授權架構 — 此架構可區分三個元件：儲存和管理原則的原則管理點 (PAP)、評估這些原則以達成授權決策的原則決策點 (PDP)，以及強制執行該決策的原則強制執行點 (PEP)。託管的授權服務，驗證的權限，作為 PAP 和 PDP。或者，您也可以使用 Cedar 或 OPA 等開放原始碼引擎，自行建置 PDP。
- 將授權邏輯與應用程式分離 — 當授權邏輯內嵌於應用程式程式碼中，或透過臨機強制機制實作時，可能會遭受意外或惡意變更，導致無意的跨租戶資料存取或其他安全性漏洞。為了協助減輕這些可能性，您可以使用 PAP (例如「已驗證的權限」) 來儲存授權原則，獨立於應用程式程式碼，並將強式治理套用至這些原則的管理。原則可以在高階宣告式語言中集中維護，這使得維護授權邏輯遠比在應用程式程式碼的多個區段中內嵌原則時要簡單得多。此方法也可確保一致地套用更新。
- 存取控制模型的彈性方法 — 角色型存取控制 (RBAC)、屬性型存取控制 (ABAC) 或兩種模型的組合都是存取控制的有效方法。這些模型試圖通過使用不同的方法來滿足企業的授權要求。本指南比較並對比這些模型，以協助您選擇適合您組織的模型。本指南也討論這些模型如何應用於不同的授權政策語言，例如 OPA/Rego 和 Cedar。本指南中討論的架構可以成功採用其中一種或兩種模型。
- 嚴格的 API 存取控制 — 本指南提供了一種方法，以最小的努力在應用程式中一致且持續地保護 API。這對於通常使用大量 API 來促進應用程式內部通訊的服務導向或微服務應用程式架構而言特別有用。嚴格的 API 存取控制有助於提高應用程式的安全性，使其不易受到攻擊或利用。

租用戶隔離和多租戶授權

本指南涉及租用戶隔離和多租戶授權的概念。租用戶隔離是指您在 SaaS 系統中使用的明確機制，以確保每個租用戶的資源 (即使在共用基礎結構上運作) 都是隔離的。多租戶授權是指授權輸入操作，並防止它們在錯誤的承租人上實施。假設用戶可以通過身份驗證和授權，並且仍然可以訪問另一個租戶的資源。驗證和授權不會封鎖此存取權，您必須實作租用戶隔離才能達成此目標。有關這兩個概念之間差異的更廣泛討論，請參閱 SaaS 架構基礎知識白皮書的租戶隔離部分。

存取控制的類型

您可以使用兩種廣泛定義的模型來實作存取控制：角色型存取控制 (RBAC) 和以屬性為基礎的存取控制 (ABAC)。每個模型都有優點和缺點，這將在本節中簡要討論。您應該使用的模型取決於您的具體使用案例。本指南中討論的架構支援這兩種模型。

RBAC

以角色為基礎的存取控制 (RBAC) 會根據通常與商務邏輯保持一致的角色來決定對資源的存取。權限會視情況與角色相關聯。例如，營銷角色將授權用戶在受限制的系統中執行營銷活動。這是一個相對簡單的訪問控制模型來實現，因為它與易於識別的業務邏輯保持一致。

在以下情況下，RBAC 模型效果較差：

- 您有不重複的使用者，其職責包含數個角色。
- 您有複雜的商務邏輯，使角色難以定義。
- 擴充至大型需要持續管理，並將權限對應至新角色和現有角色。
- 授權是以動態參數為基礎。

ABAC

以屬性為基礎的存取控制 (ABAC) 會根據屬性決定對資源的存取。屬性可以與使用者、資源、環境或甚至應用程式狀態相關聯。您的策略或規則會參照屬性，而且可以使用基本布林邏輯來判斷是否允許使用者執行動作。以下是權限的基本示例：

在付款系統中，財務部門的所有使用者都可以在工作時*/payments*間內在 API 端點處理付款。

財務部門的成員資格是決定存取權的使用者屬性/payments。還有一個與 /payments API 端點相關聯的資源屬性，該屬性僅允許在上班時間進行訪問。在 ABAC 中，使用者是否可以處理付款，取決於包含財務部門成員資格作為使用者屬性的策略，以及作為資源屬性的/payments時間。

ABAC 模型在允許動態，上下文和精細的授權決策方面非常靈活。然而，ABAC 模型是很難最初實現的。定義規則和政策以及列舉所有相關存取向量的屬性需要大量的前期投資才能實作。

轉儲混合方法

結合 RBAC 和 ABAC 可以提供兩種型號的一些優點。RBAC，與業務邏輯如此密切一致，比 ABAC 更容易實現。為了在做出授權決策時提供額外的粒度層，您可以將 ABAC 與 RBAC 結合使用。這種混合方法透過結合使用者的角色 (及其指派的權限) 與其他屬性來決定存取權限，以決定存取權限。使用這兩種模型可以簡單地管理和分配權限，同時還允許與授權決策相關的更高靈活性和細微性。

存取控制模型比較

下表比較先前討論的三種存取控制模型。這種比較旨在提供信息和高級別。在特定情況下使用存取模型可能不一定與此表格中所做的比較相關。

因素	RBAC	ABAC	混合
彈性	中	高	高
簡單	高	低	中
精細程度	低	高	中
動態決策和規則	否	是	是
上下文感知	否	是	有點
實施努力	低	高	中

實施一個 PDP

原則決策點 (PDP) 可以被指定為策略或規則引擎。此元件負責套用原則或規則，並傳回是否允許特定存取的決定。PDP 可以與基於角色的訪問控制（RBAC）和基於屬性的訪問控制（ABAC）模型一起運行; 但是，PDP 是 ABAC 的要求。PDP 允許應用程序代碼中的授權邏輯卸載到單獨的系統。這可以簡化應用程序代碼。它還提供了一個 easy-to-use 可重複的界面，用於對 API，微服務，前端後端（BFF）層或任何其他應用程序組件進行授權決策。

以下各節將討論實作 PDP 的三種方法。但是，這不是一個完整的列表。

PDP 實施方法：

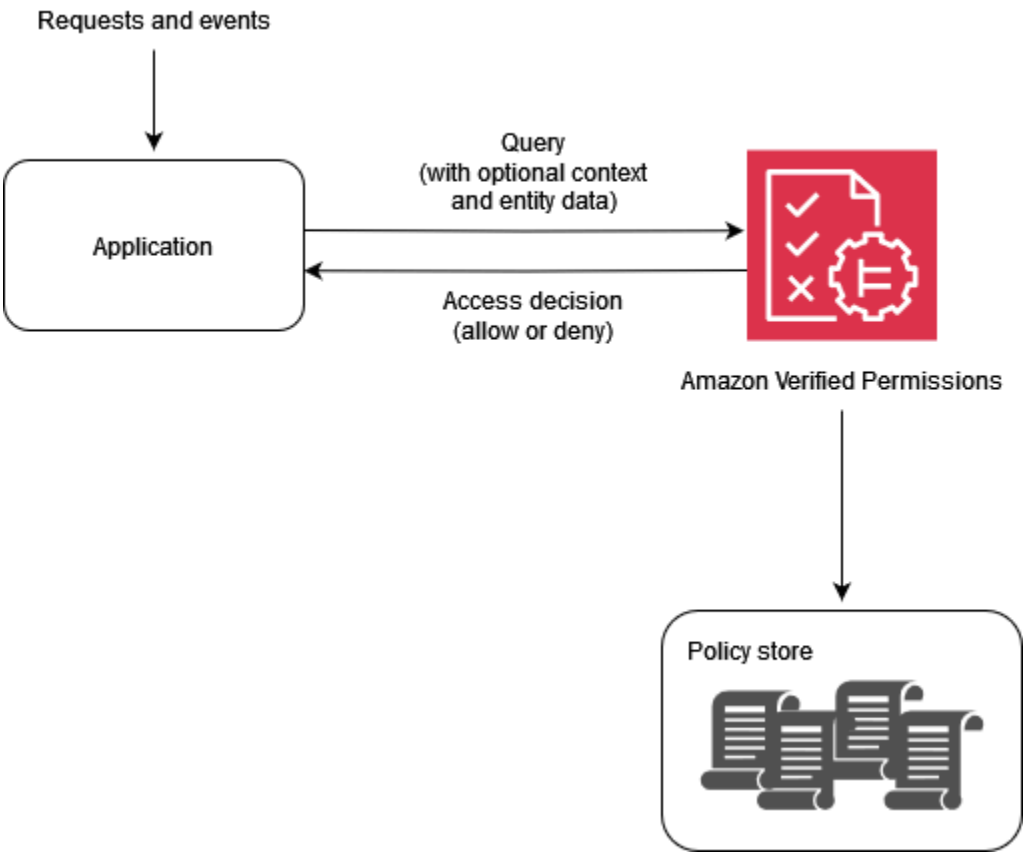
- 使用 Amazon 驗證許可實施 PDP
- 通過使用 OPA 實現 PDP
- 使用自訂原則引擎

使用 Amazon 驗證許可實施 PDP

Amazon 驗證許可是可擴展的精細許可管理和授權服務，可用於實作政策決策點 (PDP)。作為政策引擎，它可以協助您的應用程式即時驗證使用者動作，並反白顯示過度授權或無效的權限。透過外部化授權並集中管理原則管理和管理，協助您的開發人員更快地建置更安全的應用程式。透過將授權邏輯與應用程式邏輯分離，驗證的權限可支援原則解耦。

透過使用已驗證的權限來實作 PDP，並在應用程式內實作最低權限和持續驗證，開發人員可以使其應用程式存取與零信任原則保持一致。此外，安全性和稽核團隊可以更好地分析和稽核誰可以存取應用程式中的哪些資源。已驗證的權限使用 Cedar，這是一種專門打造且安全優先的開放原則原則語言，根據角色型存取控制 (RBAC) 和屬性型存取控制 (ABAC) 來定義原則型存取控制，以提供更精細的情境感知存取控制。

驗證許可為 SaaS 應用程式提供了一些有用的功能，例如透過使用多個身分供應商 (例如 Amazon Cognito、Google 和 Facebook) 啟用多租戶授權的能力。另一個對 SaaS 應用程式特別有用的「已驗證權限」功能是針對每個租用戶支援自訂角色。如果您正在設計客戶關係管理 (CRM) 系統，則一個租用戶可能會根據一組特定的條件來定義銷售商機的存取細微性。另一個承租人可能有另一個定義。「已驗證的權限」中的基礎權限系統可以支援這些變化，這使其成為 SaaS 使用案例的絕佳候選者。已驗證的權限也支援撰寫適用於所有租用戶的原則的功能，因此套用護欄原則以防止未經授權的 SaaS 提供者存取非常簡單。



為什麼要使用驗證權限？

透過身分供應商 (例如 Amazon Cognito) 使用已驗證的許可，為您的應用程式提供更動態、以政策為基礎的存取管理解決方案。您可以建置應用程式，協助使用者共用資訊和共同作業，同時維護其資料的安全性、機密性和隱私權。已驗證的權限為您提供精細的授權系統，以根據身分識別和資源的角色和屬性強制執行存取，有助於降低營運成本。您可以定義原則模型、在中央位置建立和儲存原則，以及在幾毫秒內評估存取要求。

在已驗證的權限中，您可以使用稱為 Cedar 的簡單、人類可讀的宣告式語言來表示權限。無論每個團隊的應用程式使用何種程式設計語言，都可以跨團隊共用 Cedar 編寫的原則。

使用已驗證權限時要考慮的事項

在已驗證的權限中，您可以建立原則，並將其作為佈建的一部分自動化。您也可以在執行階段建立原則，做為應用程式邏輯的一部分。最佳做法是，當您建立原則做為租用戶上線和佈建的一部分時，您應該使用持續整合和持續部署 (CI/CD) 管道來管理、修改及追蹤原則版本。或者，應用程式也可以管理、修改及追蹤原則版本；不過，應用程式邏輯本質上並不會執行此功能。若要在應用程式中支援這些功能，您必須明確設計應用程式以實作此功能。

如果需要提供來自其他來源的外部數據以實現授權決策，則必須檢索此數據並將其提供給已驗證權限，作為授權請求的一部分。依預設，此服務不會擷取其他前後關聯、實體和屬性。

雪松概述

Cedar 是一種靈活、可擴充且可擴充的原則式存取控制語言，可協助開發人員將應用程式權限表示為原則。系統管理員和開發人員可以定義允許或禁止使用者對應用程式資源採取行動的原則。可將多個策略附加至單一資源。當應用程式的使用者嘗試對資源執行動作時，您的應用程式會向 Cedar 原則引擎要求授權。Cedar 會評估適用的政策，並傳回ALLOW或DENY決策。Cedar 支援任何類型的主體和資源的授權規則，允許以角色為基礎的存取控制 (RBAC) 和以屬性為基礎的存取控制 (ABAC)，並透過自動推理工具支援分析。

Cedar 可讓您將業務邏輯與授權邏輯分開。當您透過應用程式的程式碼提出要求時，您可以呼叫 Cedar 的授權引擎來判斷該要求是否獲得授權。如果獲得授權（決定是ALLOW），您的應用程序可以執行請求的操作。如果未獲授權（決定是DENY），您的應用程序可能會返回錯誤消息。雪松的主要特點包括：

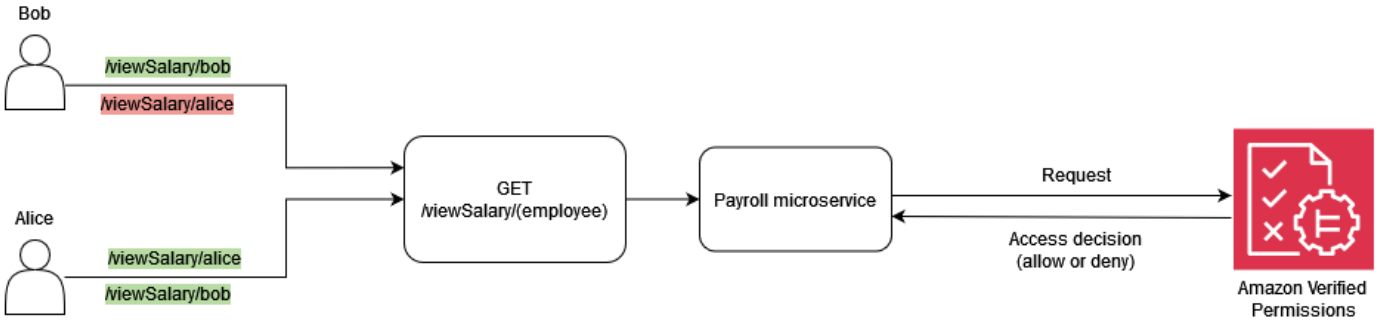
- 表現力 — Cedar 專為支援授權使用案例而打造，並以人類可讀性為考量而開發。
- 效能 — Cedar 支援索引原則以快速擷取，並提供快速且可擴充的即時評估，且延遲有限。
- 分析 — Cedar 支援可將原則最佳化並驗證您的安全模型的分析工具。

如需詳細資訊，請參閱 C edar 網站。

示例 1：具有已驗證權限和雪松的基本 ABAC

在此範例案例中，Amazon 驗證許可用於判斷允許哪些使用者存取虛構的薪資微服務中的資訊。本節包含 Cedar 程式碼片段，以示範如何使用 Cedar 呈現存取控制決策。這些範例並不是為了提供 Cedar 和已驗證權限所提供功能的完整探索。如需 Cedar 的更全面概觀，請參閱 C edar 文件。

在下圖中，我們想要強制執行與該viewSalaryGET方法相關聯的兩個一般商業規則：員工可以檢視自己的薪資，而員工可以檢視向他們報告之任何人的薪資。您可以使用已驗證的權限原則強制執行這些商業規則。



員工可以查看自己的薪水。

在 Cedar 中，基本建構是代表主參與者、動作或資源的實體。若要提出授權要求並使用已驗證權限原則開始評估，您需要提供主參與者、動作、資源和實體清單。

- 主體 (principal) 是登入的使用者或角色。
- 該操作（action）是由請求評估的操作。
- 資源（resource）是動作正在訪問的組件。
- 實體清單 (entityList) 包含評估請求所需的所有必要實體。

若要滿足商業規則員工可以檢視自己的薪資，您可以提供「已驗證的權限」原則，如下所示。

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

此原則會評估 Action is viewSalary 和請求中的資源是ALLOW否具有等於主參與者的屬性擁有者。例如，如果 Bob 是請求薪資報表的登入使用者，同時也是薪資報表的擁有者，則原則評估為ALLOW。

下列授權要求會提交至已驗證的權限，以供範例原則進行評估。在此範例中，Bob 是提出viewSalary要求的登入使用者。因此，Bob 是實體類型的主體Employee。Bob 嘗試執行的動作為，viewSalary,而viewSalary將顯示的資源Salary-Bob與類型一起執行Salary。為了評估 Bob 是否可以檢視Salary-Bob資源，您需要提供一個實體結構，將類型Employee與值 Bob (主參與者) 連結至具有該類型之資源的 owner 屬性Salary。您可以在中提供此結構entityList，其中與相關聯的屬性Salary包括擁有者，該擁有者指定包entityIdentifier含類型Employee和值的擁有者Bob。已驗證的權限會將授權要求中principal提供的與Salary資源相關聯的owner屬性進行比較，以做出決定。

```
{
  "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": {}
      }
    ]
  }
}
```

對已驗證權限的授權請求返回以下內容作為輸出，其中屬性`decision`是`ALLOW`或`DENY`。

```
{
    "determiningPolicies":
        [
            {
                "determiningPolicyId": "PAYROLLAPP_POLICYSTOREID"
            }
```

```
        ],
    "decision": "ALLOW",
    "errors": []
}
```

在此情況下，由於 Bob 嘗試檢視自己的薪資，所以傳送至「已驗證權限」的授權要求會評估為ALLOW。但是，我們的目標是使用「已驗證的權限」來強制執行兩個業務規則。說明下列內容的商業規則也應為 true：

員工可以查看向他們報告的任何人的薪水。

若要滿足此商業規則，您可以提供其他原則。下列策略會評估動作是ALLOW否為，viewSalary且請求中的資源是否具有等於主參與者的屬性owner.manager。例如，如果 Alice 是要求薪資報表的登入使用者，而 Alice 是報表擁有者的管理員，則原則評估為ALLOW。

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

下列授權要求會提交至已驗證的權限，以供範例原則進行評估。在此範例中，Alice 是提出viewSalary要求的登入使用者。因此愛麗絲是主體和實體的類型Employee。Alice 嘗試執行的動作為viewSalary，而顯viewSalary示的資源是值為Salary的類型Salary-Bob。為了評估 Alice 是否可以檢視Salary-Bob資源，您需要提供一個實體結構，將類型Employee與值連結Alice至manager屬性，之後必須與具有值的類型ownerSalary屬性相關聯Salary-Bob。您可以在中提供此結構entityList，其中與相關聯的屬性Salary包括擁有者，該擁有者指定包entityIdentifier含類型Employee和值的擁有者Bob。「已驗證的權限」會先檢查owner屬性，該屬性會評估為類型Employee和值Bob。然後，「已驗證的權限」會評估與相關聯的manager屬性，Employee並將其與提供的主體進行比較，以做出授權決定。在這種情況下，決定是ALLOW因為principal 和resource.owner.manager屬性是相等的。

```
{
  "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": []
      }
    ]
  }
}
```

到目前為止，在這個範例中，我們提供了與*viewSalary*方法相關聯的兩個商業規則，員工可以檢視自己的薪資，員工可以檢視向他們報告的任何人的薪資，以驗證權限作為原則，以獨立滿足每個商業規則的條件。您也可以使用單一已驗證權限原則來滿足這兩個商業規則的條件：

員工可以查看自己的薪水和向他們報告的任何人的薪水。

當您使用先前的授權要求時，下列原則會評估動作為`principal`，`viewSalary`且要求中`owner.manager`的資源是`ALLOW`否具有等於或屬性`owner`等於`principal`。

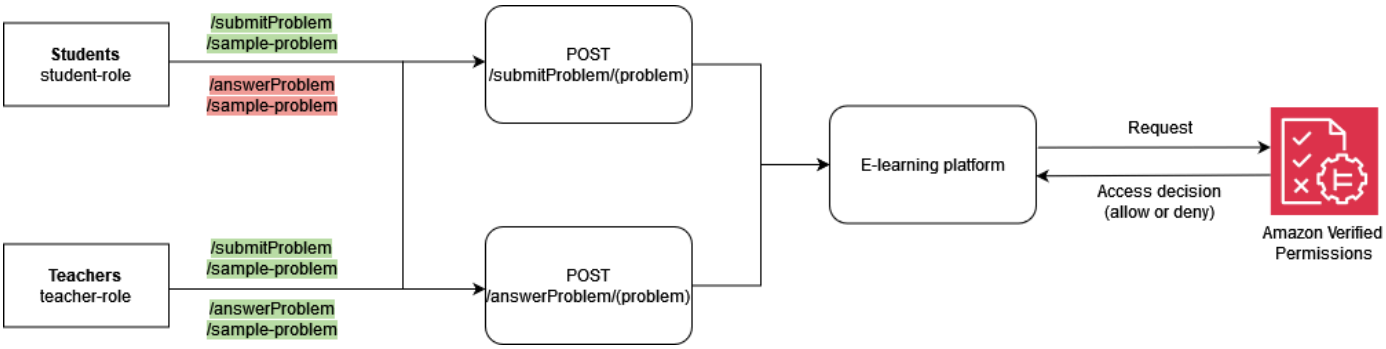
```
permit (
    principal,
    action == PayrollApp::Action::"viewSalary",
    resource
)
when {
    principal == resource.owner.manager ||
    principal == resource.owner
};
```

例如，如果 Alice 是要求薪資報表的登入使用者，而且 Alice 是報表擁有者或擁有者，則原則評估為`ALLOW`。

如需搭配 Cedar 原則使用邏輯運算子的詳細資訊，請參閱 Cedar 文件。

範例 2：具有已驗證權限和雪松的基本 RBAC

此範例使用已驗證的權限和 Cedar 來示範基本的 RBAC。如前所述，Cedar 的基本構造是一個實體。開發人員會定義自己的實體，並可選擇性地建立實體之間的關 下列範例包括三種類型的實體：UsersRoles、和Problems。 Students並且Teachers可以視為類型的圖元，Role,且每個圖元都User可以與零或任何一個相關聯Roles。



在 Cedar 中，這些關係通過鏈接RoleStudent到UserBob作為其父代來表達。該關聯以邏輯方式將所有學生用戶分組在一個組中。如需有關在 Cedar 中分組的詳細資訊，請參閱 Cedar 文件。

下列原則會ALLOW針對連結至類Role型之邏輯群組Students的所有主參與者，評估動作submitProblem,的決定。

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

下列原則會評估動作的決定answerProblem，submitProblem或ALLOW針對連結至類型Role之邏輯群組Teachers的所有主參與者進行評估。

```
permit (
    principal in ElearningApp::Role::"Teachers",
    action in [
        ElearningApp::Action::"submitProblem",
        ElearningApp::Action::"answerProblem"
    ],
    resource
);
```

若要使用這些原則評估要求，評估引擎必須知道授權要求中所參照的主參與者是否為適當群組的成員。因此，應用程式必須將相關的群組成員資格資訊傳遞給評估引擎，作為授權要求的一部分。這是透過entities屬性完成的，可讓您為 Cedar 評估引擎提供授權呼叫所涉及之主參與者和資源的屬性和群組成員資格資料。在下列程式碼中，群組成員資格是透過定義User::"Bob"為具有呼叫的父項來表示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": []
        }
```

```
      ]
  }
}
```

在此範例中，Bob 是提出answerProblem要求的登入使用者。因此，Bob 是主體，實體屬於類型User。Bob 嘗試執行的動作是answerProblem。為了評估 Bob 是否可以執行answerProblem動作，您需要提供一個實體結構，該實體結構將值User與實體連結，Bob並透過將父實體列為來指派其群組成員資格Role::"Students"。由於使用者群組中的實體Role::"Students"只能執行動作submitProblem，因此此授權要求會評估為DENY。

另一方面，如果值為Alice且User為群組一部分的類型Role::"Teachers"嘗試執行answerProblem動作，則授權要求會評估為ALLOW，因為原則會指定允許群組Role::"Teachers"中的主參與者對所有資源執answerProblem行動作。下面的代碼顯示了評估為的這種類型的授權請求ALLOW。

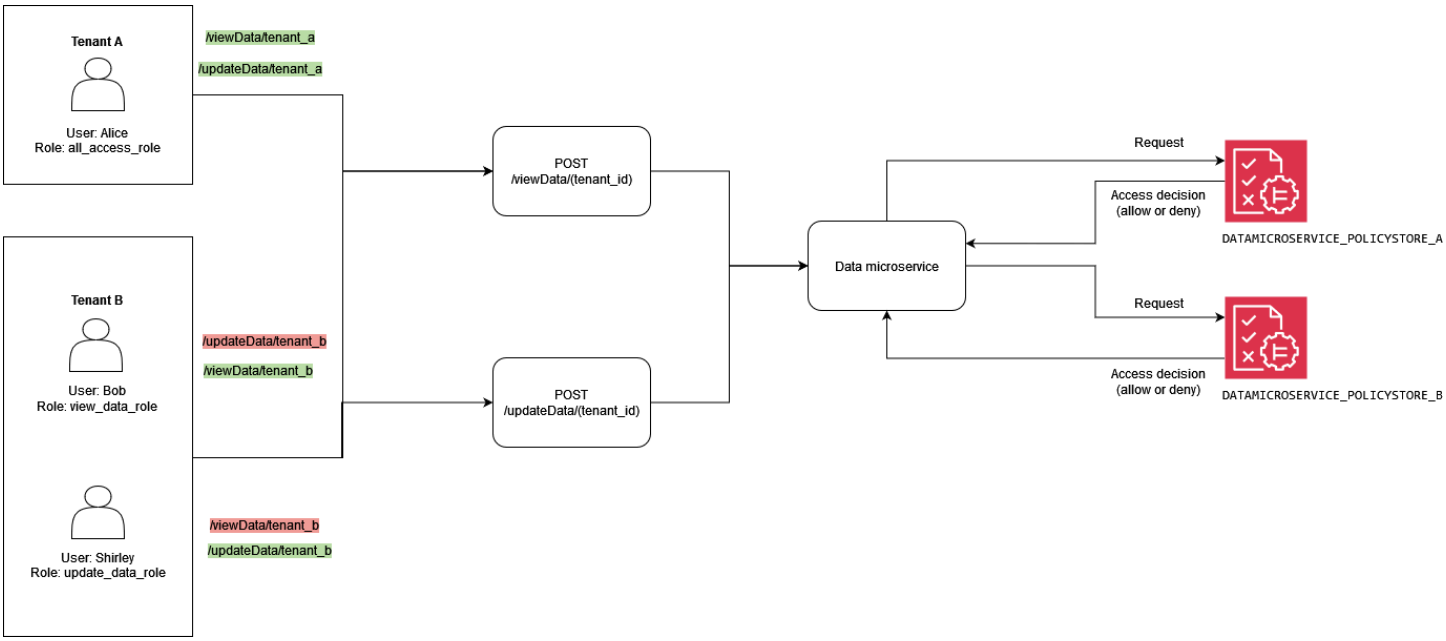
```
{
  "policyStoreId": "ELEARNING_POLICYSTOREID",
  "principal": {
    "entityType": "ElearningApp::User",
    "entityId": "Alice"
  },
  "action": {
    "actionType": "ElearningApp::Action",
    "actionId": "answerProblem"
  },
  "resource": {
    "entityType": "ElearningApp::Problem",
    "entityId":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

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

範例 3：使用 RBAC 進行多租戶存取控制

若要詳細說明先前的 RBAC 範例，您可以擴充您的需求以納入 SaaS 多租戶，這是 SaaS 提供者的常見需求。在多租用戶解決方案中，資源存取一律會代表指定的租用戶提供。也就是說，租用戶 A 的使用者無法檢視租用戶 B 的資料，即使該資料在邏輯上或實際並置於系統中。下列範例說明如何使用多個已驗證權限原則存放區來實作承租人隔離，以及如何使用使用者角色定義承租人內的權限。

使用每個承租人原則存放區設計模式是在使用已驗證權限實作存取控制時維持租用戶隔離的最佳做法。在此案例中，會分別針對個別的原則存放區和驗證租用戶 A DATAMICROSERVICE_POLICYSTORE_A 和DATAMICROSERVICE_POLICYSTORE_B承租人 B 使用者要求。如需有關多租用戶 SaaS 應用程式的已驗證權限設計考量的詳細資訊，請參閱已驗證的權限多租用戶設計考量一節。



下列原則位於原DATAMICROSERVICE_POLICYSTORE_A則存放區中。它會驗證主參與者是否屬於類型Role群組allAccessRole的一部分。在此情況下，主參與者可以對與承租人 A 相關聯的所有資源執行viewData和updateData動作。

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

下列原則位於DATAMICROSERVICE_POLICYSTORE_B原則存放區中。第一個原則會驗證主參與者是否屬於類型updateDataRoleRole群組。假設是這種情況，它會授與主參與者對與承租人 B 相關聯的資源執行updateData動作的權限。

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

第二個原則要求屬於類型viewDataRole群組的主參與者Role應允許對與承租人 B 相關聯的資源執行viewData動作。

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

從承租人 A 發出的授權要求必須傳送至DATAMICROSERVICE_POLICYSTORE_A原則存放區，並由屬於該存放區的原則進行驗證。在這種情況下，它是由前面討論的第一個原則作為本範例的一部分進行驗證。在此授權要求中，值為之類型User的主體Alice正在要求執行viewData動作。主參與者屬於類allAccessRole型群組Role。愛麗絲試圖對SampleData資源執行viewData操作。因為愛麗絲有這個allAccessRole角色，所以這個評估結果是一個ALLOW決定。

```
{
  "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，如果您檢視由租用戶 B 發出的要求User Bob，您會看到類似下列授權要求的內容。請求會傳送至原DATAMICROSERVICE_POLICYSTORE_B則存放區，因為它來自承租人 B. 在此請求中，主參與者Bob想要對資源SampleData執updateData行動作。但Bob是，不屬於可存取該資源動作updateData的群組的一部分。因此，請求會產生DENY決定。

```
{
  "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": []
      }
    ]
  }
}
```

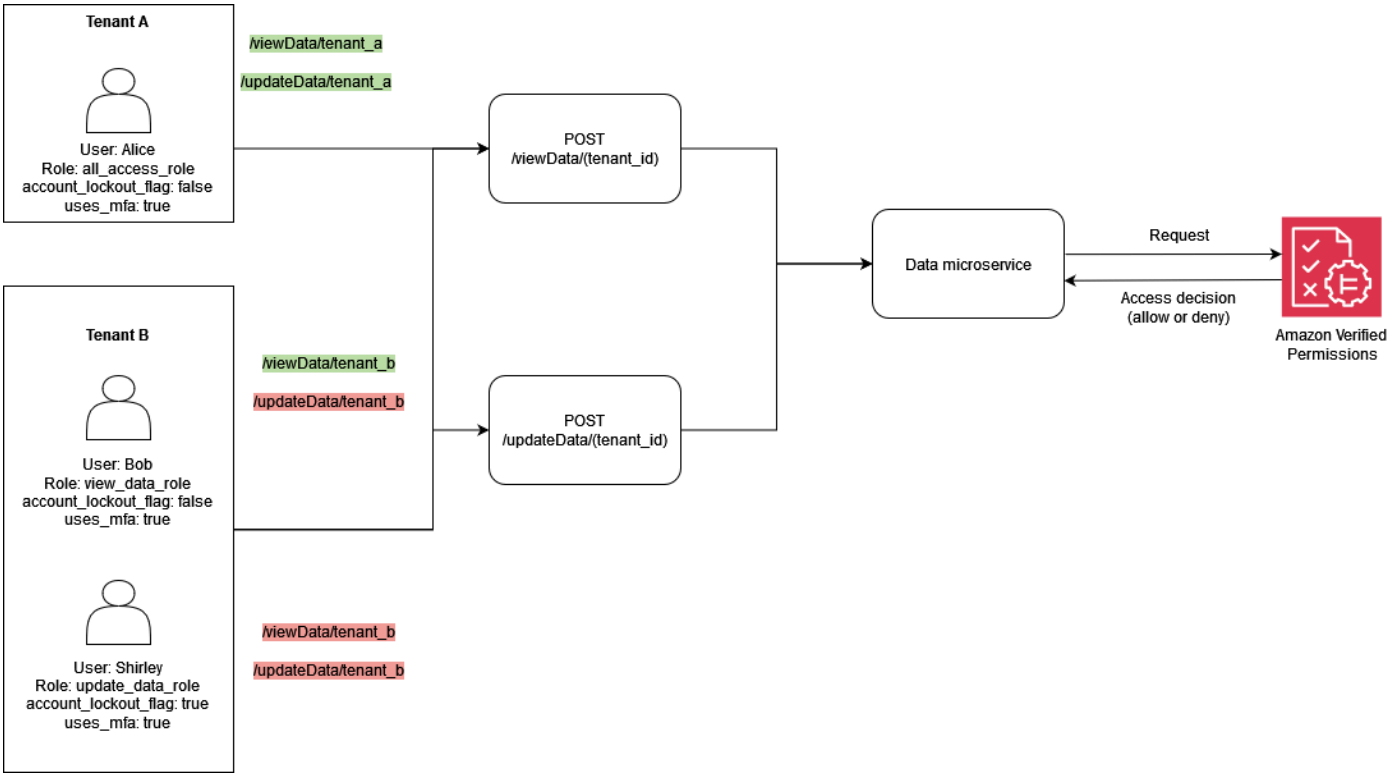
在第三個範例中，User Alice 嘗試對資源執行viewData動作SampleData。此要求會導向至DATAMICROSERVICE_POLICYSTORE_A原則存放區，因為主參與者Alice屬於承租人 A。Alice是

該類型群組allAccessRole的一部分Role，可讓她對資源執行viewData動作。因此，請求會導致一個ALLOW決定。

```
{
  "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": []
      }
    ]
  }
}
```

範例 4：透過 RBAC 和 ABAC 進行多租戶存取控制

若要增強上一節中的 RBAC 範例，您可以將屬性新增至使用者，以建立 RBAC-ABAC 混合式方法來進行多租用戶存取控制。此範例包含上一個範例中的相同角色，但會新增使用者屬性`account_lockout_flag`和前後關聯參數`uses_mfa`。此範例也採用不同的方法來同時使用 RBAC 和 ABAC 來實作多租用戶存取控制，並針對每個租用戶使用一個共用原則存放區，而不是不同的原則存放區。



此範例代表多租用戶 SaaS 解決方案，您需要在其中為租用戶 A 和租用戶 B 提供授權決策，類似於前面的範例。

若要實作使用者鎖定功能，此範例會將屬性新增`account_lockout_flag`至授權要求中的`User`實體主參與者。此旗標會鎖定使用者對系統的存取權，並將`DENY`所有權限指派給鎖定的使用者。該`account_lockout_flag`屬性與`User`實體相關聯，並且在多個工作階段中主動撤銷該旗標為`User`止。此範例使用`when`條件進行評估`account_lockout_flag`。

該示例還添加了有關請求和會話的詳細信息。上下文信息指定了會話已通過使用多因素身份驗證進行身份驗證。若要實作此驗證，此範例會使用`when`條件來評估`uses_mfa`旗標，做為前後關聯欄位的一部分。如需有關新增前後關聯的最佳作法的詳細資訊，請參閱 Cedar 文件。

```
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
};
```

除非資源與要求主體Tenant屬性位於相同的群組，否則此原則會阻止資源存取。這種維護租用戶隔離的方法稱為「單一共用多租用戶原則存放區」方法。如需有關多租用戶 SaaS 應用程式的已驗證權限設計考量的詳細資訊，請參閱已驗證的權限多租用戶設計考量一節。

該政策還可確保主體是其中的成員，allAccessRole並將行動限制為viewData和updateData。此外，此原則會驗證uses_mfa評估的前後關聯值為「account_lockout_flag是」，以falsetrue及是否為。

同樣地，下列原則可確保主參與者和資源都與相同的承租人相關聯，以防止跨承租人存取。此政策還可確保主體是其中的成員，viewDataRole並將行動限制為viewData。此外，它還會驗證account_lockout_flag是否false與uses_mfa評估的前後關聯值。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
};
```

第三個原則與前一個原則相似。此原則會要求資源是與所代表之實體對應之群組的成員principal.Tenant。這可確保主參與者和資源都與承租人 B 相關聯，以防止跨承租人存取。此原則可確保主體是其中的成員，updateDataRole並將動作限制為updateData。此外，此原則還會驗證uses_mfa評估的account_lockout_flag是false和前後關聯值是否為。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
};
```

本節前面討論的三個原則會評估下列授權要求。在此授權要求中，型別User與值的主體Alice會以角色發出updateData要求allAccessRole。 Alice具有值為Tenant的屬性Tenant::"TenantA"。該操作Alice嘗試執行是，updateData,並且將應用於該類型SampleData的資源Data。SampleData具有TenantA作為父實體。

根據原則存放區中的第一個<DATAMICROSERVICE_POLICYSTOREID>原則，假設符合原則when子句中的條件，就Alice可以對資源執行updateData動作。第一個條件需要要評估的principal.Tenant屬性TenantA。第二個條件需要主體的屬性account_lockout_flag為false。最終條件需要上下文uses_mfa是true。因為這三個條件都符合，因此請求會傳回ALLOW決定。

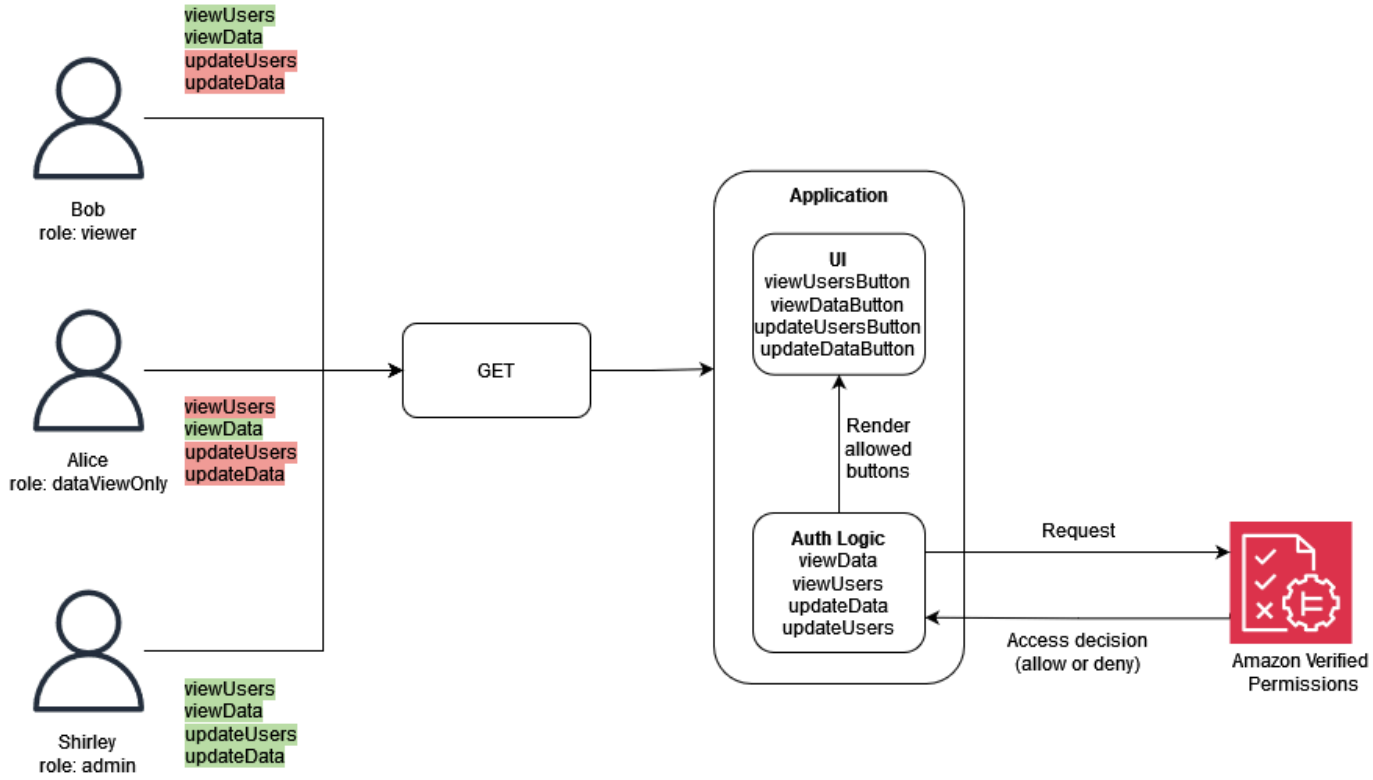
```
{
  "policyStoreId": "DATAMICROSERVICE_POLICYSTORE",
  "principal": {
      "entityType": "MultitenantApp::User",
      "entityId": "Alice"
  },
  "action": {
      "actionType": "MultitenantApp::Action",
      "actionId": "updateData"
  },
  "resource": {
      "entityType": "MultitenantApp::Data",
      "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"
            }
        ]
      }
    ]
  }
}
```

範例 5：使用已驗證的權限和 Cedar 進行 UI 篩選

您也可以使用已驗證的權限，根據授權的動作實作 UI 元素的 RBAC 篩選。這對於具有內容感應式 UI 元素的應用程式而言，在多租用戶 SaaS 應用程式的情況下，可能會與特定使用者或租用戶相關聯的應用程式而言非常有用。

在下列範例中，`Users`的`Role``viewer`不允許執行更新。對於這些使用者，UI 不應該呈現任何更新按鈕。



在此範例中，單頁 Web 應用程式有四個按鈕。哪些按鈕可見取決於目前登入應用程式`Role`的使用者。當單頁 Web 應用程式呈現 UI 時，它會查詢「已驗證的權限」，以判斷使用者獲授權執行的動作，然後根據授權決策產生按鈕。

下列原則指定值為`Role`的類型`viewer`可同時檢視使用者和資料。此原則的`ALLOW`授權決定需要`viewData`或`viewUsers`動作，而且還需要資源與類型`Data`或相關聯`Users`。`ALLOW`決定允許 UI 呈現兩個按鈕：`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"]
};
```

下列原則指定值為Role的類型只viewerDataOnly能檢視資料。此原則的ALLOW授權決定需要viewData採取動作，而且還需要資源與類型相關聯Data。ALLOW決定允許 UI 呈現按鈕viewDataButton。

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

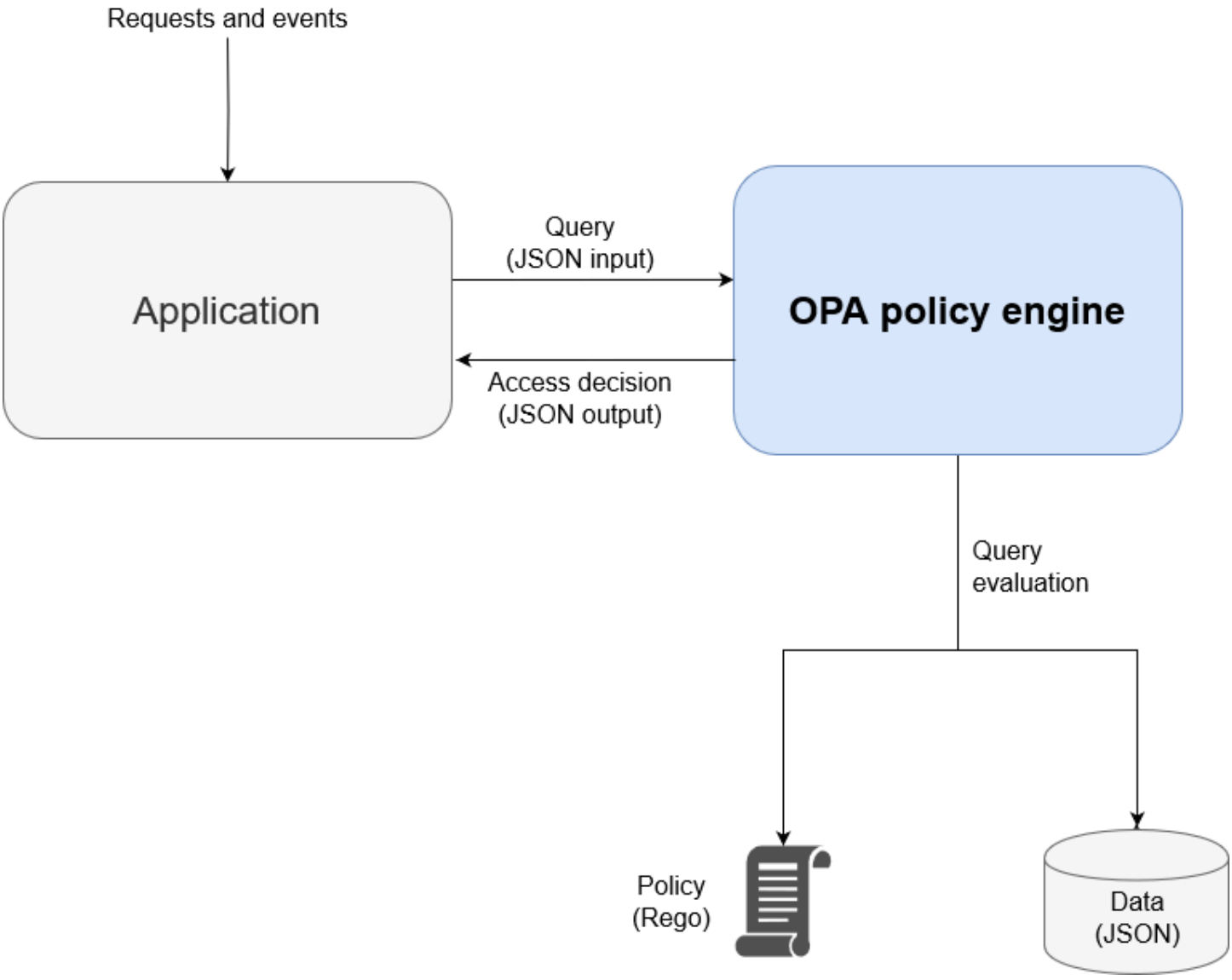
下列原則指定值為Role的類型admin可以編輯和檢視資料和使用者。此原則的ALLOW授權決策需要執行updateDataupdateUsers、viewData,或的動作viewUsers，並且還需要資源與類型Data或相關聯Users。ALLOW決定允許 UI 呈現所有四個按鈕：updateDataButtonupdateUsersButton、viewDataButton、和viewUsersButton。

```
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"]
};
```

通過使用 OPA 實現 PDP

開放原則代理程式 (OPA) 是開放原則代理程式 (OPA) 的一般用途原則引擎。OPA 有許多用例，但與 PDP 實現相關的用例是它能夠將授權邏輯與應用程序分離。這稱為原則解耦。OPA 在實施 PDP 有幾個原因很有用。它使用名為 Rego 的高階宣告式語言來草擬原則和規則。這些原則和規則與應用程式分開存在，而且無需任何應用程式特定邏輯即可轉譯授權決策。OPA 還公開了一個 RESTful API，使檢

索授權決策變得簡單明了。若要做出授權決定，應用程式會使用 JSON 輸入來查詢 OPA，而 OPA 會根據指定的原則評估輸入，以 JSON 傳回存取決策。OPA 還能夠導入在做出授權決策時可能相關的外部數據。



OPA 比自訂原則引擎有幾個優點：

- OPA 及其使用 Rego 進行原則評估提供彈性、預先建置的原則引擎，只需插入原則和任何必要資料即可進行授權決策。必須在自訂原則引擎解決方案中重新建立此原則評估邏輯。
- OPA 透過使用宣告式語言撰寫原則，簡化授權邏輯。您可以獨立於任何應用程式程式碼修改和管理這些原則和規則，而不需要應用程式開發技能。
- OPA 公開了一個 RESTful API，這簡化了與政策實施點（PEP）的集成。
- OPA 為驗證和解碼 JSON 網絡令牌（JWT）提供內置支持。

- OPA 是公認的授權標準，這意味著如果您需要幫助或研究來解決特定問題，文檔和示例將非常豐富。
- 採用 OPA 等授權標準可讓團隊之間共用以 Rego 撰寫的原則，無論團隊應用程式使用的程式設計語言為何。

OPA 不會自動提供兩件事：

- OPA 沒有強大的控制平面來更新和管理原則。OPA 確實提供了一些基本模式來實作原則更新、監視和記錄彙總，方法是公開管理 API，但與此 API 的整合必須由 OPA 使用者處理。最佳作法是，您應該使用持續整合和持續部署 (CI/CD) 管線來管理、修改和追蹤原則版本，以及管理 OPA 中的原則。
- OPA 預設無法從外部來源擷取資料。用於授權決策的外部數據源可以是保存用戶屬性的數據庫。如何將外部資料提供給 OPA 有一定的彈性 — 可以事先在本機快取，或是在要求授權決定時，從 API 動態擷取 — 但取得這些資訊並非 OPA 可以代表您執行的動作。

雷戈概述

Rego 是一種通用的策略語言，這意味著它適用於堆棧的任何層和任何域。Rego 的主要目的是接受經過評估的 JSON/YAML 輸入和資料，以針對基礎結構資源、身分識別和作業做出啟用原則的決策。Rego 可讓您撰寫有關堆疊或網域任何層的原則，而不需要變更或擴充語言。以下是 Rego 可以做出的一些決定示例：

- 這個 API 請求是允許還是拒絕？
- 這個應用程序的備份服務器的主機名是什麼？
- 這項建議的基礎設施變更的風險評分為何？
- 為了獲得高可用性，應將此容器部署到哪些叢集？
- 此微服務應該使用哪些路由資訊？

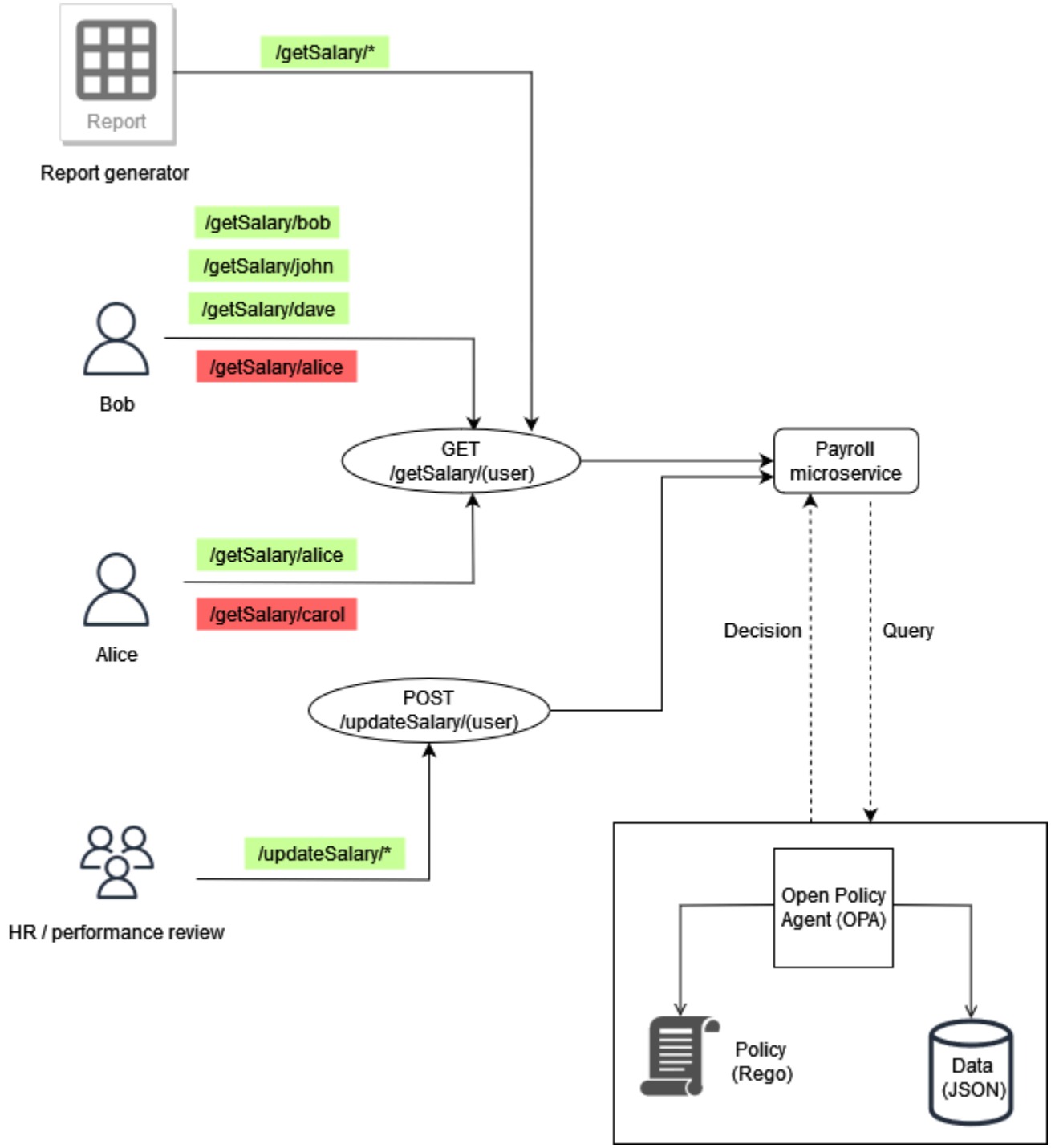
為了回答這些問題，Rego 採用了關於如何做出這些決定的基本理念。在 Rego 起草政策時的兩個關鍵原則是：

- 每個資源、身分識別或作業都可以表示為 JSON 或 YAML 資料。
- 原則是套用至資料的邏輯。

Rego 透過定義有關如何評估 JSON/YAML 資料輸入的邏輯，協助軟體系統做出授權決策。C、Java、Go 和 Python 等程式設計語言是解決這個問題的常見解決方案，但 Rego 的設計目的是專注於代表系統的資料和輸入，以及使用此資訊做出政策決策的邏輯。

範例 1：使用 OPA 和雷戈的基本 ABAC

本節描述了一種情況，其中 OPA 用於做出有關允許哪些用戶訪問虛構的薪資微服務中的信息的訪問決策。提供 Rego 程式碼片段，以示範如何使用 Rego 呈現存取控制決策。這些例子既不詳盡，也不是對 Rego 和 OPA 功能的全面探索。有關 Rego 的更全面的概述，我們建議您查閱 OPA 網站上的 Rego 文檔。



基本 OPA 規則範例

在上圖中，OPA 針對薪資微服務強制執行的存取控制規則之一是：

員工可以閱讀自己的薪水。

如果 Bob 嘗試存取薪資微服務以查看自己的薪資，薪資微服務可以將 API 呼叫重新導向至 OPA RESTful API，以做出存取決策。薪資服務會向 OPA 查詢具有以下 JSON 輸入的決策：

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

OPA 會根據查詢選取一個或多個策略。在此情況下，以 Rego 撰寫的下列原則會評估 JSON 輸入。

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

依預設，此原則會拒絕存取。然後，它會將其繫結至全域變數，以評估查詢中的輸入`input`。點運算子與此變數搭配使用，以存取變數的值。如`allow`果規則中的運算式也為真，則 Rego 規則會傳回 true。Rego 規則會驗證輸入`method`中的是否等於 GET。然後，它會在將清單`path`中的第二個元素指派給變數`getSalary``user`之前，驗證清單中的第一個元素。最後，它檢查被訪問的路徑是`/getSalary/bob`通過檢查發`user`出請`input.user`求是否匹配`user`變量。該規則`allow`應用 if-then 邏輯返回一個布爾值，如輸出所示：

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

使用外部資料的部分規則

若要示範其他 OPA 功能，您可以將需求新增至您要強制執行的存取規則。假設您想要在上圖的內容中強制執行此存取控制需求：

員工可以閱讀向他們報告的任何人的薪水。

在此範例中，OPA 可以存取可匯入的外部資料，以協助做出存取決策：

```
"managers": {
```

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

您可以在 OPA 中建立部分規則來產生任意 JSON 回應，該規則會傳回一組值而非固定回應。這是部分規則的範例：

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

此規則會傳回一組報告值的所有使用者input.user，在此情況下為 is bob。規則中的[_]構造用於遍歷集合的值。這是規則的輸出：

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

擷取此資訊有助於判斷使用者是否為管理員的直接報告。對於某些應用程序，返回動態 JSON 比返回簡單的布爾響應更好。

整合練習

最後一個存取要求比前兩個更複雜，因為它結合了兩個需求中指定的條件：

員工可以閱讀自己的工資和誰向他們報告的任何人的薪水。

若要滿足此需求，您可以使用此 Rego 政策：

```
default allow = false

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

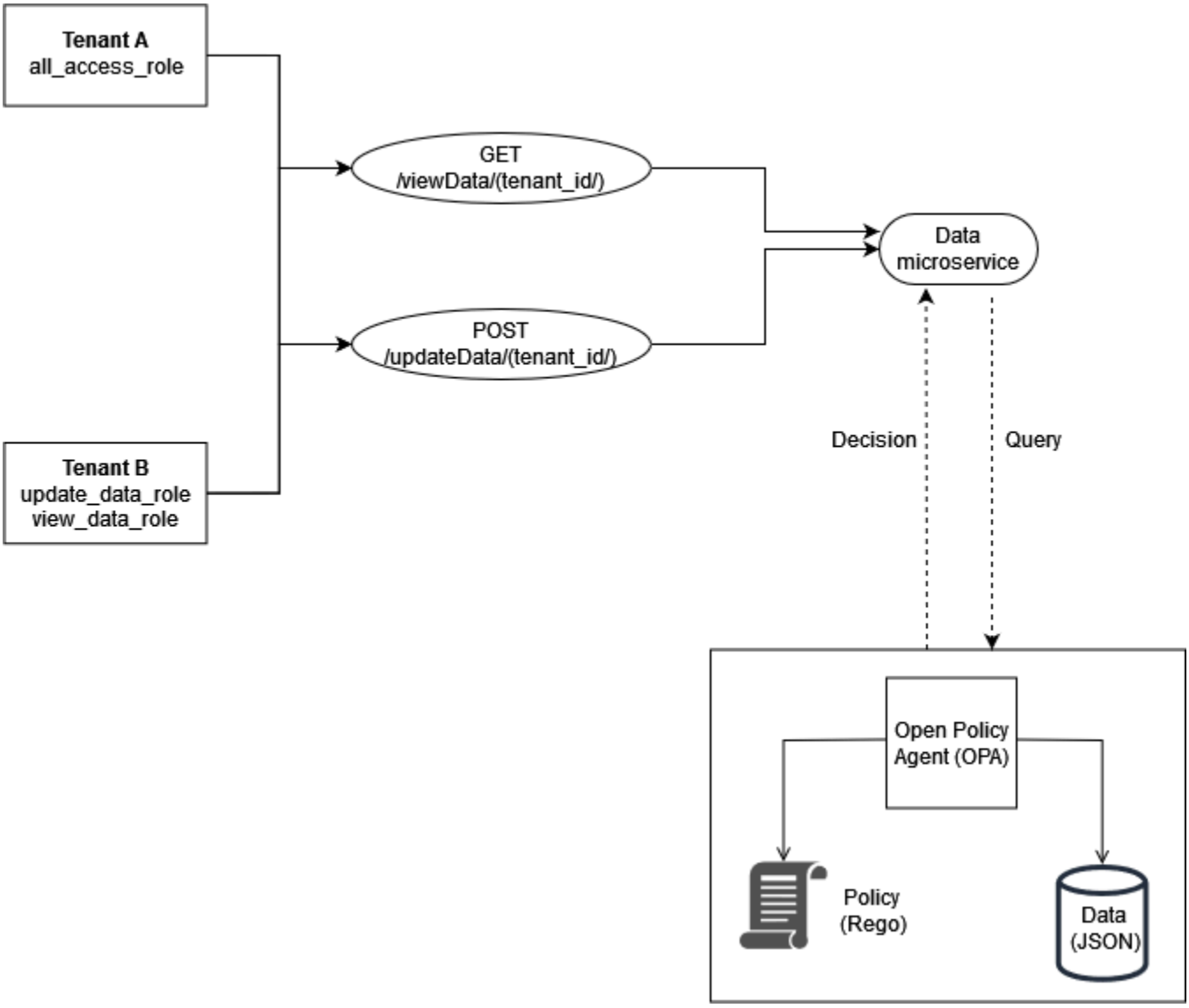
```
allow = true {
    input.method == "GET"
    input.path = ["getSalary", user]
    managers := data.managers[input.user][_]
    contains(managers, user)
}
```

如先前所述，原則中的第一個規則允許嘗試查看自己的薪資資訊的任何使用者存取。在 Rego 中有兩個具有相同名稱的規則allow、、作為邏輯或運算子的函數。第二個規則會擷取與 input.user (從上一個圖表中的資料) 相關聯的所有直屬報表清單，並將此清單指定給managers變數。最後，該規則檢查試圖查看其薪水的用戶是否是input.user通過驗證其名稱包含在managers變量中的直接報告。

本節中的示例非常基本，並不提供對 Rego 和 OPA 功能的完整或徹底的探索。有關更多信息，請查看 OPA 文檔，請參閱 OPA 自述文件，並在 GitHub Re go 遊樂場進行實驗。

範例 2：具有 OPA 和 Rego 的多租用戶存取控制和使用者定義的 RBAC

此範例使用 OPA 和 Rego 來示範如何在具有租用戶使用者定義自訂角色的多租戶應用程式的 API 上實作存取控制。它也會示範如何根據租用戶限制存取權。此模型顯示 OPA 如何根據高階角色中提供的資訊做出精細的權限決策。



租用戶的角色會儲存在外部資料 (RBAC 資料) 中，用來為 OPA 做出存取決策：

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

這些角色在由承租人使用者定義時，應儲存在外部資料來源或身分識別提供者 (IdP) 中，在將承租人定義的角色對應至權限和承租人本身時，可充當事實來源。

此範例使用 OPA 中的兩個原則來做出授權決策，並檢查這些原則如何強制執行租用戶隔離。這些原則會使用先前定義的 RBAC 資料。

```
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")
}
```

若要顯示此規則的運作方式，請考慮具有下列輸入的 OPA 查詢：

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

透過結合 RBAC 資料、OPA 原則和 OPA 查詢輸入，以下列方式做出此 API 呼叫的授權決定：

1. 來自的Tenant A使用者對其進行 API 呼叫/viewData/tenant_a。
2. Data 微服務會接收呼叫並查詢allowViewData規則，並傳遞 OPA 查詢輸入範例中顯示的輸入。
3. OPA 會使用 OPA 原則中查詢的規則來評估提供的輸入。OPA 還使用來自 RBAC 數據的數據來評估輸入。OPA 會執行下列作業：
   a. 驗證用來進行 API 呼叫的方法。GET
   b. 驗證所請求的路徑是viewData。
   c. 檢查路徑tenant_id中的是否等於與使用者input.tenant_id相關聯的。這可確保保持租用戶隔離。即使具有相同角色，其他承租人也無法在進行此 API 呼叫時獲得授權。
   d. 從角色的外部資料中提取角色權限清單，並將它們指派給變role_permissions數。此清單是使用與中的使用者相關聯的承租人定義角色來擷取 input.role.
   e. 檢role_permissions查是否包含權限 viewData.
4. OPA 會將下列決定傳回給資料微服務：

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

此程序顯示 RBAC 和租用戶感知如何有助於使用 OPA 做出授權決策。若要進一步說明這一點，請考慮/viewData/tenant_b使用下列查詢輸入的 API 呼叫：

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

此規則會傳回與 OPA 查詢輸入相同的輸出，雖然它是針對具有不同角色的不同承租人。這是因為此調用是針對的，/tenant_b並且view_data_role在 RBAC 數據仍然具有與其相關聯的viewData權限。若要強制執行相同類型的存取控制/updateData，您可以使用類似的 OPA 規則：

```
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")
}
```

此規則在功能上與allowViewData規則相同，但會驗證不同的路徑和輸入法。此規則仍可確保租用戶隔離，並檢查承租人定義的角色是否授與 API 呼叫者權限。要查看如何強制執行此操作，請檢查以下 API 調用的查詢輸入/updateData/tenant_b：

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

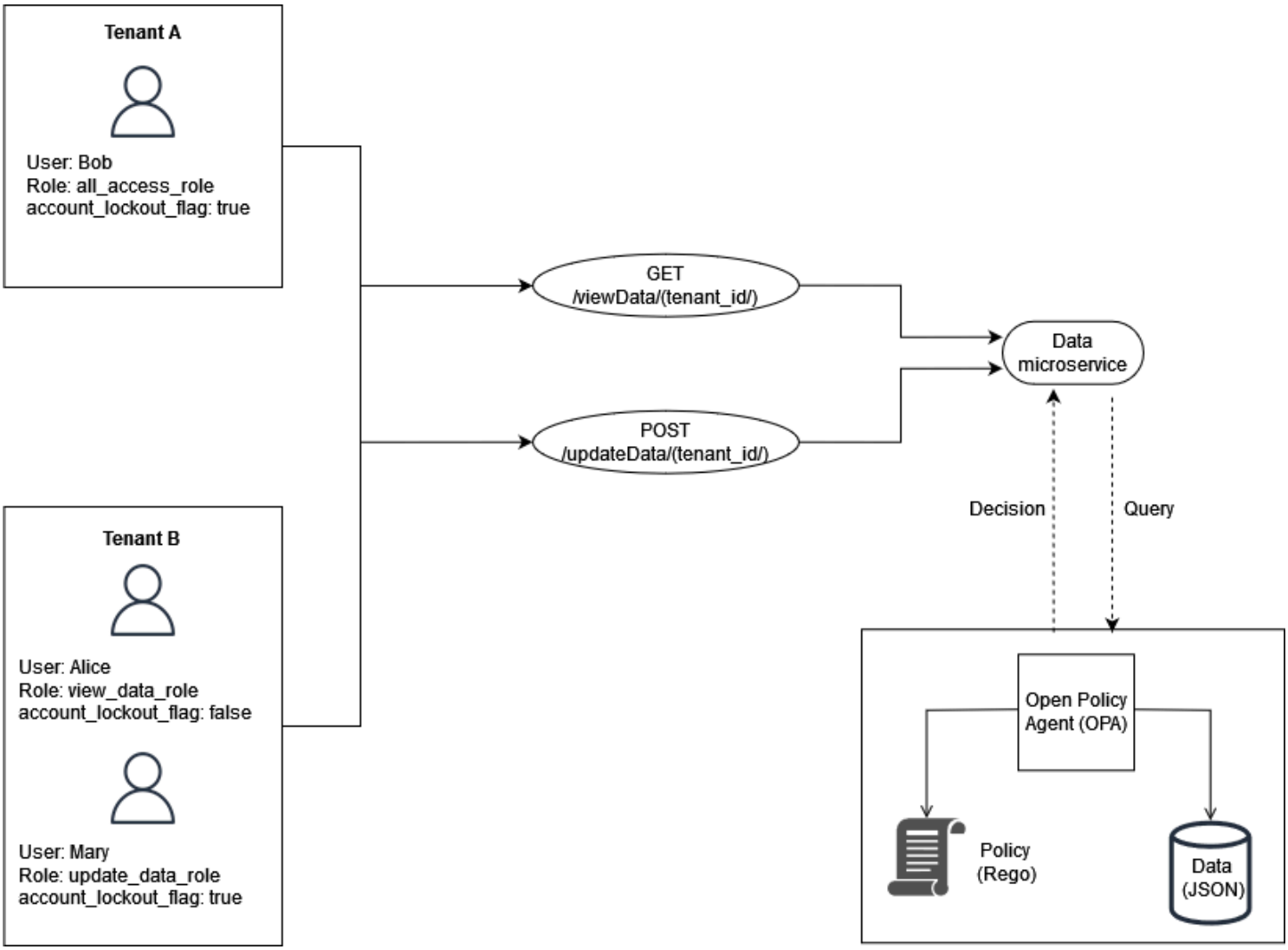
使用allowUpdateData規則評估此查詢輸入時，會傳回下列授權決策：

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

此通話將不會獲得授權。雖然 API 呼叫者與正確的呼叫者相關聯，`tenant_id`並且正在使用核准的方法呼叫 API，但`input.role`是由承租人`view_data_role`定義。`view_data_role`沒有`updateData`權限；因此，呼叫`/updateData`是未經授權的。對於`tenant_b`擁有`update_data_role`.

範例 3：RBAC 和 ABAC 的多租用戶存取控制，搭配 OPA 和 Rego

若要增強上一節中的 RBAC 範例，您可以將屬性新增至使用者。



此範例包含上一個範例中的相同角色，但會新增使用者屬性`account_lockout_flag`。這是不與任何特定角色相關聯的使用者特定屬性。您可以使用先前在此範例中使用的相同 RBAC 外部資料：

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

account_lockout_flag使用者屬性可以傳遞至資料服務，作為輸入至使用者 Bob 的 OPA 查詢/viewData/tenant_a的一部分：

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

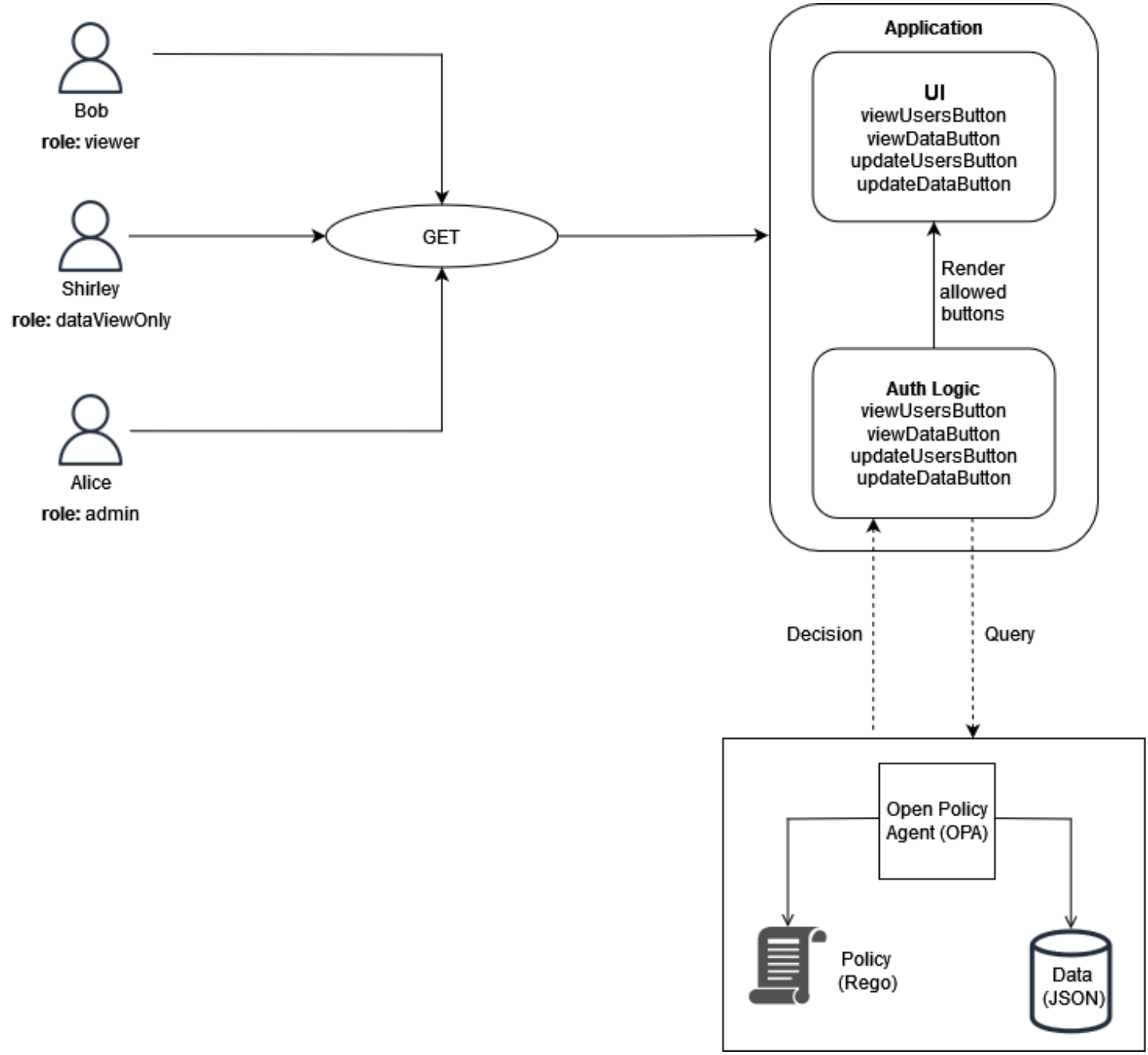
查詢存取決策的規則與先前的範例類似，但包含額外的一行來檢查account_lockout_flag屬性：

```
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"
}
```

此查詢會傳回的授權決定false。這是因為true適用於 Bob，雖然 Bob 具有正確的角色和租用戶，但 Rego 規則allowViewData拒絕存取。account_lockout_flag attribute

範例 4：使用 OPA 和 Rego 進行 UI 篩選

OPA 和 Rego 的靈活性支持過濾 UI 元素的能力。下列範例會示範 OPA 部分規則如何針對哪些元素應該顯示在 RBAC 的 UI 中做出授權決定。此方法是您可以使用 OPA 過濾 UI 元素的許多不同方法之一。



在此範例中，單頁 Web 應用程式有四個按鈕。比方說，你想過濾鮑勃的，雪莉的，和愛麗絲的用戶界面，使他們可以只看到對應於他們的角色的按鈕。當 UI 收到來自使用者的要求時，它會查詢 OPA 部

分規則，以決定應該在 UI 中顯示哪些按鈕。當 Bob（具有角色`viewer`）向 UI 發出請求時，查詢將以下內容作為輸入傳遞給 OPA：

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

OPA 使用為 RBAC 結構化的外部資料來做出存取決策：

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

OPA 部分規則會同時使用外部資料和輸入來產生允許的動作清單：

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

在部分規則中，OPA 會使用`input.role`指定的做為查詢的一部分來決定應該顯示哪些按鈕。Bob 具有角色`viewer`，而外部資料則指定檢視者有兩個權限：`viewUsers`和`viewData`。因此，此規則對 Bob (以及具有檢視者角色的任何其他使用者) 的輸出如下：

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

具有`dataViewOnly`角色的雪莉的輸出將包含一個權限按鈕：`viewData`。擁有該`admin`角色的愛麗絲的輸出將包含所有這些權限。查詢 OPA 時，這些回應會傳回至 UI。`user_permissions`然後，應用程式可以使用此回應來隱藏或顯示`viewUsersButton``viewDataButton``updateUsersButton`、、和`updateDataButton`。

使用自訂原則引擎

實作 PDP 的另一種方法是建立自訂原則引擎。此原則引擎的目標是將授權邏輯與應用程式分離。自訂原則引擎負責做出授權決策 (類似於已驗證的權限或 OPA)，以實現原則解耦。此解決方案與使用已驗證權限或 OPA 之間的主要差異在於，撰寫和評估原則的邏輯是針對自訂原則引擎自訂建置的。與引擎的任何互動都必須透過 API 或其他方法公開，才能進行授權決策才能觸及應用程式。您可以使用任何程式設計語言撰寫自訂原則引擎，或使用其他機制進行原則評估，例如通用運算式語言 (CEL)。

實作 PEP

原則強制執行點 (PEP) 負責接收傳送至原則決策點 (PDP) 進行評估的授權要求。PEP 可以位於必須保護資料和資源的應用程式中的任何位置，或套用授權邏輯的位置。與 PDP 相比，PEP 相對簡單。PEP 僅負責請求和評估授權決策，不需要任何授權邏輯。與 PDP 不同，PEP 無法集中在 SaaS 應用程序中。這是因為需要在整個應用程式及其存取點中實作授權和存取控制。PEP 可套用至 API、微服務、前端 (BFF) 層的後端，或應用程式中需要或需要存取控制的任何點。在應用程序中使 PEP 普遍可確保授權在多個點經常和獨立地進行驗證。

若要實作 PEP，第一個步驟是判斷應在應用程式中執行存取控制強制執行的位置。決定 PEP 應該整合到應用程式的位置時，請考慮以下原則：

如果應用程序公開了一個 API，則應該對該 API 進行授權和訪問控制。

這是因為在微服務導向或面向服務的架構中，API 作為不同應用程序功能之間的分隔符。在應用程序功能之間包含訪問控制作為邏輯檢查點是有意義的。強烈建議您將 PEP 納入為 SaaS 應用程式中存取每個 API 的先決條件。也可以在應用程序中的其他點集成授權。在單片應用程序中，可能需要將 PEP 集成在應用程序本身的邏輯中。沒有單一位置應包含 PEP，但請考慮使用 API 原則作為起點。

請求授權決定

PEP 必須向 PDP 請求授權決定。請求可以採用多種形式。請求授權決策的最簡單，最容易訪問的方法是將授權請求或查詢發送到 PDP（OPA 或已驗證權限）公開的 RESTful API。如果您使用的是已驗證的權限，您也可以使用 AWS SDK 來擷取授權決策來呼叫IsAuthorized方法。PEP 在這種模式中的唯一功能是轉發授權請求或查詢所需的信息。這可以像將 API 收到的請求作為輸入轉發到 PDP 一樣簡單。還有其他建立 PEP 的方法。例如，您可以將 OPA PDP 與以 Go 程式設計語言撰寫的應用程式整合為程式庫，而不是使用 API。

評估授權決策

PEP 需要包含邏輯來評估授權決策的結果。當 PDP 以 API 形式公開時，授權決策可能採用 JSON 格式，並由 API 呼叫傳回。PEP 必須評估此 JSON 代碼，以確定正在採取的操作是否獲得授權。例如，如果 PDP 旨在提供布林值允許或拒絕授權決策，則 PEP 可能只是檢查此值，然後傳回 HTTP 狀態碼 200 表示允許，並傳回 HTTP 狀態碼 403 表示拒絕。這種將 PEP 納入為存取 API 的先決條件的模式是在 SaaS 應用程式中實作存取控制的一種容易實作且高效率的模式。在更複雜的案例中，PEP 可能負責評估 PDP 傳回的任意 JSON 程式碼。必須撰寫 PEP，以包含解譯 PDP 傳回之授權決定所需的任何邏輯。因為 PEP 可能會在應用程式中的許多不同位置實作，因此建議您將 PEP 程式碼封裝成可重

複使用的程式庫或成品，以您選擇的程式設計語言。如此一來，您的 PEP 就可以在應用程式的任何時候輕鬆整合，而且只需最少的重工。

多租戶 SaaS 架構的設計模型

有許多方法可以實現 API 訪問控制和授權。本指南著重於三種對多租戶 SaaS 架構有效的設計模型。這些設計可作為實作原則決策點 (PDP) 和原則執行點 (PEP) 的高階參考，以便為應用程式形成具有凝聚力且無所不在的授權模型。

設計模型：

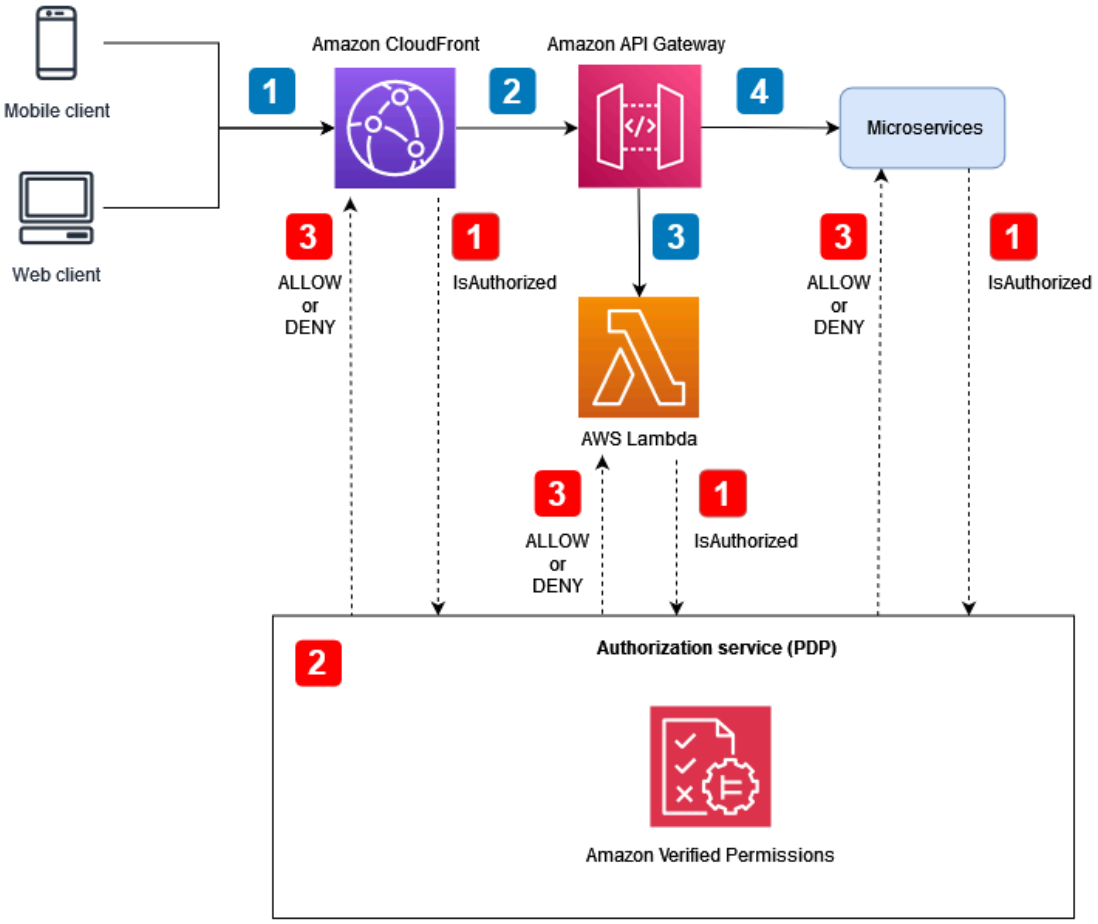
- Amazon 驗證許可的設計模型
- 適用於 OPA 的設計模型

Amazon 驗證許可的設計模型

在 API 上搭配 PEP 使用集中式 PDP

在 API 模型上具有原則強制執行點 (PEP) 的集中式原則決策點 (PDP) 遵循業界最佳實務，為 API 存取控制和授權建立有效且易於維護的系統。這種方法支持幾個關鍵原則：

- 授權和 API 存取控制會在應用程式中的多個點套用。
- 授權邏輯獨立於應用程序。
- 存取控制決策是集中的。



應用程序流程（圖中帶有藍色編號說明）：

1. 具有 JSON 網絡令牌（JWT）的身份驗證用戶向 Amazon CloudFront 生成 HTTP 請求。
2. CloudFront 將請求轉送至設定為 CloudFront來源的 Amazon API Gateway。
3. 會呼叫 API Gateway 自訂授權程式以驗證 JWT。
4. 微服務會回應要求。

授權與 API 存取控制流程 (圖中以紅色編號說明)：

1. PEP 會呼叫授權服務並傳遞要求資料，包括任何 JWT。
2. 在此情況下，授權服務 (PDP) 會使用請求資料作為查詢輸入，並根據查詢指定的相關原則對其進行評估。
3. 授權決定會傳回給 PEP 並進行評估。

此模型使用集中式 PDP 來做出授權決策。PEP 在不同的時間點實施，以向 PDP 發出授權請求。下圖顯示如何在假設的多租戶 SaaS 應用程式中實作此模型。

在此架構中，PEP 會在 Amazon CloudFront 和 Amazon API Gateway 的服務端點以及每個微服務請求授權決策。授權決定由授權服務 Amazon 驗證許可 (PDP) 決定。由於「已驗證權限」是完全受管理的服務，因此您不需要管理基礎結構。您可以使用 RESTful API 或 AWS SDK 與「已驗證的權限」進行互動。

您也可以將此架構與自訂原則引擎搭配使用。不過，從「已驗證的權限」中獲得的任何優點都必須取代為自訂原則引擎所提供的邏輯。

在 API 上使用 PEP 的集中式 PDP 提供了一個簡單的選項，可為 API 建立強大的授權系統。這簡化了授權過程，並提供了一個可重複的界面 easy-to-use，用於對 API，微服務，前端後端（BFF）層或其他應用程序組件進行授權決策。

使用雪松 SDK

Amazon 驗證許可使用 Cedar 語言在您的自訂應用程式中管理精細的許可。透過驗證權限，您可以將 Cedar 原則儲存在集中位置、利用低延遲和毫秒處理的優勢，以及稽核不同應用程式的權限。您也可以選擇性地將 Cedar SDK 直接整合到應用程式中，以提供授權決策，而無需使用已驗證的權限 此選項需要額外的自訂應用程式開發，才能管理和儲存您使用案例的原則。但是，它可能是一個可行的替代方案，特別是在由於互聯網連接不一致而間歇性或不可能訪問「已驗證權限」的情況下。

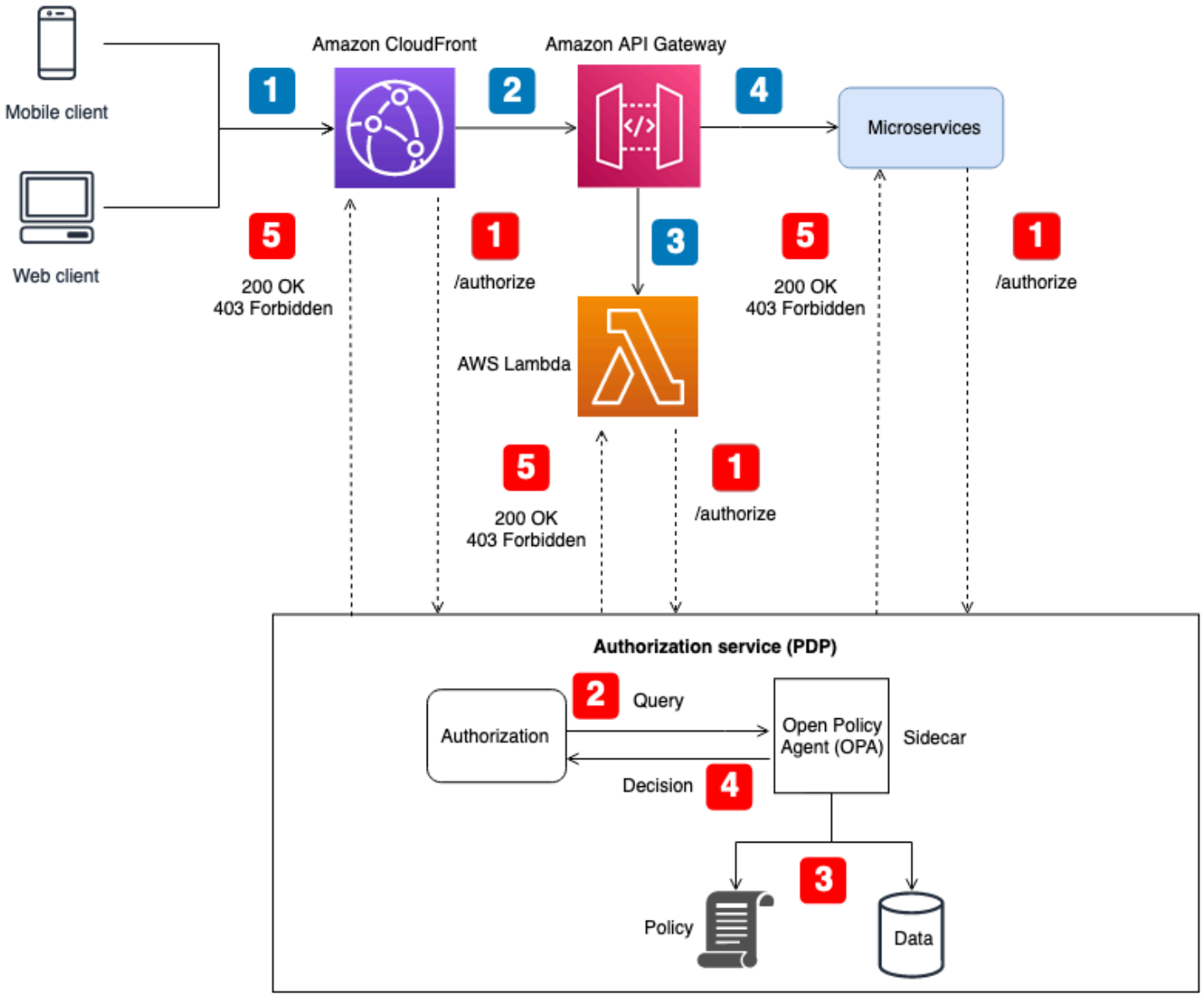
適用於 OPA 的設計模型

在 API 上搭配 PEP 使用集中式 PDP

在 API 模型上具有原則強制執行點 (PEP) 的集中式原則決策點 (PDP) 遵循業界最佳實務，為 API 存取控制和授權建立有效且易於維護的系統。這種方法支持幾個關鍵原則：

- 授權和 API 存取控制會在應用程式中的多個點套用。
- 授權邏輯獨立於應用程序。
- 存取控制決策是集中的。

此模型使用集中式 PDP 來做出授權決策。PEP 會在所有 API 中實作，以向 PDP 發出授權要求。下圖顯示如何在假設的多租戶 SaaS 應用程式中實作此模型。



應用程序流程（圖中帶有藍色編號說明）：

1. 具有 JWT 的身份驗證用戶向 Amazon CloudFront 生成 HTTP 請求。
2. CloudFront 將請求轉送至設定為 CloudFront來源的 Amazon API Gateway。
3. 會呼叫 API Gateway 自訂授權程式以驗證 JWT。
4. 微服務會回應要求。

授權與 API 存取控制流程 (圖中以紅色編號說明)：

1. PEP 會呼叫授權服務並傳遞要求資料，包括任何 JWT。

2. 授權服務 (PDP) 會取得要求資料，並查詢 OPA 代理程式 REST API，這是以附屬方式執行。請求數據作為查詢的輸入。
3. OPA 會根據查詢指定的相關原則評估輸入。如有必要，會匯入資料以作出授權決定。
4. OPA 將決定傳回授權服務。
5. 授權決定會傳回給 PEP 並進行評估。

在此架構中，PEP 會在 Amazon CloudFront 和 Amazon API Gateway 的服務端點以及每個微服務請求授權決策。授權決定由具有 OPA 側車的授權服務 (PDP) 進行。您可以將此授權服務當做容器或傳統伺服器執行個體來操作。OPA 附屬程式會在本機公開其 RESTful API，因此只有授權服務才能存取 API。授權服務公開可供 PEP 使用的單獨 API。讓授權服務充當 PEP 和 OPA 之間的中介，允許在 PEP 和 OPA 之間插入任何可能必要的轉換邏輯 — 例如，當 PEP 的授權要求不符合 OPA 預期的查詢輸入時。

您也可以將此架構與自訂原則引擎搭配使用。但是，從 OPA 獲得的任何優勢都必須由自訂原則引擎提供的邏輯取代。

在 API 上使用 PEP 的集中式 PDP 提供了一個簡單的選項，可為 API 建立強大的授權系統。它很容易實作，也提供可重複的介面 easy-to-use，用於針對 API、微服務、前端後端 (BFF) 層或其他應用程式元件進行授權決策。但是，這種方法可能會在應用程序中造成太多延遲，因為授權決策需要調用單獨的 API。如果網路延遲有問題，您可能會考慮使用分散式 PDP。

在 API 上搭配 PEP 使用分散式 PDP

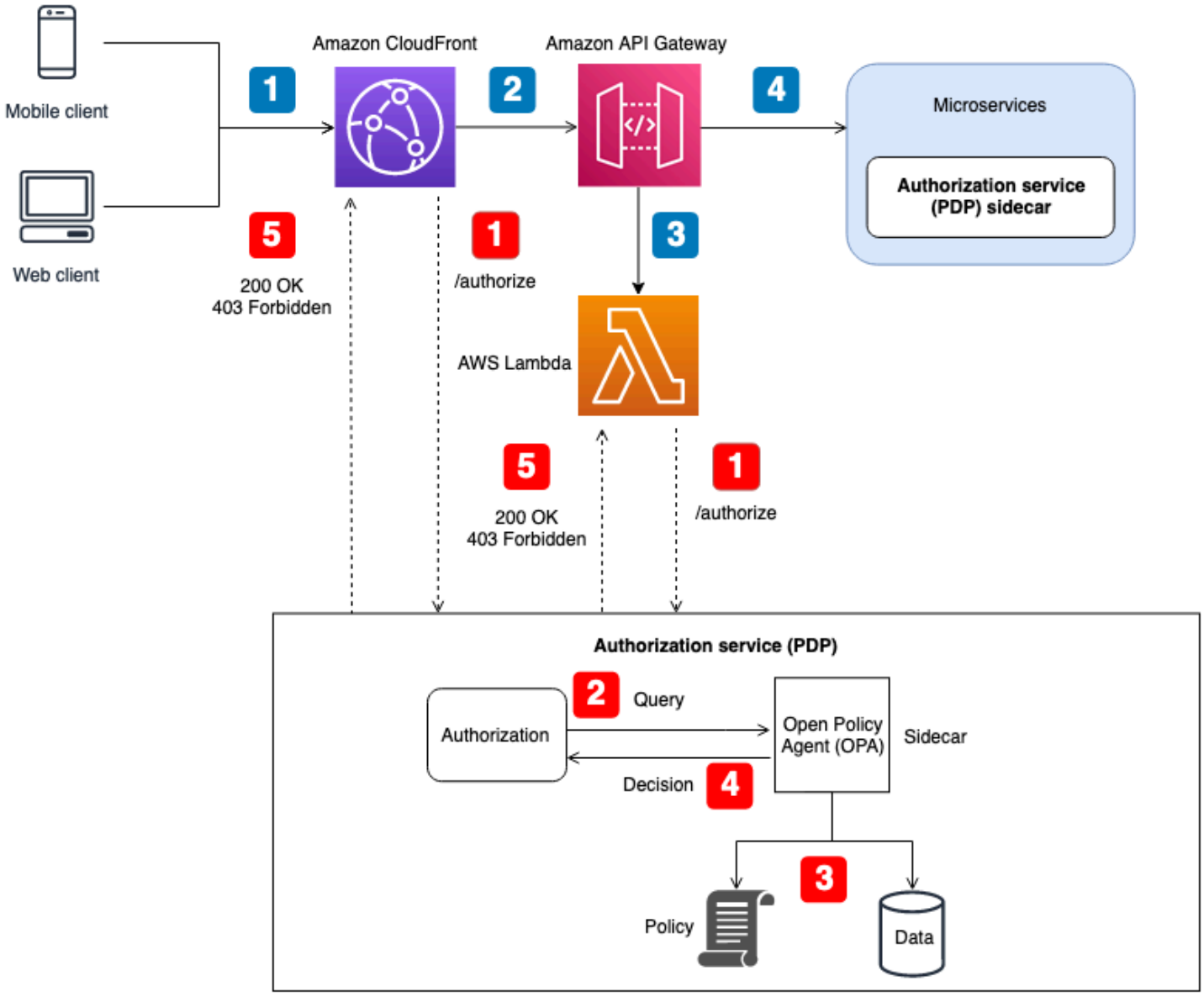
在 API 模型上具有原則強制執行點 (PEP) 的分散式原則決策點 (PDP) 遵循業界最佳實務，為 API 存取控制和授權建立有效的系統。與在 API 模型上使用 PEP 的集中式 PDP 一樣，此方法支援下列關鍵原則：

- 授權和 API 存取控制會在應用程式中的多個點套用。
- 授權邏輯獨立於應用程序。
- 存取控制決策是集中的。

您可能會想知道為什麼在分佈 PDP 時集中存取控制決策。雖然 PDP 可能存在於應用程式中的多個位置，但它必須使用相同的授權邏輯來做出存取控制決策。所有 PDP 在給予相同輸入的情況下提供相同的訪問控制決策。PEP 會在所有 API 中實作，以向 PDP 發出授權要求。下圖顯示了如何在假設的多租戶 SaaS 應用程序中實現此分佈式模型。

在這種方法中，PDP 在應用程序中的多個地方實現。對於具有可執行 OPA 並支援 PDP 的內建運算功能的應用程式元件，例如具有側載的容器化服務或 Amazon Elastic Compute Cloud (Amazon EC2) 執行個體，可將 PDP 決策直接整合到應用程式元件中，而無需對集中式 PDP 服務進行 RESTful API 呼叫。這有助於減少您在集中式 PDP 模型中可能遇到的延遲，因為並非每個應用程式元件都必須進行額外的 API 呼叫，以取得授權決策。不過，對於沒有可直接整合 PDP 的內建運算功能的應用程式元件 (例如 Amazon CloudFront 或 Amazon API Gateway 服務)，在此模型中仍然需要集中式 PDP。

下圖顯示了如何在假設的多租戶 SaaS 應用程序中實現集中式 PDP 和分散式 PDP 的組合。



應用程序流程（圖中帶有藍色編號說明）：

1. 具有 JWT 的身份驗證用戶向 Amazon CloudFront 生成 HTTP 請求。

2. CloudFront 將請求轉送至設定為 CloudFront來源的 Amazon API Gateway。
3. 會呼叫 API Gateway 自訂授權程式以驗證 JWT。
4. 微服務會回應要求。

授權與 API 存取控制流程 (圖中以紅色編號說明)：

1. PEP 會呼叫授權服務並傳遞要求資料，包括任何 JWT。
2. 授權服務 (PDP) 會取得要求資料，並查詢 OPA 代理程式 REST API，這是以附屬方式執行。請求數據作為查詢的輸入。
3. OPA 會根據查詢指定的相關原則評估輸入。如有必要，會匯入資料以作出授權決定。
4. OPA 將決定傳回授權服務。
5. 授權決定會傳回給 PEP 並進行評估。

在此架構中，PEP 會在服務端點 CloudFront 和 API Gateway 以及每個微服務要求授權決策。微服務的授權決定是由授權服務 (PDP) 做出的，該服務與應用程式元件一起運作。此模型適用於在容器或 Amazon 彈性運算雲端 (Amazon EC2) 執行個體上執行的微型服務 (或服務)。對於諸如 API Gateway 之類的服務的授權決策，仍然需要聯繫外部授權服務。 CloudFront 無論如何，授權服務都會公開可供 PEP 使用的 API。讓授權服務充當 PEP 和 OPA 之間的中介，允許在 PEP 和 OPA 之間插入任何可能需要的轉換邏輯 — 例如，當 PEP 的授權要求不符合 OPA 預期的查詢輸入時。

您也可以將此架構與自訂原則引擎搭配使用。但是，從 OPA 獲得的任何優勢都必須由自訂原則引擎提供的邏輯取代。

在 API 上使用 PEP 的分散式 PDP 可提供建立強大的 API 授權系統的選項。您可以輕鬆實作並提供可重複的介面 easy-to-use，以便針對 API、微服務、前端後端 (BFF) 層或其他應用程式元件進行授權決策。這種方法還具有減少您在集中式 PDP 模型中可能遇到的延遲的優點。

使用分散式 PDP 作為物件庫

您也可以從 PDP 中請求授權決策，該 PDP 可作為物件庫或封裝，以便在應用程式中使用。OPA 可以用作 Go 第三方庫。對於其他程式設計語言，採用此模型通常意味著您必須建立自訂原則引擎。

Amazon 驗證許可多租戶設計考量事項

在多租戶 SaaS 解決方案中使用 Amazon 驗證許可來實作授權時，有幾個設計選項需要考量。在探索這些選項之前，讓我們先釐清多租戶 SaaS 環境中隔離與授權之間的差異。隔離租用戶可防止輸入和輸出資料暴露給錯誤的承租人。授權可確保使用者具有存取承租人的權限。

在 [已驗證的權限] 中，原則會儲存在原則存放區中。如 [已驗證權限] 文件中所述，您可以針對每個租用戶使用個別的原則存放區來隔離租用戶的原則，或允許租用戶使用單一原則存放區為所有承租人共用原則。本節討論這兩種隔離策略的優缺點，並說明如何使用分層部署模型來部署這些策略。如需其他內容，請參閱已驗證的權限文件。

雖然在本節中討論的 critieria 著重於已驗證的權限，但一般概念植根於隔離心態和它提供的指導。SaaS 應用程式必須始終將租用戶隔離視為其設計的一部分，而這項隔離的一般原則延伸至包含 SaaS 應用程式中的已驗證權限。本節還引用了核心 SaaS 隔離模型，例如孤立的 SaaS 模型和集區 SaaS 模型。如需其他資訊，請參閱 AWS Well-Architected 的架構 SaaS Lens 中的核心隔離概念。

設計多租戶 SaaS 解決方案時的主要考量因素是租用戶隔離和租用戶上線。租用戶隔離會影響安全性、隱私權、備援和效能。租戶加入會影響您的營運流程，因為它與營運開銷和可觀察性有關。經歷 SaaS 旅程或實施多租戶解決方案的 Organizations 必須始終優先考慮 SaaS 應用程序處理租賃的方式。雖然 SaaS 解決方案可能傾向於特定的隔離模型，但整個 SaaS 解決方案並不一定需要一致性。例如，您為應用程式的前端元件選擇的隔離模型可能與您為微服務或授權服務選擇的隔離模型不同。

設計考量：

- 租戶入職和用戶租戶註冊
- 每個租用戶原則儲存
- 一個共用多租用戶原則存放區
- 分層部署模型

租戶入職和用戶租戶註冊

SaaS 應用程式會觀察 SaaS 身分識別的概念，並遵循將使用者身分繫結至租用戶身分的一般最佳做法。繫結涉及將租用戶識別碼儲存為身分識別提供者中使用者的宣告或屬性。這會將識別對應至租用戶的責任從每個應用程式轉移到使用者註冊程序。然後，每個已驗證的使用者都具有正確的租用戶身分，做為 JSON Web Token (JWT) 的一部分。

同樣地，選擇授權要求的正確原則存放區不應由應用程式邏輯決定。若要決定應使用特定授權要求的原則儲存區，請維護使用者與原則存放區的對應，或維護租用戶與原則存放區的對應。這些對應通常

會維護在應用程式參考的資料存放區中，例如 Amazon DynamoDB 或 Amazon Relational Database Service 服務 (Amazon RDS)。您也可以透過身分識別提供者 (IdP) 中的資料提供或補充這些對映。然後，通常會透過 JWT 將租用戶、使用者和原則存放區之間的關係提供給使用者，該 JWT 包含授權要求所需的所有關係。

此範例顯示 JWT 可能如何顯示屬於承租人的使用者`Alice`，以`TenantA`及使用原則存放區 ID 與原則存放區 ID 進行授權的使`ps-43214321`用者。

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

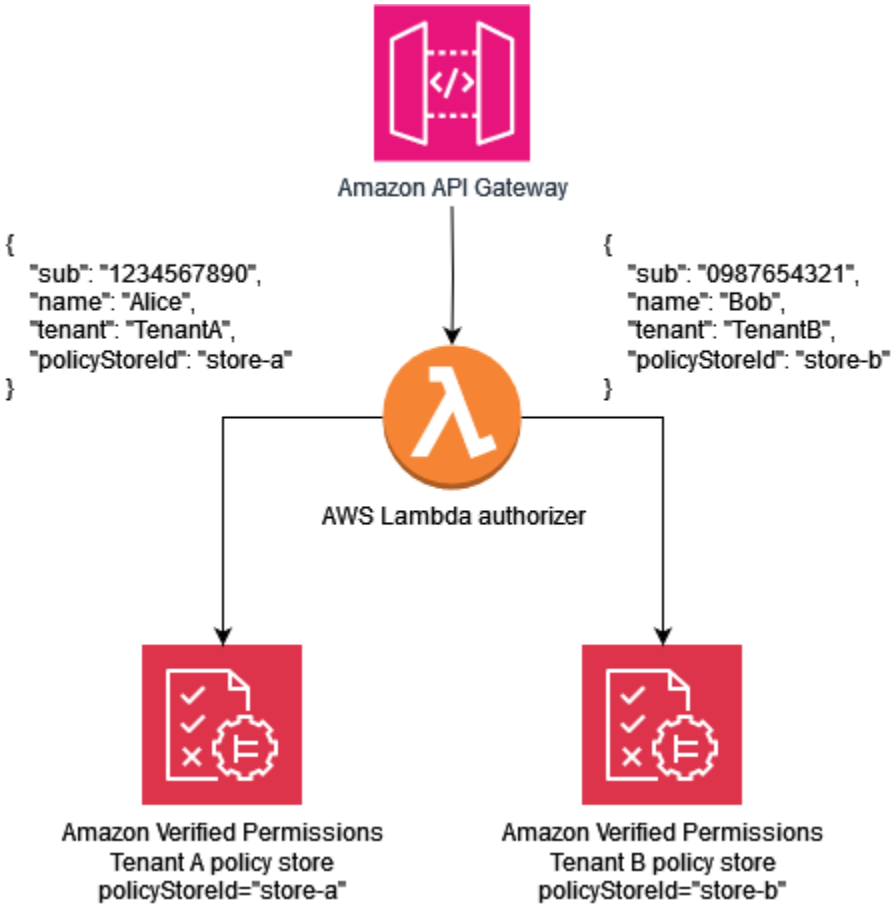
每個租用戶原則儲存

Amazon 驗證許可中的每個租用戶原則存放區設計模型會將 SaaS 應用程式中的每個租用戶與其自己的政策存放區相關聯。此模型類似於 SaaS 筒倉隔離模型。這兩種模式都要求創建特定於租戶的基礎設施，並具有類似的優點和缺點。這種方法的主要優點是基礎結構強制承租人隔離、支援每個租用戶的獨特授權模型、消除吵雜的鄰居問題，以及減少原則更新或部署失敗的影響範圍。此方法的缺點包括更複雜的租用戶上線程序、部署和作業。如果解決方案具有每個租用戶唯一的原則，則建議使用每個租用戶原則存放區。

如果您的 SaaS 應用程式需要，每個租用戶原則存放區模型可以為租用戶隔離提供高度孤立的方法。您也可以將此模型與集區隔離搭配使用，但您的「已驗證權限」實作不會共用更廣泛的集區隔離模型 (例如簡化的管理和作業) 的標準優點。

在每個租用戶原則存放區中，租用戶隔離是透過在使用者註冊程序期間將租用戶的原則存放區識別碼對應至使用者的 SaaS Identity (如前面所述) 來達成。這種方法會強烈地將租用戶的原則存放區與使用者主體關聯，並提供一致的方式來共用整個 SaaS 解決方案的對應。您可以將 SaaS 應用程式的對應作為 IdP 的一部分或在外部資料來源 (例如 DynamoDB) 中維護來提供對應。這也可確保主體是承租人的一部分，並確保使用承租人的原則存放區。

此範例顯示包含`policyStoreId`和`tenant`使用者的 JWT 如何從 API 端點傳送至 AWS Lambda 授權者中的政策評估點，並將要求路由至正確的原則存放區。



下列範例原則說明每個租用戶原則存放區設計範例。使用者`Alice`所屬`TenantA`.的policyStoreId`store-a`也會對應至的承租人識別，`Alice`,並強制使用正確的原則存放區。這可確保使用`TenantA`的策略。

> **Note**
>
> 每個租用戶原則存放區模型會隔離租用戶的原則。授權強制執行允許用戶對其數據執行的操作。使用此模型的任何假設應用程式中涉及的資源，都應該透過使用其他隔離機制來隔離，如 AWS Well-Architected 的架構、 SaaS Lens 文件中所定義。

在此原則中，`Alice`具有檢視所有資源資料的權限。

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

若要提出授權要求並使用已驗證權限原則開始評估，您需要提供與對應至承租人的唯一 ID 對應的原則存放區 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":[]
            }
         ]
      ]
   }
}
```

使用者Bob屬於承租人 B，也會 policyStoreId`store-b`對應至的承租人識別碼Bob，以強制使用正確的原則存放區。這可確保使用租戶 B 的政策。

在此原則中，Bob具有自訂所有資源資料的權限。在此範例中，customizeData可能是只針對租用戶 B 的動作，因此該原則對於租用戶 B 而言是唯一的。每個租用戶原則存放區模型本質上支援每個租用戶的自訂原則。

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

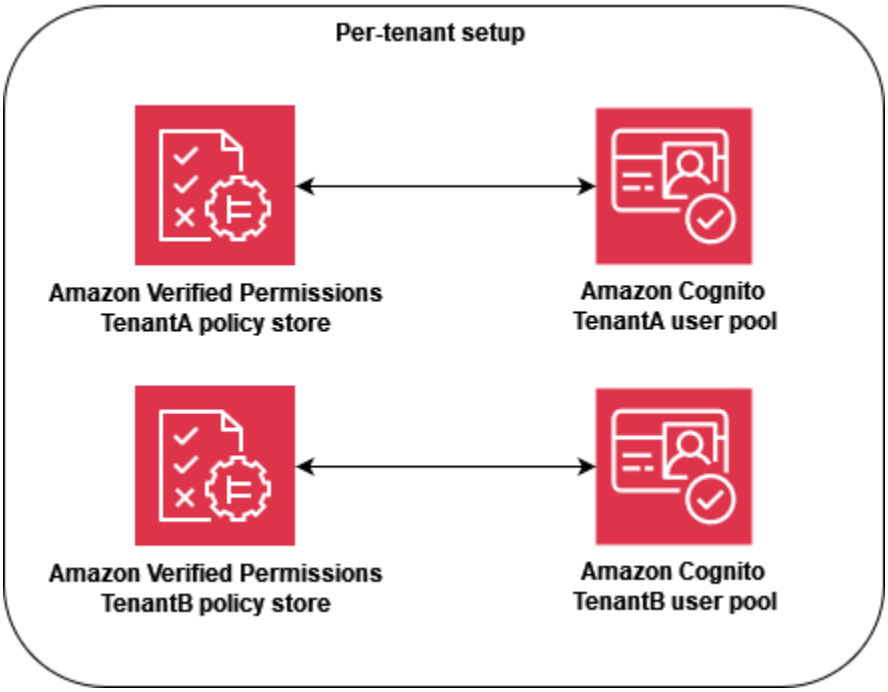
若要提出授權要求並使用已驗證權限原則開始評估，您需要提供與對應至承租人的唯一 ID 對應的原則存放區 ID store-b。

```
{
   "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":[]
            }
        ]
    ]
  }
}
```

使用已驗證權限，可以 (但不是必要) 將 IdP 與原則存放區整合。此整合可讓原則明確參照識別身分存放區中的主體做為原則的主體。如需如何將 Amazon Cognito 整合為已驗證許可的 IdP 的詳細資訊，請參閱已驗證的許可文件和 Amazon Cognito 文件。

將原則存放區與 IdP 整合時，每個原則存放區只能使用一個身分識別來源。例如，如果您選擇將已驗證許可與 Amazon Cognito 整合，則必須鏡像用於已驗證許可政策存放區和 Amazon Cognito 使用者集區的租用戶隔離的策略。策略存放區和使用者集區也必須位於相同的位置 AWS 帳戶。



在作業層級上，每個租用戶原則存放區具有稽核優勢，因為您可以輕鬆地針對每個租用戶AWS CloudTrail 單獨查詢記錄的活動。不過，我們仍建議您將每個租用戶維度上的其他自訂指標記錄到 Amazon CloudWatch。

每個租用戶原則存放區方法還需要密切注意兩個「已驗證權限」配額，以確保它們不會干擾 SaaS 解決方案的操作。這些配額是每個帳戶每個區域的政策存放區，以及每個帳戶每個區域每秒的IsAuthorized請求數。您可以要求增加這兩個配額。

如需如何實作每個租用戶政策存放區模型的詳細範例，請參閱 AWS 部落格文章使用 Amazon 驗證許可與每個租用戶政策存放區使用 SaaS 存取控制。

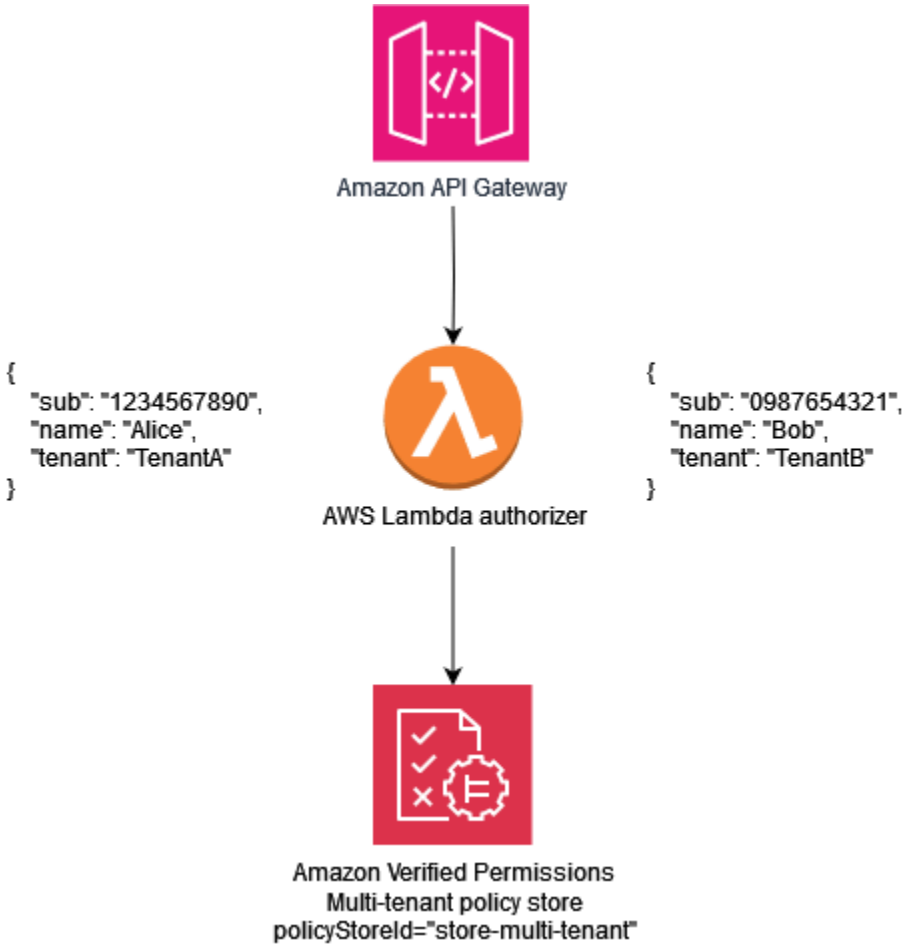
一個共用多租用戶原則存放區

單一共用多租戶政策存放區設計模型在 SaaS 解決方案中，針對所有租戶使用 Amazon 驗證許可中的單一多租戶政策存放區。此方法的主要優點是簡化管理和作業，特別是因為您不需要在租用戶上線期間建立其他原則存放區。這種方法的缺點包括政策更新或部署中的任何故障或錯誤所造成的影響範圍增加，以及更大程度地暴露於嘈雜的鄰居影響。此外，如果您的解決方案需要每個租用戶的唯一政策，我們不建議使用此方法。在此情況下，請改用每個租用戶原則存放區模型來確保使用正確承租人的原則。

單一共用多租用戶原則存放區方法類似於 SaaS 集區隔離模型。如果您的 SaaS 應用程式需要，它可以提供租用戶隔離的集區方法。如果您的 SaaS 解決方案將孤立隔離套用至其微服務，您也可以使用此模型。當您選擇模型時，您應該獨立評估租用戶資料隔離的需求，以及 SaaS 應用程式所需的已驗證權限原則結構。

若要在整個 SaaS 解決方案中強制執行共用租用戶識別碼的一致方式，最好在使用者註冊期間將識別碼對應至使用者的 SaaS 身分，如前所述。您可以將此對應提供給 SaaS 應用程式，方法是將其作為 IdP 的一部分或在外部資料來源 (例如 DynamoDB) 中進行維護。我們也建議您將共用原則存放區 ID 對應至使用者。雖然 ID 不會用作租用戶隔離的一部分，但這是一個很好的做法，因為它有助於 future 的變更。

下列範例顯示 API 端點如何為使用者傳送 JWTBob，以`Alice`及屬於不同承租人但與原則存放區 ID 共用原則存放區以進行授權的使`store-multi-tenant`用者。由於所有租用戶共用單一原則存放區，因此您不需要在 Token 或資料庫中維護原則存放區識別碼。由於所有承租人共用單一原則存放區識別碼，因此您可以提供 ID 做為環境變數，讓應用程式可用來呼叫原則存放區。



下列範例原則說明單一共用多租用戶原則設計範例。在此原則中，具有父項`MultiTenantApp::User`的主體`MultiTenantApp::RoleAdmin`具有檢視所有資源資料的權限。

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

由於單一原則存放區正在使用中，因此「已驗證權限」原則存放區必須確保與主體關聯的租用屬性符合與資源相關聯的租用屬性。這可以透過在原則存放區中包含下列原則來完成，以確保拒絕資源和主參與者上沒有相符租用屬性的所有授權要求。

```
forbid(
    principal,
    action,
    resource
)
```

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

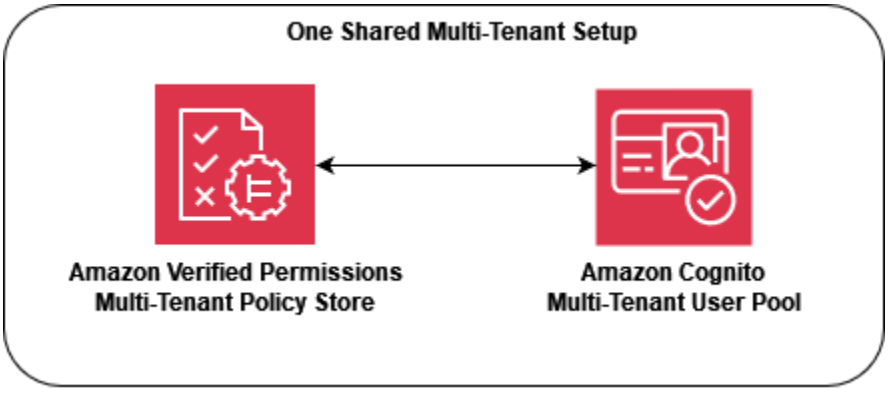
對於使用單一共用多租用戶原則存放區模型的授權要求，原則存放區 ID 是共用原則存放區的識別碼。在下列要求中，允許存取，因為她具有Role的Admin，且與資源和主參與者相關聯的Tenant屬性都UserAlice是TenantA。

```
{
   "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":[]
        }
    ]
  }
}
```

使用已驗證權限，可以 (但不是必要) 將 IdP 與原則存放區整合。此整合可讓原則明確參考識別身分存放區中的主體做為原則的主體。如需如何將 Amazon Cognito 整合為已驗證許可的 IdP 的詳細資訊，請參閱已驗證的許可文件和 Amazon Cognito 文件。

將原則存放區與 IdP 整合時，每個原則存放區只能使用一個身分識別來源。例如，如果您選擇將已驗證許可與 Amazon Cognito 整合，則必須鏡像用於已驗證許可政策存放區和 Amazon Cognito 使用者集區的租用戶隔離的策略。策略存放區和使用者集區也必須位於相同的位置 AWS 帳戶。



從操作和稽核的角度來看，單一共用的多租用戶原則存放區模型有一個缺點，即中記錄的活動 AWS CloudTrail需要更多相關的查詢來篩選出租用戶上的個別活動，因為每個記錄的 CloudTrail 呼叫都使用相同的原則存放區。在此案例中，將每個租用戶維度上的其他自訂指標記錄到 Amazon 會很有幫助，CloudWatch 以確保適當的可觀察性和稽核功能層級。

單一共用多租用戶原則存放區方法也需要密切注意「已驗證權限」配額，以確保它們不會干擾 SaaS 解決方案的作業。特別是，我們建議您監控每個區域每個帳戶配額的每秒IsAuthorized 請求數，以確保不會超出其限制。您可以要求增加此配額。

分層部署模型

透過建立分層部署模型，您可以將高優先順序的「企業層」租用戶與潛在數量較高的「標準層」客戶隔離開來。在此模型中，您可以針對每個層個別推出政策存放區中部署至政策的任何變更，這會將每個客戶層與其層以外的變更隔離開來。在階層式部署模型中，原則存放區通常會建立為每個層的初始基礎結構佈建的一部分，而不是在租用戶登入時進行部署。

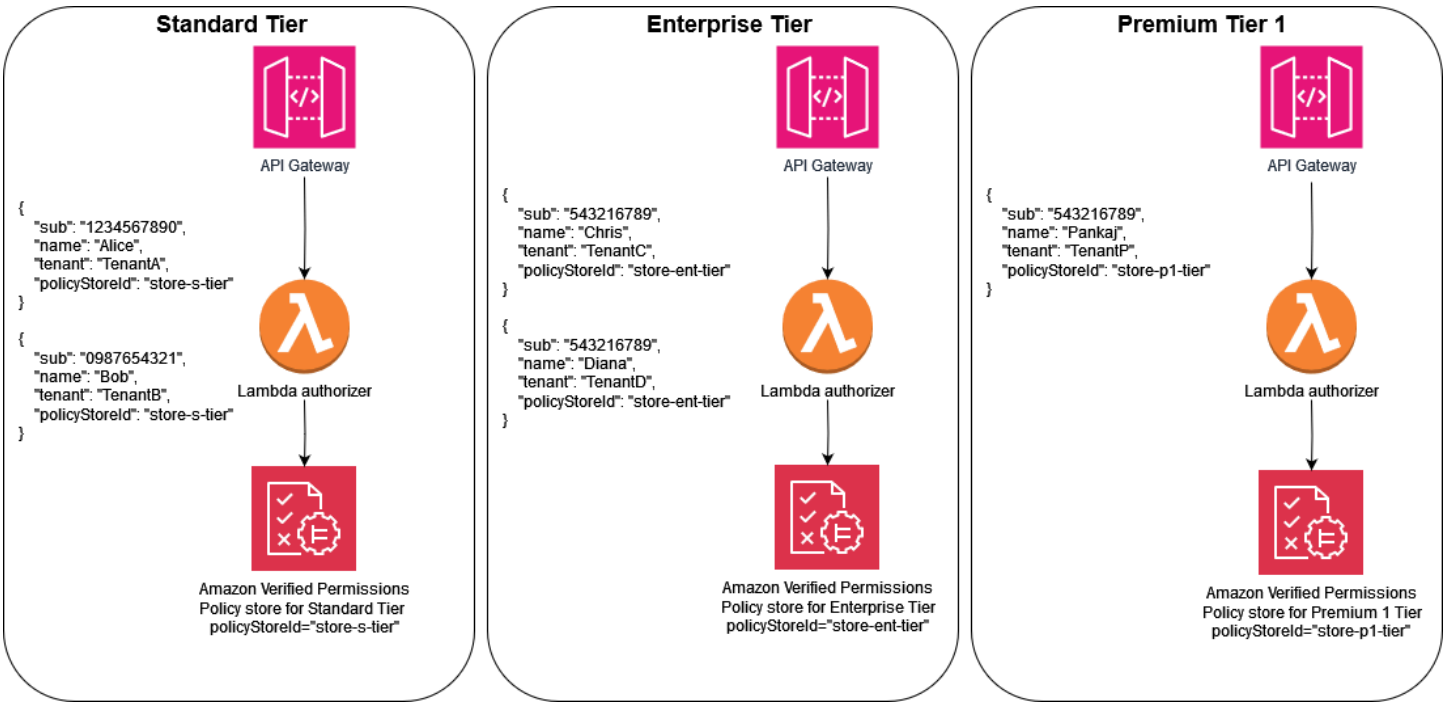
如果您的解決方案主要使用集區隔離模型，則可能需要額外的隔離或自訂。例如，您可以建立一個「進階層」，讓每個租用戶都能取得自己的租用戶層基礎結構，透過部署只有一個租用戶的集區執行個體來建立孤立的模型。這可能採用完全分開的「高級租戶 A」和「高級租戶 B」基礎結構的形式，包括政策存放區。這種方法為最高等級的客戶提供孤立的隔離模型。

在階層式部署模型中，每個原則存放區應遵循相同的隔離模型，儘管它是個別部署的。由於有多個原則存放區正在使用，因此您需要強制執行一致的方式來共用整個 SaaS 解決方案中與租用戶相關聯的原則存放區識別碼。與每個租用戶原則存放區模型一樣，最好在使用者註冊期間將租用戶識別碼對應至使用者的 SaaS 識別碼。

下圖顯示三個層級：Standard TierEnterprise Tier、和Premium Tier 1。每個層都會分別部署在自己的基礎結構中，並在該層中使用一個共用原則存放區。標準層和企業層包含多個租用戶。TenantA並TenantB位於和Standard Tier，TenantC且TenantD位於企業層中。

Premium Tier 1僅包含TenantP，因此您可以為高級租用戶提供服務，就像解決方案具有完全孤立的隔離模型，並提供自訂原則等功能。加入新的進階層客戶將導致建立Premium Tier 2基礎結構。

> **Note**
>
> 進階層中的應用程式、部署和租用戶上線與標準層和企業層相同。唯一的差別是進階層上線工作流程始於佈建新的層級基礎結構。


Standard Tier
API Gateway
{
"sub": "1234567890",
"name": "Alice",
"tenant": "TenantA",
"policyStoreId": "store-s-tier"
}
{
"sub": "0987654321",
"name": "Bob",
"tenant": "TenantB",
"policyStoreId": "store-s-tier"
}
Lambda authorizer
Amazon Verified Permissions
Policy store for Standard Tier
policyStoreId="store-s-tier"
Enterprise Tier
API Gateway
{
"sub": "543216789",
"name": "Chris",
"tenant": "TenantC",
"policyStoreId": "store-ent-tier"
}
{
"sub": "543216789",
"name": "Diana",
"tenant": "TenantD",
"policyStoreId": "store-ent-tier"
}
Lambda authorizer
Amazon Verified Permissions
Policy store for Enterprise Tier
policyStoreId="store-ent-tier"
Premium Tier 1
API Gateway
{
"sub": "543216789",
"name": "Pankaj",
"tenant": "TenantP",
"policyStoreId": "store-p1-tier"
}
Lambda authorizer
Amazon Verified Permissions
Policy store for Premium 1 Tier
policyStoreId="store-p1-tier"

OPA 多租戶設計考量

開放原則代理程式 (OPA) 是一項彈性的服務，可套用至許多需要應用程式才能做出原則和授權決策的使用案例。將 OPA 與多租戶 SaaS 應用程式搭配使用需要考量獨特的準則，以確保重要的 SaaS 最佳做法 (例如租用戶隔離) 仍然是 OPA 實作的一部分。這些準則包括 OPA 部署模式、租用戶隔離和 OPA 文件模型，以及租用戶上線。這些都會影響 OPA 的最佳設計，因為它與多租戶應用程式有關。

雖然本節中的討論側重於 OPA，但一般概念植根於隔離心態和它提供的指導。SaaS 應用程式必須始終將租用戶隔離視為其設計的一部分，而這項隔離的一般原則延伸到將 OPA 納入 SaaS 應用程式中。如果使用適當，OPA 可能是 SaaS 應用程式中如何強制執行隔離的關鍵部分。本節還引用了核心 SaaS 隔離模型，例如孤立的 SaaS 模型和集區 SaaS 模型。如需其他資訊，請參閱 AWS Well-Architected 的架構 SaaS Lens 中的核心隔離概念。

設計考量：

- 比較集中式和分散式部署模式
- 使用 OPA 文件模型進行租用戶隔離
- 租戶入職

比較集中式和分散式部署模式

您可以採用集中式或分散式部署模式來部署 OPA，而多租用戶應用程式的理想方法則取決於使用案例。如需這些模式的範例，請參閱本指南稍早的 < 在 API 上使用集中式 PDP 搭配 PEP > 和 < 在 API 上使用分散式 PDP 和 PEP > 一節。由於 OPA 可以部署為作業系統或容器中的精靈，因此可以透過多種方式實作，以支援多租用戶應用程式。

在集中式部署模式中，OPA 會部署為容器或精靈，其 RESTful API 可供應用程式中的其他服務使用。當服務需要 OPA 的決定時，會呼叫中央 OPA RESTful API 來產生這項決定。這種方法很容易部署和維護，因為只有 OPA 的單一部署。這種方法的缺點是它不提供任何機制來維護租用戶數據的分離。因為 OPA 只有單一部署，所以 OPA 決策中使用的所有租用戶資料 (包括 OPA 參照的任何外部資料) 都必須可供 OPA 使用。您可以使用這種方法維護租用戶資料隔離，但必須由 OPA 的原則和文件結構或對外部資料的存取來強制執行。集中式部署模式也需要較高的延遲，因為每個授權決策都必須對另一個服務進行 RESTful API 呼叫。

在分散式部署模式中，OPA 會與多租用戶應用程式的服務一起部署為容器或精靈。它可以部署為附屬容器或作為在操作系統上本地運行的守護進程。若要從 OPA 擷取決策，此服務會對本機 OPA 部署進行 RESTful API 呼叫。（因為 OPA 可以部署為 Go 包，因此您可以使用 Go 本地來檢索決策，而不是使用 RESTful API 調用。）與集中式部署模式不同，分散式模式需要更強大的工作來部署、維護和更

新，因為它存在於應用程式的多個區域。分散式部署模式的好處是能夠維護租用戶資料的隔離，特別是對於使用孤立 SaaS 模型的應用程式而言。在該租用戶專屬的 OPA 部署中，可以隔離承租人特定的資料，因為分散式模型中的 OPA 會與租用戶一起部署。此外，分散式部署模式的延遲時間遠低於集中式部署模式，因為每個授權決策都可以在本機上進行。

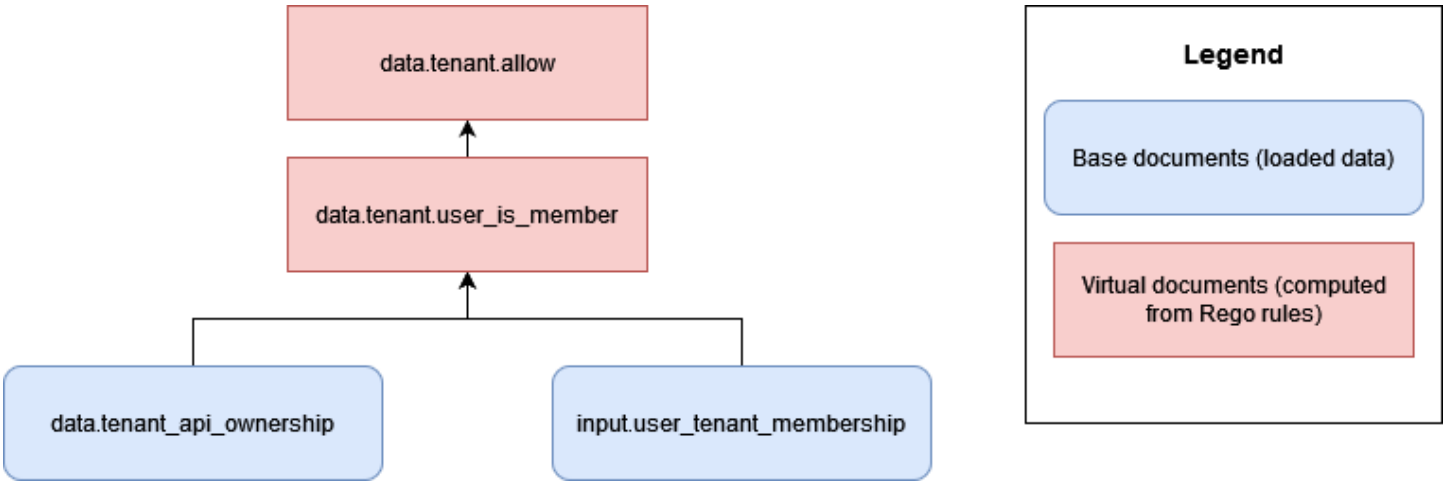
當您在多租用戶應用程式中選擇 OPA 部署模式時，請務必評估對應用程式最重要的準則。如果您的多租用戶應用程式對延遲很敏感，則分散式部署模式可提供更好的效能，但會犧牲更複雜的部署和維護。雖然您可以透過 DevOps 和自動化來管理部分複雜性，但與集中式部署模式相比，仍需要額外的努力。

如果您的多租用戶應用程式使用孤立的 SaaS 模型，您可以使用分散式 OPA 部署模式來模擬孤立的租用戶資料隔離方法。這是因為當 OPA 與每個承租人特定應用程式服務一起執行時，您可以自訂每個 OPA 部署，以僅包含與該承租人相關聯的資料。在集中式 OPA 部署模式中孤立 OPA 資料是不可能的。如果您使用集中式部署模式或分散式模式搭配集區 SaaS 模型，則必須在 OPA 文件模型中維護租用戶資料隔離。

使用 OPA 文件模型進行租用戶隔離

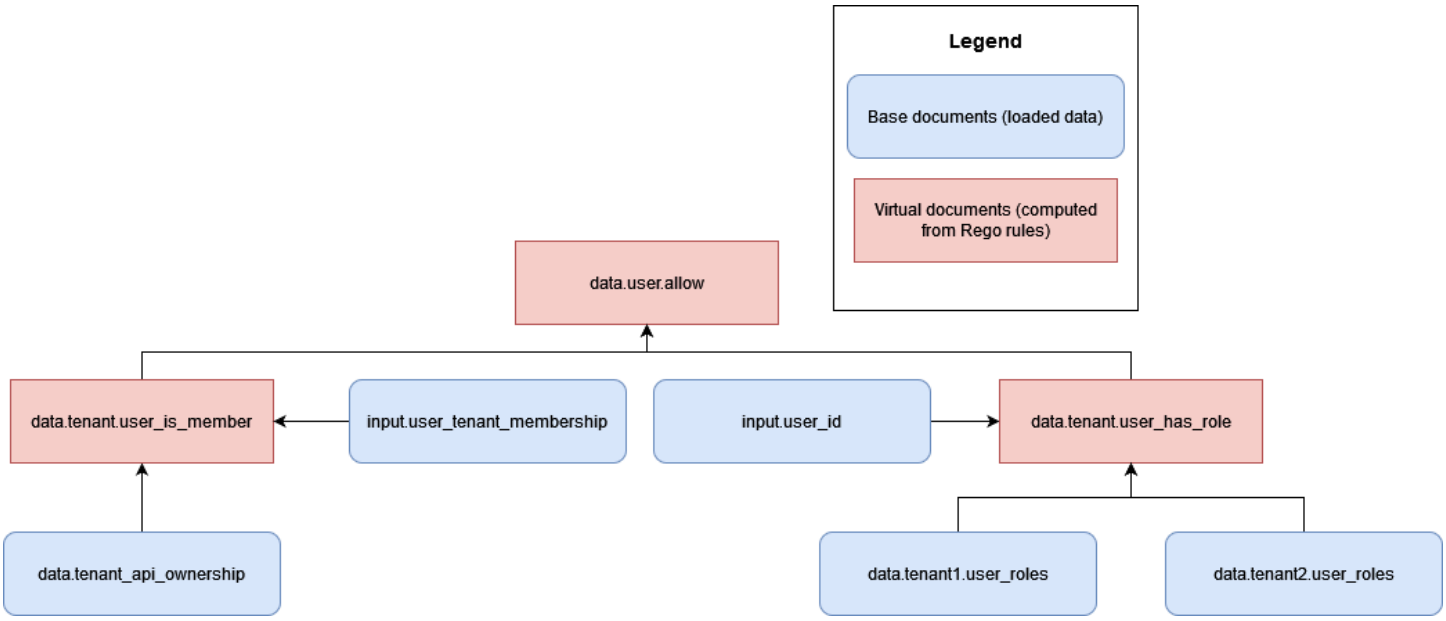
OPA 使用文件做出決策。這些文件可以包含承租人特定的資料，因此您必須考慮如何維護租用戶資料隔離。OPA 文件由基本文件和虛擬文件組成。基本文件包含來自外界的數據。這包括直接提供給 OPA 的資料、OPA 要求的相關資料，以及可能傳遞給 OPA 作為輸入的資料。虛擬文件是依原則計算，並包含 OPA 原則與規則。如需詳細資訊，請參閱 OPA 文件。

若要在 OPA 中為多租戶應用程式設計文件模型，您必須先考慮您需要在 OPA 中做出決定的基礎文件類型。如果這些基本文件包含承租人特定的資料，您必須採取措施以確保此資料不會意外暴露給跨租用戶存取。幸運的是，在許多情況下，在 OPA 中做出決定並不需要承租人特定的數據。下列範例顯示假設的 OPA 文件模型，該模型允許根據哪個租用戶擁有 API，以及使用者是否為租用戶的成員 (如輸入文件中所示) 存取 API。



在此方法中，OPA 無法存取任何承租人特定資料，除了有關哪些租用戶擁有 API 的資訊。在這種情況下，沒有關於 OPA 促進跨租戶存取，因為 OPA 用來做出存取決策的唯一資訊是使用者與租用戶的關聯，以及租用戶與 API 的關聯。您可以將此類型的 OPA 文件模型套用至孤立的 SaaS 模型，因為每個租用戶都擁有獨立資源的擁有權。

但是，在許多 RBAC 授權方法中，存在跨租戶暴露資訊的可能性。在下列範例中，假設性的 OPA 文件模型會根據使用者是否為租用戶的成員，以及使用者是否具有存取 API 的正確角色來存取 API。



此模型會帶來跨租戶存取的風險，因為 OPA 現在`data.tenant2.user_roles`必須讓 OPA 存取多個租戶的角色`data.tenant1.user_roles`和權限，才能做出授權決策。若要維護租用戶隔離和角色對應的隱私權，此資料不應位於 OPA 內。RBAC 資料應該位於外部資料來源 (例如資料庫) 中。此外，不應使用 OPA 將預先定義的角色對應至特定權限，因為這會讓租用戶難以定義自己的角色和權限。它還使您的授權邏輯變得剛性，並且需要不斷更新。如需如何將 RBAC 資料安全地整合至 OPA 決策程序的指引，請參閱本指南稍後的租用戶隔離和資料隱私權建議一節。

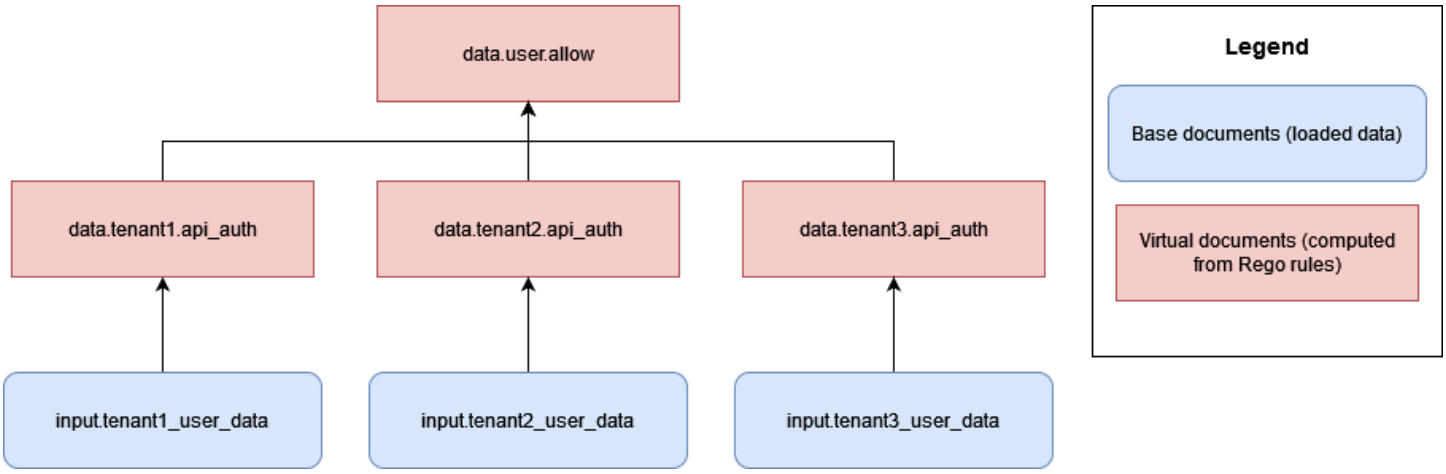
您可以輕鬆地在 OPA 中維護租用戶隔離，方法是不將任何承租人特定資料儲存為非同步基礎文件。非同步基本文件是儲存在記憶體中的資料，可在 OPA 中定期更新。其他基本文件 (例如 OPA 輸入) 會同步傳遞，而且只能在決策時使用。例如，將承租人特定資料作為 OPA 輸入的一部分提供給查詢並不構成違反租用戶隔離，因為該資料只能在做出決策的過程中同步提供。

租戶入職

OPA 文件的結構必須允許直接的租用戶上線，而不會引入繁瑣的需求。您可以使用套件組織 OPA 文件模型層次結構中的虛擬文件，而且這些套件可以包含許多規則。當您規劃多租用戶應用程式的 OPA

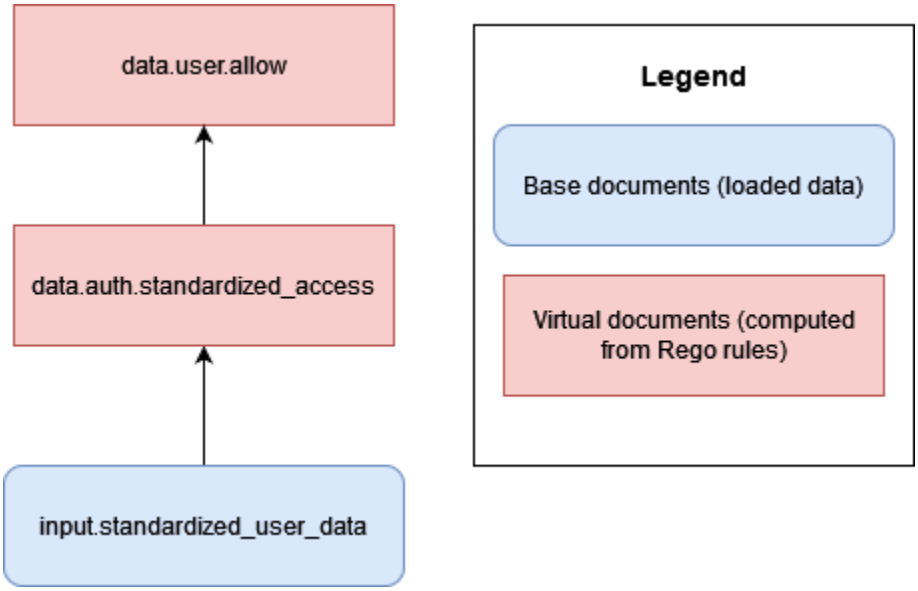
文件模型時，請先判斷 OPA 做出決定所需的資料。您可以提供資料做為輸入、預先載入 OPA，或在決定時間或定期從外部資料來源提供資料。如需有關搭配 OPA 使用外部資料的詳細資訊，請參閱本指南稍後的在 OPA 中擷取 PDP 的外部資料一節。

確定在 OPA 中做出決定所需的資料之後，請考慮如何實作組織為套件的 OPA 規則，以便使用該資料做出決策。例如，在孤立的 SaaS 模型中，每個租用戶對於如何做出授權決策可能有獨特的需求，您可以實作承租人特定的 OPA 規則套件。



此方法的缺點是，您必須針對每個租用戶新增至 SaaS 應用程式的每個租用戶新增一組新的 OPA 規則。這很麻煩且難以擴展，但可能是不可避免的，具體取決於您租用戶的需求。

或者，在集區 SaaS 模型中，如果所有租用戶都根據相同的規則做出授權決策，並使用相同的資料結構，您可以使用具有一般適用規則的標準 OPA 套件，以便更輕鬆地將租用戶上線並擴展 OPA 實作。



在可能的情況下，我們建議您使用一般化的 OPA 規則和套件 (或虛擬文件)，根據每個租用戶提供的標準化資料做出決策。這種方法使 OPA 易於擴展，因為您只更改為每個租戶提供給 OPA 的數據，而不

是 OPA 如何通過其規則提供決策。只有當個別租戶需要獨特的決定或必須向 OPA 提供與其他租戶不同的數據時，才需要引入 rules-per-tenant 模型。

DevOps、監視、記錄和擷取 PDP 的資料

在這個提議的授權範例中，政策集中在授權服務中。這種集中化是故意的，因為本指南中討論的設計模型的其中一個目標是實現原則解耦，或從應用程式中的其他元件移除授權邏輯。Amazon 驗證許可和開放政策代理程式 (OPA) 都提供機制，以便在需要變更授權邏輯時更新政策。

在已驗證許可的情況下， AWS SDK 提供程式設計方式更新政策的機制 (請參閱 Amazon 驗證許可 API 參考指南)。使用 SDK，您可以根據需要推送新政策。此外，由於「已驗證權限」是受管理的服務，因此您不必管理、設定或維護控制平面或代理程式即可執行更新。不過，我們建議您使用持續整合和持續部署 (CI/CD) 管道來管理「已驗證權限」原則存放區的部署，以及使用 SDK 的原則更新。 AWS

驗證權限可讓您輕鬆存取可觀察性功能。它可以設定為記錄對 Amazon 日 CloudWatch 誌群組 AWS CloudTrail、Amazon 簡單儲存服務 (Amazon S3) 儲存貯體或 Amazon Data Firehose 交付串流的所有存取嘗試，以便快速回應安全事件和稽核請求。此外，您可以透過監視已驗證權限服務的健全狀況 AWS Health Dashboard。由於「已驗證權限」是受管理的服務，因此其健全狀況由維護 AWS，而且您可以使用其他 AWS 受管理的服務來設定可觀察性功能。

在 OPA 的情況下，REST API 提供了以程式設計方式更新原則的方法。您可以將 API 設定為從建立的位置提取新版本的原則集，或視需要推送原則。此外，OPA 還提供基本探索服務，讓新的代理程式可以動態設定，並透過散佈探索服務包的控制平面集中管理。OPA 的控制平面必須由 OPA 操作員設置和配置。 ） 我們建議您建立健全的 CI/CD 管道來進行版本控制、驗證和更新原則，無論原則引擎是否為已驗證權限、OPA 或其他解決方案。

對於 OPA，控制平面也提供用於監視和稽核的選項。您可以將包含 OPA 授權決策的記錄匯出至遠端 HTTP 伺服器，以進行記錄彙總。這些決策日誌對於審計目的而言是非常寶貴的。

如果您正在考慮採用授權模式，而存取控制決策與您的應用程式分離，請確定您的授權服務具有有效的監控、記錄和 CI/CD 管理功能，以便上架新 PDP 或更新原則。

主題

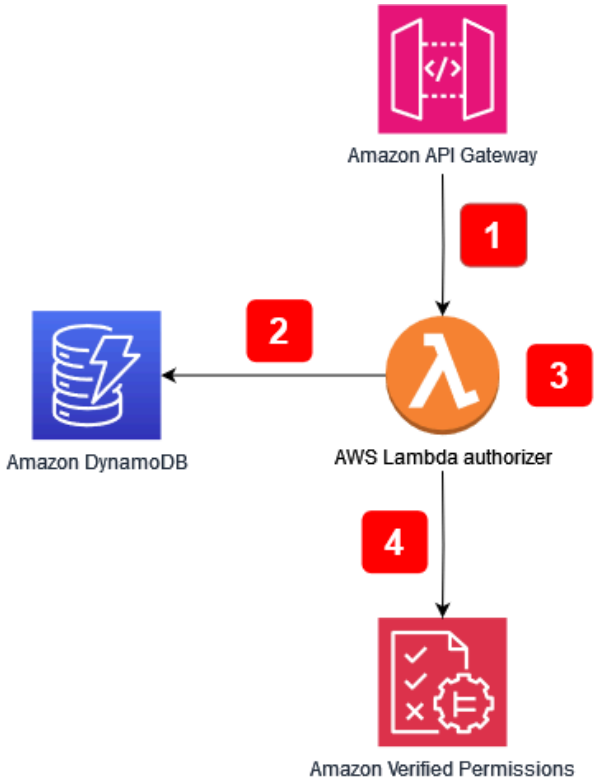
- 在 Amazon 驗證許可中檢索 PDP 的外部數據
- 擷取 OPA 中 PDP 的外部資料
- 對租戶隔離和數據隱私的建議

在 Amazon 驗證許可中檢索 PDP 的外部數據

Amazon 驗證許可不支援擷取 PDP 的外部資料，但它可以將使用者提供的資料存放為其結構描述的一部分。與 OPA 一樣，如果授權決策的所有數據都可以作為授權請求的一部分提供，或作為請求一部

分傳遞的 JSON Web 令牌（JWT）的一部分，則不需要額外的配置。不過，您可以透過授權要求，將外部來源的其他資料提供給「已驗證的權限」，作為呼叫「已驗證權限」之應用程式授權者服務的一部分。例如，應用程式的授權者服務可以查詢外部來源 (例如 DynamoDB 或 Amazon RDS) 以取得資料，然後這些服務可以在授權請求中包含外部提供的資料。

下圖顯示如何擷取其他資料並併入已驗證權限授權要求的範例。您可能需要使用此方法擷取資料 (例如 RBAC 角色對應)、擷取與資源或主參與者相關的其他屬性，或是資料位於應用程式的不同部分且無法透過身分識別提供者 (IdP) 權杖提供。



申請流程：

1. 應用程式會接收 API 呼叫至 Amazon API Gateway，並將呼叫轉送給 AWS Lambda 授權者。
2. Lambda 授權者呼叫 Amazon DynamoDB，以擷取有關提出請求之主體的其他資料。
3. Lambda 授權者會將其他資料併入已驗證許可的授權請求中。
4. Lambda 授權者向已驗證許可發出授權請求，並接收授權決策。

該圖包括稱為 Lambda 授權者的 Amazon API Gateway 的功能。雖然此功能可能不適用於其他服務或應用程式所提供的 API，但是您可以複製使用授權者的一般模型，以擷取其他資料，以便在多種使用案例中併入「已驗證權限」授權要求中。

擷取 OPA 中 PDP 的外部資料

對於 OPA，如果授權決策所需的所有數據都可以作為輸入提供，或作為查詢組件傳遞的 JSON Web 令牌（JWT）的一部分，則不需要額外的配置。（將 JWT 和 SaaS 上下文數據作為查詢輸入的一部分傳遞給 OPA 相對簡單。）OPA 可以在所謂的重載輸入方法中接受任意 JSON 輸入。如果 PDP 需要的數據超出了可以作為輸入或 JWT 包含的數據，OPA 提供了幾個選項來檢索這些數據。其中包括捆綁，發送數據（複製）和動態數據檢索。

OPA 捆綁

OPA 捆綁功能支持以下外部數據檢索過程：

1. 原則強制執行點 (PEP) 要求授權決定。
2. OPA 會下載新的規則服務包，包括外部資料。
3. 捆綁服務複製來自數據源的數據。

當您使用捆綁功能時，OPA 會定期從集中式服務下載策略和數據服務包。（OPA 不提供捆綁服務的實現和設置。）從套裝軟體服務提取的所有規則和外部資料都會儲存在記憶體中。如果外部資料大小太大而無法儲存在記憶體中，或資料變更過頻繁，則此選項將無法運作。

如需組合功能的詳細資訊，請參閱 OPA 文件。

OPA 複寫 (推送資料)

OPA 複寫方法支援外部資料擷取的下列程序：

1. PEP 請求授權決定。
2. 資料複製器會將資料推送至 OPA。
3. 資料複製器會從資料來源複製資料。

在這種捆綁方法的替代方法中，數據被推送到，而不是由 OPA 定期提取。(OPA 不提供複製器的實作和設定。) 推送方法與捆綁方法具有相同的數據大小限制，因為 OPA 將所有數據存儲在內存中。推送選項的主要優點是您可以使用增量來更新 OPA 中的資料，而不是每次都取代所有外部資料。這使得推送選項更適合經常變更的資料集。

如需複製選項的詳細資訊，請參閱 OPA 說明文件。

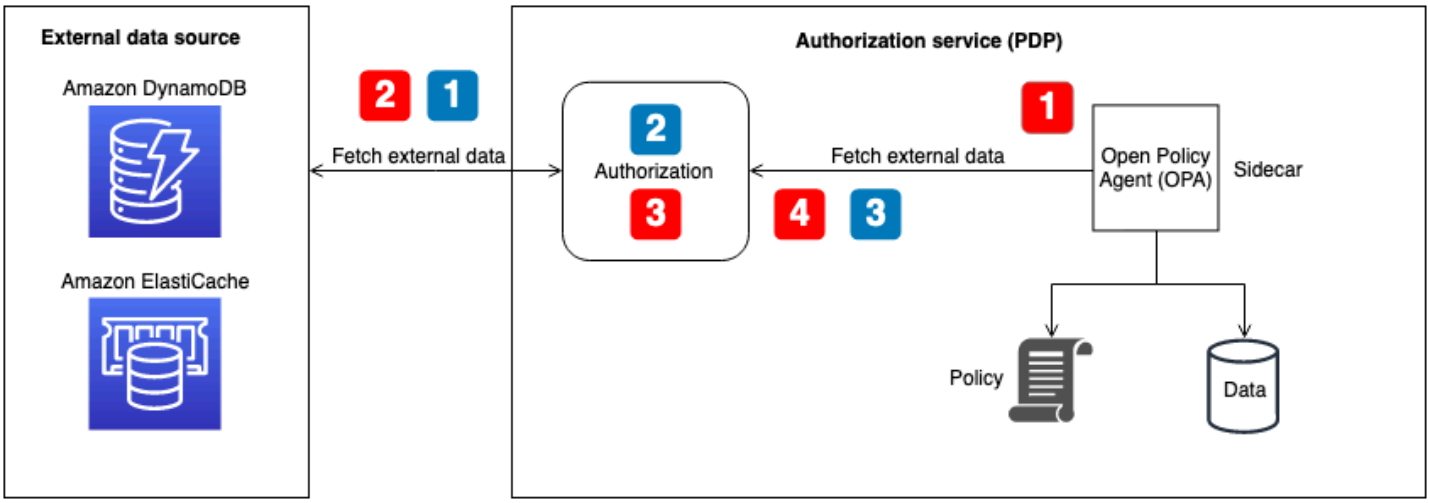
OPA 動態數據檢索

如果要擷取的外部資料太大而無法快取到 OPA 的記憶體中，則可在評估授權決策期間從外部來源動態提取資料。當您使用這種方法時，資料永遠是最新的。這種方法有兩個缺點：網絡延遲和可訪問性。目前，OPA 只能透過 HTTP 要求在執行階段擷取資料。如果前往外部資料來源的呼叫無法以 HTTP 回應的形式傳回資料，則需要自訂 API 或其他機制，才能將此資料提供給 OPA。由於 OPA 只能透過 HTTP 請求擷取資料，而且擷取資料的速度相當關鍵，因此建議您盡可能使用 Amazon DynamoDB AWS 服務 等來保存外部資料。

如需有關提取方法的詳細資訊，請參閱 OPA 文件。

使用授權服務與 OPA 實施

當您使用捆綁，複製或動態提取方法來獲取外部數據時，我們建議授權服務促進此交互。這是因為授權服務可以擷取外部資料，並將其轉換為 JSON，以便 OPA 做出授權決策。下圖顯示了授權服務如何與這三個外部數據檢索方法的功能。



擷取 OPA 流程的外部資料 — 在決策時束或動態資料擷取 (圖中以紅色編號說明)：

1. OPA 會呼叫授權服務的本機 API 端點，該端點會在授權決策期間設定為服務包端點或用於動態資料擷取的端點。
2. 授權服務會查詢或呼叫外部資料來源以擷取外部資料。(對於套裝軟體端點，此資料也應包含 OPA 策略和規則。 套裝軟體更新會取代 OPA 快取中的所有項目 (包括資料和原則)。
3. 授權服務會對傳回的資料執行任何必要的轉換，以將其轉換為預期的 JSON 輸入。
4. 資料會傳回至 OPA。它被緩存在內存中用於捆綁配置，並立即用於動態授權決策。

擷取 OPA 流程的外部資料 — 複製器 (圖中以藍色編號說明)：

1. 複製器 (授權服務的一部分) 會呼叫外部資料來源，並擷取要在 OPA 中更新的任何資料。這可能包括策略、規則和外部資料。此呼叫可以按照設定的頻率進行，也可以在回應外部來源中的資料更新時發生。
2. 授權服務會對傳回的資料執行任何必要的轉換，以將其轉換為預期的 JSON 輸入。
3. 授權服務會呼叫 OPA 並將資料快取至記憶體中。授權服務可以選擇性地更新資料、原則和規則。

對租戶隔離和數據隱私的建議

上一節提供了幾種透過 OPA 和 Amazon 驗證許可使用外部資料的方法，以協助做出授權決策。在可能的情況下，我們建議您使用多載輸入方法將 SaaS 內容資料傳遞給 OPA，以做出授權決策，而不是將資料儲存在 OPA 的記憶體中。此使用案例不適用於 AWS Cloud Map，因為它不支援在服務中儲存外部資料。

在基於角色的訪問控制（RBAC）或 RBAC 和基於屬性的訪問控制（ABAC）混合模型中，僅由授權請求或查詢提供的數據可能不足，因為必須參考角色和權限來做出授權決策。若要維護租用戶隔離和角色對應的隱私權，此資料不應位於 OPA 內。RBAC 資料應該位於外部資料來源 (例如資料庫) 中，或者應該從 IdP 作為 JWT 中宣告的一部分傳遞。在已驗證的權限中，RBAC 資料可以作為每個租用戶原則存放區模型中原則和結構描述的一部分進行維護，因為每個租用戶都有自己的邏輯分隔原則存放區。不過，在單一共用多租用戶原則存放區模型中，角色對應資料不應位於已驗證的權限內，以維護租用戶隔離。

此外，不應使用 OPA 和已驗證的權限來將預先定義的角色對應至特定權限，因為這會讓租用戶難以定義自己的角色和權限。它還使您的授權邏輯變得剛性，並且需要不斷更新。此準則的例外狀況是「已驗證權限」中的每個租用戶原則存放區模型，因為此模型允許每個承租人擁有自己的原則，這些原則可以根據每個租用戶個別進行評估。

Amazon Verified Permissions

「已驗證的權限」可以儲存潛在私人 RBAC 資料的唯一位置是在結構描述中。這在每個租用戶原則存放區模型中是可以接受的，因為每個租用戶都有自己的邏輯分離原則存放區。不過，它可能會危害單一共用多租用戶原則存放區模型中的租用戶隔離。如果需要此資料才能做出授權決策，則應從 DynamoDB 或 Amazon RDS 等外部來源擷取資料，並併入已驗證許可授權請求中。

OPA

使用 OPA 來維護 RBAC 資料的隱私權和租用戶隔離的安全方法，包括使用動態資料擷取或複寫來取得外部資料。這是因為您可以使用上圖所示的授權服務，僅提供承租人特定或使用者特定的外部資料，以便做出授權決策。例如，您可以使用複寫器在使用者登入時，將 RBAC 資料或權限矩陣提供給 OPA 快取，並根據輸入資料中提供的使用者參考資料。您可以使用類似的方法與動態提取的數據來僅檢索相關數據進行授權決策。此外，在動態資料擷取方法中，此資料不需要在 OPA 中快取。捆綁方法不如維護租用戶隔離的動態檢索方法那麼有效，因為它會更新 OPA 緩存中的所有內容，並且無法處理精確的更新。捆綁模型仍然是更新 OPA 政策和非 RBAC 數據的好方法。

最佳實務

本節列出了本指南中的一些高級要點。有關每個點的詳細討論，請點擊相應部分的鏈接。

選取適用於您應用程式的存取控制模型

本指南討論數種存取控制模型。根據您的應用程序和業務需求，您應該選擇適合您的模型。考慮如何使用這些模型來滿足您的存取控制需求，以及您的存取控制需求如何演變，需要變更您選取的方法。

實施一個 PDP

原則決策點 (PDP) 可以被指定為策略或規則引擎。此元件負責套用原則或規則，並傳回是否允許特定存取的決定。PDP 允許應用程序代碼中的授權邏輯卸載到單獨的系統。這可以簡化應用程序代碼。它還提供了一個 easy-to-use 冪等接口，用於對 API，微服務，前端後端（BFF）層或任何其他應用程序組件進行授權決策。PDP 可用於在整個應用程式中一致地強制執行租用需求。

為應用程式中的每個 API 實作 PEP

原則強制執行點 (PEP) 的實作需要決定應該在應用程式中執行存取控制的位置。第一步，在您的應用程序中找到可以合併 PEP 的點。決定在何處新增 PEP 時，請考慮以下原則：

如果應用程序公開了一個 API，則應該對該 API 進行授權和訪問控制。

考慮使用 Amazon 驗證許可或 OPA 作為 PDP 的政策引擎

Amazon 驗證許可比自訂政策引擎具有優勢。它是可擴充、精細的權限管理和授權服務，適用於您建置的應用程式。它支援以高階宣告式開放原則語言 Cedar 撰寫原則。因此，使用已驗證權限實作原則引擎所需的開發工作比實作您自己的解決方案更少。此外，「已驗證的權限」是完全受管的，因此您不必管理基礎結構。

開放原則代理程式 (OPA) 的優點優於自訂原則引擎。OPA 及其使用 Rego 進行原則評估提供彈性、預先建置的原則引擎，可支援以高階宣告式語言撰寫原則。這會讓實作原則引擎所需的工作量遠低於建置您自己的解決方案。此外，OPA 正在迅速成為一個受到良好支持的授權標準。

實作 OPA 用於 DevOps、監控和記錄的控制平面

因為 OPA 不提供透過原始檔控制來更新和追蹤授權邏輯變更的方法，所以我們建議您實作控制平面來執行這些功能。這樣可以更輕鬆地將更新分發給 OPA 代理程式，特別是如果 OPA 在分散式系統中操

作，這將減少使用 OPA 的管理負擔。此外，控制平面可用來收集記錄以進行彙總，以及監視 OPA 代理程式的狀態。

在已驗證的權限中設定記錄和可觀察性功能

驗證權限可讓您輕鬆存取可觀察性功能。您可以將服務設定為記錄對 Amazon 日 CloudWatch 誌群組 AWS CloudTrail、S3 儲存貯體或 Amazon Data Firehose 交付串流的所有存取嘗試，以便快速回應安全事件和稽核請求。此外，您可以透過監視服務的健全狀況 AWS Health Dashboard。由於「已驗證權限」是受管理的服務，因此其健全狀況由維護 AWS，而且您可以使用其他 AWS 受管理的服務來設定其可觀察性功能。

使用 CI/CD 管線在已驗證的權限中佈建和更新原則存放區和原則

驗證權限是受管理的服務，因此您不必管理、設定或維護控制平面或代理程式即可執行更新。不過，我們仍建議您使用持續整合和持續部署 (CI/CD) 管道，透過使用 SDK 來管理已驗證權限原則存放區和原則更新的 AWS部署。這項工作可以消除手動工作，並減少當您對「已驗證權限」資源進行變更時，操作員出錯的可能性。

判斷授權決策是否需要外部資料，並選取適合的模型

如果 PDP 可以僅根據 JSON Web 令牌（JWT）中包含的數據進行授權決策，則通常不需要導入外部數據以幫助做出這些決策。如果您使用已驗證權限或 OPA 作為 PDP，它也可以接受作為請求一部分傳遞的其他輸入，即使這些數據不包含在 JWT 中。對於已驗證的權限，您可以為其他資料使用上下文參數。對於 OPA，您可以使用 JSON 資料做為多載輸入。如果使用 JWT，上下文或多載輸入方法通常比在另一個源中維護外部數據要容易得多。如果需要更複雜的外部資料來進行授權決策，OPA 會提供數種模型來擷取外部資料，而「已驗證的權限」則可以透過參照授權服務的外部來源來補充其授權要求中的資料。

常見問答集

本節提供有關在多租戶 SaaS 應用程式中實作 API 存取控制和授權的常見問題的解答。

問：授權和身份驗證有什麼區別？

答：身份驗證是驗證用戶是誰的過程。授權會授與使用者存取特定資源的權限。

問：SaaS 應用程式中的授權和租用戶隔離之間有何差異？

答：租用戶隔離是指 SaaS 系統中使用的明確機制，以確保每個租用戶的資源（即使在共用基礎架構上操作）都是隔離的。多租戶授權是指輸入操作的授權，並防止它們在錯誤的承租人上實施。假設用戶可以通過身份驗證和授權，但仍然可以訪問另一個租戶的資源。關於驗證和授權的任何內容都不一定會阻止此存取，但需要租用戶隔離才能達到此目標。如需有關這兩個概念的詳細資訊，請參閱 AWS SaaS 架構基礎知識白皮書中的租用戶隔離討論。

問：為什麼我需要考慮 SaaS 應用程式的租用戶隔離？

答：SaaS 應用程序有多個租戶。租用戶可以是客戶組織，也可以是使用該 SaaS 應用程式的任何外部實體。視應用程式的設計方式而定，這表示租用戶可能正在存取共用 API、資料庫或其他資源。維護承租人隔離非常重要，也就是說，建構能夠嚴格控制資源存取，並封鎖任何嘗試存取其他承租人資源的嘗試，以防止一個承租人的使用者存取另一個承租人的私人資訊。SaaS 應用程式通常是為了確保在整個應用程式中維護租用戶隔離，而且租用戶只能存取自己的資源。

問：為什麼需要存取控制模型？

A. 存取控制模型可用來建立一致的方法，以決定如何授予應用程式中資源的存取權。這可以透過將角色指派給與企業邏輯密切一致的使用者來完成，也可以根據其他內容屬性 (例如一天中的時間或使用者是否符合預先定義的條件) 來完成。存取控制模型構成應用程式在做出授權決策以決定使用者權限時所使用的基本邏輯。

問：我的應用程式是否需要 API 存取控制？

答：是的。API 應始終驗證呼叫者是否具有適當的訪問權限。普遍的 API 存取控制還可確保僅根據租用戶授予存取權，以便維護適當的隔離。

問：為什麼建議使用政策引擎或 PDP 進行授權？

答：原則決策點 (PDP) 可讓應用程式程式碼中的授權邏輯卸載至個別的系統。這可以簡化應用程序代碼。它還提供了一個 easy-to-use 冪等接口，用於對 API，微服務，前端後端（BFF）層或任何其他應用程序組件進行授權決策。

問：什麼是 PEP？

答：原則強制執行點 (PEP) 負責接收傳送至 PDP 進行評估的授權要求。PEP 可以位於必須保護資料和資源的應用程式中的任何位置，或套用授權邏輯的位置。與 PDP 相比，PEP 相對簡單。PEP 僅負責請求和評估授權決策，不需要將任何授權邏輯納入其中。

問：我應該如何在 Amazon 驗證許可和 OPA 之間進行選擇？

答：若要在「已驗證的權限」和「開放原則代理程式」(OPA) 之間進行選擇，請務必記住您的使用案例和您的獨特需求。驗證權限提供完全受控的方式，可定義精細的權限、跨應用程式稽核權限，並集中管理應用程式的原則管理系統，同時以毫秒的處理方式滿足應用程式延遲需求。OPA 是開放原始碼的一般用途原則引擎，也可協助您統一整合應用程式堆疊中的原則。若要執行 OPA，您需要將它裝載在您的 AWS 環境中，通常使用容器或 AWS Lambda 函式。

已驗證的權限使用開放原始碼 Cedar 原則語言，而 OPA 則使用 Rego。因此，熟悉這些語言之一可能會讓您選擇該解決方案。不過，我們建議您閱讀這兩種語言的相關資訊，然後再從您嘗試解決的問題中找出最適合您使用案例的解決方案。

問：已驗證許可和 OPA 是否有開放原始碼替代方案？

答：有幾個開放原則系統類似於「已驗證的權限」和「開放原則代理程式」(OPA)，例如「通用運算式語言」(CEL)。本指南著重於「已驗證的權限」(作為可擴充的權限管理和精細授權服務) 和 OPA (已廣泛採用、記錄和適應許多不同類型的應用程式和授權需求)。

問：是否需要撰寫授權服務才能使用 OPA，還是可以直接與 OPA 互動？

答：您可以直接與 OPA 互動。本指南內容中的授權服務是指將授權決策請求轉換為 OPA 查詢的服務，反之亦然。如果您的應用程式可以直接查詢和接受 OPA 回應，就不需要引入這個額外的複雜性。

問：如何監控 OPA 代理程式的正常運作時間和稽核目的？

答：OPA 提供記錄和基本執行時間監控，但其預設組態可能不足以進行企業部署。如需詳細資訊，請參閱本指南前面的 < 、監視和記錄 > 一節。DevOps

問：如何監控已驗證的許可以用於正常運作時間和稽核目的？

答：已驗證的權限是 AWS 受管理的服務，可透過 AWS Health Dashboard. 此外，經過驗證的許可能能夠記錄到 AWS CloudTrail Amazon CloudWatch 日誌、Amazon S3 和 Amazon 數據 Firehose 件。

問：我可以使用哪些作業系統和 AWS 服務來執行 OPA？

答：您可以在 macOS、視窗和 Linux 上執行 OPA。OPA 代理程式可以在 Amazon 彈性運算雲端 (Amazon EC2) 代理程式以及容器化服務，例如 Amazon Elastic Container Service (Amazon ECS) 和 Amazon Elastic Kubernetes Service (亞馬遜 EKS) 上進行設定。

問：我可以使用哪些作業系統和 AWS服務來執行已驗證的許可？

答：已驗證的權限是 AWS 受管理的服務，由操作 AWS。除了向服務發出授權請求的能力外，不需要其他配置、安裝或託管即可使用「已驗證的權限」。

問：是否可以在 AWS Lambda上執行 OPA？

答：您可以在 Lambda 上以 Go 程式庫的形式執行 OPA。如需如何針對 API Gateway Lambda 授權者執行此作業的詳細資訊，請參閱 AWS 部落格文章使用開放原則代理程式建立自訂 Lambda 授權者。

問：我應該如何決定分散式 PDP 和集中式 PDP 方法？

答：這取決於您的應用。它很可能會根據分散式和集中式 PDP 模型之間的延遲差異來確定。我們建議您建立概念驗證並測試應用程式的效能，以驗證您的解決方案。

問：是否可以將 OPA 用於 API 以外的使用案例？

答：是的。OPA 文檔提供了庫伯尼特人，特使，碼頭工人，卡夫卡，SSH 和須藤和地形的示例。此外，OPA 可以使用 Rego 部分規則將任意 JSON 回應傳回查詢。根據查詢，OPA 可以用來回答 JSON 回應的許多問題。

問：是否可以針對 API 以外的使用案例使用已驗證的許可？

答：是的。已驗證的權限可以為收到的任何授權請求提供`ALLOW`或`DENY`回應。已驗證的權限可以為需要授權決策的任何應用程式或服務提供授權回應。

問：是否可以使用 IAM 政策語言在已驗證的許可中建立政策？

答：不是。 您必須使用 Cedar 政策語言來編寫政策。Cedar 的設計旨在支援客戶應用程式資源的許可管理，而 AWS Identity and Access Management (IAM) 政策語言演進為支援 AWS 資源的存取控制。

後續步驟

透過採用標準化、與語言無關的方法來制定授權決策，可以解決多租用戶 SaaS 應用程式的授權和 API 存取控制的複雜性。這些方法納入了原則決策點 (PDP) 和原則強制執行點 (PDP)，以彈性且普遍的方式強制存取。多種存取控制方法 [例如角色型存取控制 (RBAC)、屬性型存取控制 (ABAC) 或兩者的組合] 可以整合到一個一致的存取控制策略。從應用程式中移除授權邏輯，可消除在應用程式程式碼中包含臨時解決方案來解決存取控制的額外負荷。本指南中討論的實作和最佳實務旨在通知和標準化多租用戶 SaaS 應用程式中授權和 API 存取控制的實作方法。您可以使用本指引作為收集資訊並為您的應用程式設計強大的存取控制和授權系統的第一步。接下來的步驟：

- 檢閱您的授權和租用戶隔離需求，並為您的應用程式選取存取控制模型。
- 使用 Amazon 驗證許可或開放政策代理程式 (OPA)，或撰寫您自己的自訂政策引擎，建立用於測試的概念證明。
- 識別應用程式中應實作 PEP 的 API 和位置。

資源

參考

- Amazon 驗證許可文檔（AWS 文檔）
- 如何使用 Amazon 驗證許可進行授權（AWS 博客文章）
- 使用 Amazon 驗證許可為 ASP.NET 核心應用程式實作自訂授權原則提供者 (AWS 部落格文章)
- 使用 Amazon 驗證許可，透過 PBAC 管理角色和權利 (部落格文章)AWS
- 透過每個租用戶政策存放區使用 Amazon 驗證許可的 SaaS 存取控制 (AWS 部落格文章)
- OPA 官方文件
- 為什麼企業必須擁抱最近畢業的 CNCF 項目 — 開放政策代理 (福布斯文章由加納基拉姆 MSV, 2021 年 2 月 8 日)
- 使用開放式原則代理程式建立自訂 Lambda 授權者 (AWS 部落格文章)
- 透過開放原則代理程式使用 AWS Cloud Development Kit 實現原則即程式碼 (AWS 部落格文章)
- 雲端治理 AWS 與合規性政策即程式碼 (AWS 部落格文章)
- 在 Amazon EKS 上使用開放式政策代理程式 (AWS 部落格文章)
- 使用開放式政策代理程式、Amazon 和 AWS Lambda(AWS 部落格文章)，適用於 Amazon EventBridge ECS 的合規即程式碼
- 針對 Kubernetes 的原則式對策 — 第 1 部分 (部落格文章)AWS
- 使用 API Gateway Lambda 授權者 (AWS 文件)

工具

- 雪松遊樂場（用於在瀏覽器中測試雪松）
- 雪松 Github 存儲庫
- 雪松語言參考
- 雷戈遊樂場（用於在瀏覽器中測試 Rego）
- OPA 儲存庫 GitHub

合作夥伴

- Identity and Access Management 合作夥伴

- 應用程式安全性
- 雲端治理夥伴
- 安全與合規合作夥伴
- 安全性作業與自動化合作
- 安全工程合作夥

文件歷史紀錄

下表描述了本指南的重大變更。如果您想收到有關未來更新的通知，可以訂閱 RSS 摘要。

變更	描述	日期
新增 Amazon 驗證許可的詳細資訊和範例	新增使用 Amazon 驗證許可實作 PDP 的詳細討論、範例和程式碼。新章節包括： • 通過使用 Amazon 驗證許可實施 PDP • Amazon 驗證許可的設計模型 • Amazon 驗證許可多租用戶設計考量 • 在 Amazon 驗證許可中檢索 PDP 的外部數據	2024年5月28日
澄清的信息	在 API 設計模型上使用 PEP 澄清了分佈式 PDP。	2024 年 1 月 10 日
已新增有關新 AWS 服務的簡短資訊	已新增 Amazon 驗證許可的相關資訊，提供與 OPA 相同的功能和優點。	2023 年 5 月 22 日
—	初次出版	2021 年 8 月 17 日

AWS 規定指引詞彙

以下是 AWS 規範性指引所提供的策略、指南和模式中常用的術語。若要建議項目，請使用詞彙表末尾的提供意見回饋連結。

數字

7 R

將應用程式移至雲端的七種常見遷移策略。這些策略以 Gartner 在 2011 年確定的 5 R 為基礎，包括以下內容：

- 重構/重新架構 – 充分利用雲端原生功能來移動應用程式並修改其架構，以提高敏捷性、效能和可擴展性。這通常涉及移植作業系統和資料庫。範例：將您的現場部署 Oracle 資料庫遷移到與 Amazon Aurora PostgreSQL 相容的版本。
- 平台轉換 (隨即重塑) – 將應用程式移至雲端，並引入一定程度的優化以利用雲端功能。範例：將您的現場部署 Oracle 資料庫遷移到 Amazon Relational Database Service 服務 (Amazon RDS)，適用於 AWS 雲端.
- 重新購買 (捨棄再購買) – 切換至不同的產品，通常從傳統授權移至 SaaS 模型。範例：將您的客戶關係管理 (CRM) 系統遷移至 SalesForce.com。
- 主機轉換 (隨即轉移) – 將應用程式移至雲端，而不進行任何變更以利用雲端功能。範例：將您的現場部署 Oracle 資料庫遷移至中 EC2 執行個體上的 Oracle 資料庫 AWS 雲端。
- 重新放置 (虛擬機器監視器等級隨即轉移) – 將基礎設施移至雲端，無需購買新硬體、重寫應用程式或修改現有操作。您可以將伺服器從內部部署平台遷移到相同平台的雲端服務。範例：將 Microsoft Hyper-V應用程式移轉至 AWS.
- 保留 (重新檢視) – 將應用程式保留在來源環境中。其中可能包括需要重要重構的應用程式，且您希望將該工作延遲到以後，以及您想要保留的舊版應用程式，因為沒有業務理由來進行遷移。
- 淘汰 – 解除委任或移除來源環境中不再需要的應用程式。

A

ABAC

請參閱以屬性為基礎的存取控制。

抽象的服務

請參閱受管理服務。

酸

請參閱原子性、一致性、隔離性、耐用性。

主動-主動式遷移

一種資料庫遷移方法，其中來源和目標資料庫保持同步 (透過使用雙向複寫工具或雙重寫入操作)，且兩個資料庫都在遷移期間處理來自連接應用程式的交易。此方法支援小型、受控制批次的遷移，而不需要一次性切換。它比主動-被動遷移更具彈性，但需要更多的工作。

主動-被動式遷移

一種資料庫遷移方法，其中來源和目標資料庫保持同步，但只有來源資料庫處理來自連接應用程式的交易，同時將資料複寫至目標資料庫。目標資料庫在遷移期間不接受任何交易。

聚合函數

在一組資料列上運作，並計算群組的單一傳回值的 SQL 函數。彙總函式的範例包括SUM和MAX。

AI

請參閱人工智慧。

艾奧運

請參閱人工智慧作業。

匿名化

永久刪除資料集中個人資訊的程序。匿名化可以幫助保護個人隱私。匿名資料不再被視為個人資料。

反模式

一種經常使用的解決方案，用於解決方案的生產力適得其反，效果不佳或效果低於替代方案。

應用控制

一種安全性方法，只允許使用核准的應用程式，以協助保護系統免受惡意軟體的攻擊。

應用程式組合

有關組織使用的每個應用程式的詳細資訊的集合，包括建置和維護應用程式的成本及其商業價值。此資訊是產品組合探索和分析程序的關鍵，有助於識別要遷移、現代化和優化的應用程式並排定其優先順序。

人工智慧 (AI)

電腦科學領域，致力於使用運算技術來執行通常與人類相關的認知功能，例如學習、解決問題和識別模式。如需詳細資訊，請參閱什麼是人工智慧？

人工智慧操作 (AIOps)

使用機器學習技術解決操作問題、減少操作事件和人工干預以及提高服務品質的程序。如需有關如何在 AWS 遷移策略中使用 AIOps 的詳細資訊，請參閱操作整合指南。

非對稱加密

一種加密演算法，它使用一對金鑰：一個用於加密的公有金鑰和一個用於解密的私有金鑰。您可以共用公有金鑰，因為它不用於解密，但對私有金鑰存取應受到高度限制。

原子性、一致性、隔離性、耐久性 (ACID)

一組軟體屬性，即使在出現錯誤、電源故障或其他問題的情況下，也能確保資料庫的資料有效性和操作可靠性。

屬性型存取控制 (ABAC)

根據使用者屬性 (例如部門、工作職責和團隊名稱) 建立精細許可的實務。如需詳細資訊，請參閱 AWS Identity and Access Management (IAM) 文件 AWS中的 ABAC。

授權資料來源

儲存資料主要版本的位置，被認為是最可靠的資訊來源。您可以將授權資料來源中的資料複製到其他位置，以便處理或修改資料，例如匿名化、編輯或將其虛擬化。

可用區域

一個獨立的位置，與其他 AWS 區域 可用區域中的故障隔離，並為相同區域中的其他可用區域提供廉價、低延遲的網路連線能力。

AWS 雲端採用架構 (AWS CAF)

指導方針和最佳做法的架構，可協 AWS 助組織制定有效率且有效的計畫，以順利移轉至雲端。AWS CAF 將指導組織到六個重點領域，稱為觀點：業務，人員，治理，平台，安全性和運營。業務、人員和控管層面著重於業務技能和程序；平台、安全和操作層面著重於技術技能和程序。例如，人員層面針對處理人力資源 (HR)、人員配備功能和人員管理的利害關係人。針對此觀點，AWS CAF 為人員開發、訓練和通訊提供指導，以協助組織為成功採用雲端做好準備。如需詳細資訊，請參閱 AWS CAF 網站和 AWS CAF 白皮書。

AWS 工作負載資格架構 (AWS WQF)

可評估資料庫移轉工作負載、建議移轉策略並提供工作預估的工具。 AWS WQF 包含在 AWS Schema Conversion Tool ()AWS SCT中。它會分析資料庫結構描述和程式碼物件、應用程式程式碼、相依性和效能特性，並提供評估報告。

B

壞機器人

旨在破壞或對個人或組織造成傷害的機器人。

BCP

請參閱業務連續性規劃。

行為圖

資源行為的統一互動式檢視，以及一段時間後的互動。您可以將行為圖與 Amazon Detective 搭配使用來檢查失敗的登入嘗試、可疑的 API 呼叫和類似動作。如需詳細資訊，請參閱偵測文件中的行為圖中的資料。

大端序系統

首先儲存最高有效位元組的系統。另請參閱 「位元順序」。

二進制分類

預測二進制結果的過程 (兩個可能的類別之一)。例如，ML 模型可能需要預測諸如「此電子郵件是否是垃圾郵件？」等問題 或「產品是書還是汽車？」

Bloom 篩選條件

一種機率性、記憶體高效的資料結構，用於測試元素是否為集的成員。

藍/綠部署

建立兩個獨立但相同環境的部署策略。您可以在一個環境中執行目前的應用程式版本 (藍色)，而在另一個環境 (綠色) 中執行新的應用程式版本。此策略可協助您以最小的影響快速回復。

機器人

透過網際網路執行自動化工作並模擬人類活動或互動的軟體應用程式。某些漫遊器是有用的或有益的，例如用於索引 Internet 上信息的網絡爬蟲。其他一些機器人（稱為不良機器人）旨在破壞或對個人或組織造成傷害。

殭屍網絡

受惡意軟件感染並受到單一方（稱為機器人牧民或機器人操作員）控制的機器人網絡。殭屍網絡是擴展機器人及其影響的最著名機制。

分支

程式碼儲存庫包含的區域。儲存庫中建立的第一個分支是主要分支。您可以從現有分支建立新分支，然後在新分支中開發功能或修正錯誤。您建立用來建立功能的分支通常稱為功能分支。當準備好發佈功能時，可以將功能分支合併回主要分支。如需詳細資訊，請參閱關於分支 (GitHub 文件)。

防碎玻璃訪問

在特殊情況下，並透過核准的程序，使用者可以快速取得他 AWS 帳戶 們通常沒有存取權限的存取權。如需詳細資訊，請參閱 AWS Well-Architected 指南中的實作防破玻璃程序指標。

棕地策略

環境中的現有基礎設施。對系統架構採用棕地策略時，可以根據目前系統和基礎設施的限制來設計架構。如果正在擴展現有基礎設施，則可能會混合棕地和綠地策略。

緩衝快取

儲存最常存取資料的記憶體區域。

業務能力

業務如何創造價值 (例如，銷售、客戶服務或營銷)。業務能力可驅動微服務架構和開發決策。如需詳細資訊，請參閱在 AWS上執行容器化微服務白皮書的圍繞業務能力進行組織部分。

業務連續性規劃 (BCP)

一種解決破壞性事件 (如大規模遷移) 對營運的潛在影響並使業務能夠快速恢復營運的計畫。

C

咖啡

請參閱AWS 雲端採用架構。

金絲雀部署

向最終用戶發行版本的緩慢和增量版本。當您有信心時，您可以部署新版本並完全取代目前的版本。

CCoE

請參閱雲端卓越中心。

CDC

請參閱變更資料擷取。

變更資料擷取 (CDC)

追蹤對資料來源 (例如資料庫表格) 的變更並記錄有關變更的中繼資料的程序。您可以將 CDC 用於各種用途，例如稽核或複寫目標系統中的變更以保持同步。

混沌工程

故意引入故障或破壞性事件來測試系統的彈性。您可以使用 AWS Fault Injection Service (AWS FIS) 執行實驗來 stress 您的 AWS 工作負載並評估其回應。

CI/CD

請參閱持續整合和持續交付。

分類

有助於產生預測的分類程序。用於分類問題的 ML 模型可預測離散值。離散值永遠彼此不同。例如，模型可能需要評估影像中是否有汽車。

用戶端加密

在目標 AWS 服務 接收資料之前，在本機加密資料。

雲端卓越中心 (CCoE)

一個多學科團隊，可推動整個組織的雲端採用工作，包括開發雲端最佳實務、調動資源、制定遷移時間表以及領導組織進行大規模轉型。如需詳細資訊，請參閱 AWS 雲端 企業策略部落格上的 CCoE 文章。

雲端運算

通常用於遠端資料儲存和 IoT 裝置管理的雲端技術。雲計算通常連接到邊緣計算技術。

雲端運作模式

在 IT 組織中，這是用來建置、成熟和最佳化一或多個雲端環境的作業模型。如需詳細資訊，請參閱建立您的雲端作業模型。

採用雲端階段

組織移轉至下列四個階段時通常會經歷 AWS 雲端：

- 專案 – 執行一些與雲端相關的專案以進行概念驗證和學習用途
- 基礎 – 進行基礎投資以擴展雲端採用 (例如，建立登陸區域、定義 CCoE、建立營運模型)
- 遷移 – 遷移個別應用程式
- 重塑 – 優化產品和服務，並在雲端中創新

這些階段是 Stephen Orban 在 AWS 雲端 企業策略部落格部落格文章「邁向雲端優先的旅程與採用階段」中所定義的。如需其與 AWS 移轉策略之間關聯的詳細資訊，請參閱移轉準備指南。

CMDB

請參閱組態管理資料庫。

程式碼儲存庫

透過版本控制程序來儲存及更新原始程式碼和其他資產 (例如文件、範例和指令碼) 的位置。常見的雲存儲庫包括 GitHub 或 AWS CodeCommit。程式碼的每個版本都稱為分支。在微服務結構中，每個儲存庫都專用於單個功能。單一 CI/CD 管道可以使用多個儲存庫。

冷快取

一種緩衝快取，它是空的、未填充的，或者包含過時或不相關的資料。這會影響效能，因為資料庫執行個體必須從主記憶體或磁碟讀取，這比從緩衝快取讀取更慢。

冷資料

很少存取且通常是歷史資料。查詢此類資料時，通常可以接受緩慢的查詢。將此資料移至效能較低且成本較低的儲存層或類別可降低成本。

計算機視覺（CV）

一個 AI 領域，它使用機器學習來分析和從數字圖像和視頻等視覺格式中提取信息。例如，提 AWS Panorama 供將 CV 添加到現場部署攝像機網絡的設備，Amazon 為 CV SageMaker 提供圖像處理算法。

配置漂移

對於工作負載，組態會從預期的狀態變更。這可能會導致工作負載變得不合規，而且通常是漸進且無意的。

組態管理資料庫 (CMDB)

儲存和管理有關資料庫及其 IT 環境的資訊的儲存庫，同時包括硬體和軟體元件及其組態。您通常在遷移的產品組合探索和分析階段使用 CMDB 中的資料。

一致性套件

AWS Config 規則和補救動作的集合，您可以組合這些動作來自訂合規性和安全性檢查。您可以使用 YAML 範本，將一致性套件部署為 AWS 帳戶 和區域中的單一實體，或跨組織部署。如需詳細資訊，請參閱文件中的AWS Config 一致性套件。

持續整合和持續交付 (CI/CD)

自動化軟體發行程序的來源、建置、測試、暫存和生產階段的程序。CI/CD 通常被描述為管道。CI/CD 可協助您將程序自動化、提升生產力、改善程式碼品質以及加快交付速度。如需詳細資訊，請參閱持續交付的優點。CD 也可表示持續部署。如需詳細資訊，請參閱持續交付與持續部署。

CV

請參閱電腦視覺。

D

靜態資料

網路中靜止的資料，例如儲存中的資料。

資料分類

根據重要性和敏感性來識別和分類網路資料的程序。它是所有網路安全風險管理策略的關鍵組成部分，因為它可以協助您確定適當的資料保護和保留控制。資料分類是 AWS Well-Architected 架構中安全性支柱的一個組成部分。如需詳細資訊，請參閱資料分類。

資料漂移

生產資料與用來訓練 ML 模型的資料之間有意義的變化，或輸入資料隨著時間的推移有意義的變化。資料漂移可降低機器學習模型預測中的整體品質、準確性和公平性。

傳輸中的資料

在您的網路中主動移動的資料，例如在網路資源之間移動。

資料網格

透過集中式管理和控管，提供分散式、分散式資料擁有權的架構架構。

資料最小化

僅收集和處理絕對必要的數據的原則。在中執行資料最小化 AWS 雲端 可降低隱私權風險、成本和分析碳足跡。

資料周長

您 AWS 環境中的一組預防性護欄，可協助確保只有受信任的身分正在存取來自預期網路的受信任資源。若要取得更多資訊，請參閱〈在上建立資料周長〉 AWS。

資料預先處理

將原始資料轉換成 ML 模型可輕鬆剖析的格式。預處理資料可能意味著移除某些欄或列，並解決遺失、不一致或重複的值。

數據來源

在整個生命週期中追蹤資料來源和歷史記錄的程序，例如資料的產生、傳輸和儲存方式。

資料主體

正在收集和處理資料的個人。

資料倉儲

支援商業智慧 (例如分析) 的資料管理系統。資料倉儲通常包含大量歷史資料，通常用於查詢和分析。

資料庫定義語言 (DDL)

用於建立或修改資料庫中資料表和物件之結構的陳述式或命令。

資料庫處理語言 (DML)

用於修改 (插入、更新和刪除) 資料庫中資訊的陳述式或命令。

DDL

請參閱資料庫定義語言。

深度整體

結合多個深度學習模型進行預測。可以使用深度整體來獲得更準確的預測或估計預測中的不確定性。

深度學習

一個機器學習子領域，它使用多層人工神經網路來識別感興趣的輸入資料與目標變數之間的對應關係。

defense-in-depth

這是一種資訊安全方法，其中一系列的安全機制和控制項會在整個電腦網路中精心分層，以保護網路和其中資料的機密性、完整性和可用性。在上採用此策略時 AWS，您可以在 AWS

Organizations 結構的不同層加入多個控制項，以協助保護資源。例如，一 defense-in-depth 種方法可能會結合多因素驗證、網路分段和加密。

委派的管理員

在中 AWS Organizations，相容的服務可以註冊成 AWS 員帳戶，以管理組織的帳戶並管理該服務的權限。此帳戶稱為該服務的委派管理員。如需詳細資訊和相容服務清單，請參閱 AWS Organizations 文件中的可搭配 AWS Organizations運作的服務。

部署

在目標環境中提供應用程式、新功能或程式碼修正的程序。部署涉及在程式碼庫中實作變更，然後在應用程式環境中建置和執行該程式碼庫。

開發環境

請參閱環境。

偵測性控制

一種安全控制，用於在事件發生後偵測、記錄和提醒。這些控制是第二道防線，提醒您注意繞過現有預防性控制的安全事件。如需詳細資訊，請參閱在 AWS上實作安全控制中的偵測性控制。

發展價值流映射

用於識別限制並排定優先順序，對軟體開發生命週期中的速度和品質產生不利影響的程序。DVSM 擴展了最初為精益生產實踐而設計的價值流映射流程。它著重於創造和通過軟件開發過程中移動價值所需的步驟和團隊。

數字雙胞胎

真實世界系統的虛擬表現法，例如建築物、工廠、工業設備或生產線。數位雙胞胎支援預測性維護、遠端監控和生產最佳化。

維度表

在 star 結構描述中，較小的資料表包含事實資料表中定量資料的相關資料屬性。維度表格屬性通常是文字欄位或離散數字，其行為類似於文字。這些屬性通常用於查詢限制、篩選和結果集標籤。

災難

防止工作負載或系統在其主要部署位置達成其業務目標的事件。這些事件可能是自然災害、技術故障或人為行為造成的結果，例如意外設定錯誤或惡意軟體攻擊。

災難復原 (DR)

您使用的策略和程序，將因災難造成的停機時間和資料遺失降到最低。如需詳細資訊，請參閱 AWS Well-Architected 的架構中的雲端中的工作負載的災難復原 AWS：雲端復原。

DML

請參閱資料庫操作語言。

領域驅動的設計

一種開發複雜軟體系統的方法，它會將其元件與每個元件所服務的不斷發展的領域或核心業務目標相關聯。Eric Evans 在其著作 Domain-Driven Design: Tackling Complexity in the Heart of Software (Boston: Addison-Wesley Professional, 2003) 中介紹了這一概念。如需有關如何將領域驅動的設計與 strangler fig 模式搭配使用的資訊，請參閱使用容器和 Amazon API Gateway 逐步現代化舊版 Microsoft ASP.NET (ASMX) Web 服務。

博士

請參閱災難復原。

漂移檢測

追蹤基線組態的偏差。例如，您可以用 AWS CloudFormation 來偵測系統資源中的漂移，也可以用 AWS Control Tower 來偵測 landing zone 中可能會影響法規遵循治理要求的變更。

DVSM

請參閱開發價值流映射。

E

EDA

請參閱探索性資料分析。

邊緣運算

提升 IoT 網路邊緣智慧型裝置運算能力的技術。與雲計算相比，邊緣計算可以減少通信延遲並縮短響應時間。

加密

一種計算過程，將純文本數據（這是人類可讀的）轉換為密文。

加密金鑰

由加密演算法產生的隨機位元的加密字串。金鑰長度可能有所不同，每個金鑰的設計都是不可預測且唯一的。

端序

位元組在電腦記憶體中的儲存順序。大端序系統首先儲存最高有效位元組。小端序系統首先儲存最低有效位元組。

端點

請參閱服務端點。

端點服務

您可以在虛擬私有雲端 (VPC) 中託管以與其他使用者共用的服務。您可以使用其他或 (IAM) 主體建立端點服務， AWS PrivateLink 並將權限授予其他 AWS 帳戶 或 AWS Identity and Access Management (IAM) 主體。這些帳戶或主體可以透過建立介面 VPC 端點私下連接至您的端點服務。如需詳細資訊，請參閱 Amazon Virtual Private Cloud (Amazon VPC) 文件中的建立端點服務。

企業資源規劃

可自動化並管理企業關鍵業務流程 (例如會計、MES 和專案管理) 的系統。

信封加密

使用另一個加密金鑰對某個加密金鑰進行加密的程序。如需詳細資訊，請參閱 AWS Key Management Service (AWS KMS) 文件中的信封加密。

環境

執行中應用程式的執行個體。以下是雲端運算中常見的環境類型：

- 開發環境 – 執行中應用程式的執行個體，只有負責維護應用程式的核心團隊才能使用。開發環境用來測試變更，然後再將開發環境提升到較高的環境。此類型的環境有時稱為測試環境。
- 較低的環境 – 應用程式的所有開發環境，例如用於初始建置和測試的開發環境。
- 生產環境 – 最終使用者可以存取的執行中應用程式的執行個體。在 CI/CD 管道中，生產環境是最後一個部署環境。
- 較高的環境 – 核心開發團隊以外的使用者可存取的所有環境。這可能包括生產環境、生產前環境以及用於使用者接受度測試的環境。

epic

在敏捷方法中，有助於組織工作並排定工作優先順序的功能類別。epic 提供要求和實作任務的高層級描述。例如， AWS CAF 安全史詩包括身份和訪問管理，偵探控制，基礎結構安全性，數據保護和事件響應。如需有關 AWS 遷移策略中的 Epic 的詳細資訊，請參閱計畫實作指南。

ERP

請參閱企業資源規劃。

探索性資料分析 (EDA)

分析資料集以了解其主要特性的過程。您收集或彙總資料，然後執行初步調查以尋找模式、偵測異常並檢查假設。透過計算摘要統計並建立資料可視化來執行 EDA。

F

事實表

星型架構中的中央表格。它存儲有關業務運營的定量數據。事實資料表通常包含兩種類型的資料欄：包含計量的資料欄，以及包含維度表格外部索引鍵的資料欄。

快速失敗

一種使用頻繁和增量測試來減少開發生命週期的理念。這是敏捷方法的關鍵部分。

故障隔離邊界

在中 AWS 雲端，可用區域、 AWS 區域控制平面或資料平面等界限，可限制故障的影響，並協助改善工作負載的彈性。如需詳細資訊，請參閱AWS 錯誤隔離邊界。

功能分支

請參閱分支。

特徵

用來進行預測的輸入資料。例如，在製造環境中，特徵可能是定期從製造生產線擷取的影像。

功能重要性

特徵對於模型的預測有多重要。這通常表示為可以透過各種技術來計算的數值得分，例如 Shapley Additive Explanations (SHAP) 和積分梯度。如需詳細資訊，請參閱機器學習模型可解釋性：AWS。

特徵轉換

優化 ML 程序的資料，包括使用其他來源豐富資料、調整值、或從單一資料欄位擷取多組資訊。這可讓 ML 模型從資料中受益。例如，如果將「2021-05-27 00:15:37」日期劃分為「2021」、「五月」、「週四」和「15」，則可以協助學習演算法學習與不同資料元件相關聯的細微模式。

FGAC

請參閱精細的存取控制。

精細的存取控制 (FGAC)

使用多個條件來允許或拒絕訪問請求。

閃切遷移

一種資料庫移轉方法，透過變更資料擷取使用連續資料複寫，在最短的時間內移轉資料，而不是使用階段化方法。目標是將停機時間降至最低。

G

地理阻塞

請參閱地理限制。

地理限制 (地理封鎖)

在 Amazon 中 CloudFront，防止特定國家/地區的使用者存取內容分發的選項。您可以使用允許清單或封鎖清單來指定核准和禁止的國家/地區。如需詳細資訊，請參閱 CloudFront 文件中的限制內容的地理分佈。

Gitflow 工作流程

這是一種方法，其中較低和較高環境在原始碼儲存庫中使用不同分支。Gitflow 工作流程被認為是遺留的，基於主幹的工作流程是現代的首選方法。

綠地策略

新環境中缺乏現有基礎設施。對系統架構採用綠地策略時，可以選擇所有新技術，而不會限制與現有基礎設施的相容性，也稱為棕地。如果正在擴展現有基礎設施，則可能會混合棕地和綠地策略。

防護機制

有助於跨組織單位 (OU) 來管控資源、政策和合規的高層級規則。預防性防護機制會強制執行政策，以確保符合合規標準。透過使用服務控制政策和 IAM 許可界限來將其實作。偵測性防護機制可偵測政策違規和合規問題，並產生提醒以便修正。它們是通過使用 AWS Config，Amazon AWS Security Hub GuardDuty， AWS Trusted Advisor亞馬遜檢查 Amazon Inspector 和自定義 AWS Lambda 檢查來實現的。

H

公頃

查看高可用性。

異質資料庫遷移

將來源資料庫遷移至使用不同資料庫引擎的目標資料庫 (例如，Oracle 至 Amazon Aurora)。異質遷移通常是重新架構工作的一部分，而轉換結構描述可能是一項複雜任務。AWS 提供有助於結構描述轉換的 AWS SCT。

高可用性 (HA)

工作負載在遇到挑戰或災難時持續運作的能力，無需干預。HA 系統的設計可自動容錯移轉、持續提供高品質的效能，以及處理不同的負載和故障，並將效能影響降到最低。

歷史學家現代化

一種用於現代化和升級操作技術 (OT) 系統的方法，以更好地滿足製造業的需求。歷史學家是一種類型的數據庫，用於收集和存儲工廠中的各種來源的數據。

異質資料庫遷移

將您的來源資料庫遷移至共用相同資料庫引擎的目標資料庫 (例如，Microsoft SQL Server 至 Amazon RDS for SQL Server)。同質遷移通常是主機轉換或平台轉換工作的一部分。您可以使用原生資料庫公用程式來遷移結構描述。

熱數據

經常存取的資料，例如即時資料或最近的轉譯資料。此資料通常需要高效能的儲存層或類別，才能提供快速的查詢回應。

修補程序

緊急修正生產環境中的關鍵問題。由於其緊迫性，修補程式通常是在典型的 DevOps 發行工作流程之外進行。

超級護理期間

在切換後，遷移團隊在雲端管理和監控遷移的應用程式以解決任何問題的時段。通常，此期間的長度為 1-4 天。在超級護理期間結束時，遷移團隊通常會將應用程式的責任轉移給雲端營運團隊。

I

IaC

查看基礎結構即程式碼。

身分型政策

附加至一或多個 IAM 主體的政策，用於定義其在 AWS 雲端 環境中的許可。

閒置應用程式

90 天期間 CPU 和記憶體平均使用率在 5% 至 20% 之間的應用程式。在遷移專案中，通常會淘汰這些應用程式或將其保留在內部部署。

IIoT

請參閱工業物聯網。

不可變基礎設施

為生產工作負載部署新基礎結構的模型，而不是更新、修補或修改現有基礎結構。不可變的基礎架構本質上比可變基礎架構更加一致、可靠且可預測。如需詳細資訊，請參閱 Well-Architected 的架構中的使用不可變基礎結 AWS 構進行部署最佳作法。

傳入 (輸入) VPC

在 AWS 多帳戶架構中，VPC 可接受、檢查和路由來自應用程式外部的網路連線。AWS 安全參考架構建議您使用傳入、傳出和檢查 VPC 來設定網路帳戶，以保護應用程式與更廣泛的網際網路之間的雙向介面。

增量遷移

一種切換策略，您可以在其中將應用程式分成小部分遷移，而不是執行單一、完整的切換。例如，您最初可能只將一些微服務或使用者移至新系統。確認所有項目都正常運作之後，您可以逐步移動其他微服務或使用者，直到可以解除委任舊式系統。此策略可降低與大型遷移關聯的風險。

工業 4.0

Klaus Schwab 於 2016 年推出的一個術語，指的是透過連線能力、即時資料、自動化、分析和 AI/ML 的進步來實現製造流程的現代化。

基礎設施

應用程式環境中包含的所有資源和資產。

基礎設施即程式碼 (IaC)

透過一組組態檔案來佈建和管理應用程式基礎設施的程序。IaC 旨在協助您集中管理基礎設施，標準化資源並快速擴展，以便新環境可重複、可靠且一致。

工業物聯網 (IIoT)

在製造業、能源、汽車、醫療保健、生命科學和農業等產業領域使用網際網路連線的感測器和裝置。如需詳細資訊，請參閱建立工業物聯網 (IIoT) 數位轉型策略。

檢查 VPC

在 AWS 多帳戶架構中，集中式 VPC 可管理 VPC（相同或不同 AWS 區域）、網際網路和內部部署網路之間的網路流量檢查。AWS 安全參考架構建議您使用傳入、傳出和檢查 VPC 來設定網路帳戶，以保護應用程式與更廣泛的網際網路之間的雙向介面。

物聯網 (IoT)

具有內嵌式感測器或處理器的相連實體物體網路，其透過網際網路或本地通訊網路與其他裝置和系統進行通訊。如需詳細資訊，請參閱什麼是 IoT？

可解釋性

機器學習模型的一個特徵，描述了人類能夠理解模型的預測如何依賴於其輸入的程度。如需詳細資訊，請參閱 AWS 的機器學習模型可解釋性。

IoT

請參閱物聯網。

IT 資訊庫 (ITIL)

一組用於交付 IT 服務並使這些服務與業務需求保持一致的最佳實務。ITIL 為 ITSM 提供了基礎。

IT 服務管理 (ITSM)

與組織的設計、實作、管理和支援 IT 服務關聯的活動。如需有關將雲端操作與 ITSM 工具整合的資訊，請參閱操作整合指南。

ITIL

請參閱 IT 資訊庫。

ITSM

請參閱 IT 服務管理。

L

標籤式存取控制 (LBAC)

強制存取控制 (MAC) 的實作，其中每個使用者和資料本身都明確指派一個安全性標籤值。使用者安全性標籤與資料安全性標籤之間的交集決定了使用者可以看到哪些列與欄。

登陸區域

landing zone 是一個架構良好的多帳戶 AWS 環境，具有可擴展性和安全性。這是一個起點，您的組織可以從此起點快速啟動和部署工作負載與應用程式，並對其安全和基礎設施環境充滿信心。如需有關登陸區域的詳細資訊，請參閱設定安全且可擴展的多帳戶 AWS 環境。

大型遷移

遷移 300 部或更多伺服器。

LBAC

請參閱以標示為基礎的存取控制。

最低權限

授予執行任務所需之最低許可的安全最佳實務。如需詳細資訊，請參閱 IAM 文件中的套用最低權限許可。

隨即轉移

見 7 盧比

小端序系統

首先儲存最低有效位元組的系統。另請參閱 「位元順序」。

較低的環境

請參閱環境。

M

機器學習 (ML)

一種使用演算法和技術進行模式識別和學習的人工智慧。機器學習會進行分析並從記錄的資料 (例如物聯網 (IoT) 資料) 中學習，以根據模式產生統計模型。如需詳細資訊，請參閱機器學習。

主要分支

請參閱分支。

惡意軟體

旨在危及計算機安全性或隱私的軟件。惡意軟件可能會破壞計算機系統，洩漏敏感信息或獲得未經授權的訪問。惡意軟體的例子包括病毒、蠕蟲、勒索軟體、特洛伊木馬程式、間諜軟體和鍵盤記錄程式。

受管理服務

AWS 服務 用於 AWS 操作基礎架構層、作業系統和平台，並且您可以存取端點以儲存和擷取資料。Amazon Simple Storage Service (Amazon S3) 和 Amazon DynamoDB 是受管服務的範例。這些也稱為抽象服務。

製造執行系統

用於跟踪，監控，記錄和控制生產過程的軟件系統，可在現場將原材料轉換為成品。

MAP

請參閱 Migration Acceleration Program。

機制

一個完整的過程，您可以在其中創建工具，推動工具的採用，然後檢查結果以進行調整。機制是一個循環，它加強和改善自己，因為它運行。如需詳細資訊，請參閱 AWS Well-Architected 的架構中建置機制。

成員帳戶

屬於 AWS 帳戶 中組織的管理帳戶以外的所有帳戶 AWS Organizations。一個帳戶一次只能是一個組織的成員。

MES

請參閱製造執行系統。

郵件佇列遙測傳輸 (MQTT)

以發佈/訂閱模式為基礎的輕量型 machine-to-machine (M2M) 通訊協定，適用於資源受限 IoT 裝置。

微服務

一種小型的獨立服務，它可透過定義明確的 API 進行通訊，通常由小型獨立團隊擁有。例如，保險系統可能包含對應至業務能力 (例如銷售或行銷) 或子領域 (例如購買、索賠或分析) 的微服務。微服

務的優點包括靈活性、彈性擴展、輕鬆部署、可重複使用的程式碼和適應力。如需詳細資訊，請參閱使用 AWS 無伺服器服務整合微服務。

微服務架構

一種使用獨立元件來建置應用程式的方法，這些元件會以微服務形式執行每個應用程式程序。這些微服務會使用輕量型 API，透過明確定義的介面進行通訊。此架構中的每個微服務都可以進行更新、部署和擴展，以滿足應用程式特定功能的需求。如需詳細資訊，請參閱上 AWS的實作微服務。

Migration Acceleration Program (MAP)

提供諮詢支援、訓練和服務的 AWS 計畫，協助組織為移轉至雲端建立穩固的營運基礎，並協助抵消移轉的初始成本。MAP 包括用於有條不紊地執行舊式遷移的遷移方法以及一組用於自動化和加速常見遷移案例的工具。

大規模遷移

將大部分應用程式組合依波次移至雲端的程序，在每個波次中，都會以更快的速度移動更多應用程式。此階段使用從早期階段學到的最佳實務和經驗教訓來實作團隊、工具和流程的遷移工廠，以透過自動化和敏捷交付簡化工作負載的遷移。這是 AWS 遷移策略的第三階段。

遷移工廠

可透過自動化、敏捷的方法簡化工作負載遷移的跨職能團隊。移轉工廠團隊通常包括營運、業務分析師和擁有者、移轉工程師、開發人員和 DevOps 專業人員。20% 至 50% 之間的企業應用程式組合包含可透過工廠方法優化的重複模式。如需詳細資訊，請參閱此內容集中的遷移工廠的討論和雲端遷移工廠指南。

遷移中繼資料

有關完成遷移所需的應用程式和伺服器的資訊。每種遷移模式都需要一組不同的遷移中繼資料。移轉中繼資料的範例包括目標子網路、安全性群組和 AWS 帳戶。

遷移模式

可重複的遷移任務，詳細描述遷移策略、遷移目的地以及所使用的遷移應用程式或服務。範例：使 AWS 用應用程式遷移服務將遷移重新託管到 Amazon EC2。

遷移組合評定 (MPA)

這是一種線上工具，可提供驗證要移轉至的商業案例的 AWS 雲端資訊。MPA 提供詳細的組合評定 (伺服器適當規模、定價、總體擁有成本比較、遷移成本分析) 以及遷移規劃 (應用程式資料分析和資料收集、應用程式分組、遷移優先順序，以及波次規劃)。所有 AWS 顧問和 APN 合作夥伴顧問均可免費使用 MPA 工具 (需要登入)。

遷移準備程度評定 (MRA)

使用 AWS CAF 獲得有關組織雲端準備狀態的見解、識別優勢和弱點，以及建立行動計劃以縮小已識別差距的過程。如需詳細資訊，請參閱遷移準備程度指南。MRA 是 AWS 遷移策略的第一階段。

遷移策略

將工作負載移轉至 AWS 雲端. 如需詳細資訊，請參閱本詞彙表中的 7 Rs 項目，並參閱動員您的組織以加速大規模移轉。

機器學習 (ML)

請參閱機器學習。

現代化

將過時的 (舊版或單一) 應用程式及其基礎架構轉換為雲端中靈活、富有彈性且高度可用的系統，以降低成本、提高效率並充分利用創新。如需詳細資訊，請參閱. AWS 雲端

現代化準備程度評定

這項評估可協助判斷組織應用程式的現代化準備程度；識別優點、風險和相依性；並確定組織能夠在多大程度上支援這些應用程式的未來狀態。評定的結果就是目標架構的藍圖、詳細說明現代化程序的開發階段和里程碑的路線圖、以及解決已發現的差距之行動計畫。如需詳細資訊，請參閱評估應用程式的現代化準備程度 AWS 雲端。

單一應用程式 (單一)

透過緊密結合的程序作為單一服務執行的應用程式。單一應用程式有幾個缺點。如果一個應用程式功能遇到需求激增，則必須擴展整個架構。當程式碼庫增長時，新增或改進單一應用程式的功能也會變得更加複雜。若要解決這些問題，可以使用微服務架構。如需詳細資訊，請參閱將單一體系分解為微服務。

MPA

請參閱移轉組合評估。

MQTT

請參閱佇列遙測傳輸的郵件。

多類別分類

一個有助於產生多類別預測的過程 (預測兩個以上的結果之一)。例如，機器學習模型可能會詢問「此產品是書籍、汽車還是電話？」 或者「這個客戶對哪種產品類別最感興趣？」

可變的基礎

一種模型，用於更新和修改生產工作負載的現有基礎結構。為了提高一致性，可靠性和可預測性，AWS Well-Architected 框架建議使用不可變的基礎結構作為最佳實踐。

O

OAC

請參閱原始存取控制。

OAI

請參閱原始存取身分。

OCM

請參閱組織變更管理。

離線遷移

一種遷移方法，可在遷移過程中刪除來源工作負載。此方法涉及延長停機時間，通常用於小型非關鍵工作負載。

OI

請參閱作業整合。

OLA

請參閱作業層級協定。

線上遷移

一種遷移方法，無需離線即可將來源工作負載複製到目標系統。連接至工作負載的應用程式可在遷移期間繼續運作。此方法涉及零至最短停機時間，通常用於關鍵的生產工作負載。

OPCA

請參閱開放程序通訊-統一架構。

開放程序通訊-統一架構 (OPC-UA)

用於工業自動化的 machine-to-machine (M2M) 通訊協定。OPC-UA 提供數據加密，身份驗證和授權方案的互操作性標準。

操作水準協議 (OLA)

一份協議，闡明 IT 職能群組承諾向彼此提供的內容，以支援服務水準協議 (SLA)。

操作準備程度檢討 (ORR)

問題和相關最佳做法的檢查清單，可協助您瞭解、評估、預防或減少事件和可能的故障範圍。如需詳細資訊，請參閱 AWS Well-Architected 的架構中的作業準備檢閱 (ORR)。

操作技術

可與實體環境搭配使用的硬體和軟體系統，以控制工業作業、設備和基礎設施。在製造業中，整合 OT 和資訊技術 (IT) 系統是工業 4.0 轉型的關鍵焦點。

操作整合 (OI)

在雲端中將操作現代化的程序，其中包括準備程度規劃、自動化和整合。如需詳細資訊，請參閱操作整合指南。

組織追蹤

由建立的追蹤 AWS CloudTrail 記錄中組織 AWS 帳戶 中所有人的所有事件 AWS Organizations。在屬於組織的每個 AWS 帳戶 中建立此追蹤，它會跟蹤每個帳戶中的活動。如需詳細資訊，請參閱CloudTrail文件中的為組織建立追蹤。

組織變更管理 (OCM)

用於從人員、文化和領導力層面管理重大、顛覆性業務轉型的架構。OCM 透過加速變更採用、解決過渡問題，以及推動文化和組織變更，協助組織為新系統和策略做好準備，並轉移至新系統和策略。在 AWS 移轉策略中，這個架構稱為人員加速，因為雲端採用專案所需的變更速度。如需詳細資訊，請參閱 OCM 指南。

原始存取控制 (OAC)

在中 CloudFront，限制存取權限以保護 Amazon Simple Storage Service (Amazon S3) 內容的增強選項。OAC 支援所有 S3 儲存貯體 AWS 區域、伺服器端加密 AWS KMS (SSE-KMS)，以及 S3 儲存貯體的動態PUT和DELETE請求。

原始存取身分 (OAI)

在中 CloudFront，用於限制存取以保護 Amazon S3 內容的選項。當您使用 OAI 時， CloudFront 會建立 Amazon S3 可用來進行驗證的主體。經驗證的主體只能透過特定散發存取 S3 儲存 CloudFront 貯體中的內容。另請參閱 OAC，它可提供更精細且增強的存取控制。

ORR

請參閱作業整備檢閱。

OT

請參閱操作技術。

傳出 (輸出) VPC

在 AWS 多帳戶架構中，處理從應用程式內啟動的網路連線的 VPC。AWS 安全參考架構建議您使用傳入、傳出和檢查 VPC 來設定網路帳戶，以保護應用程式與更廣泛的網際網路之間的雙向介面。

P

許可界限

附接至 IAM 主體的 IAM 管理政策，可設定使用者或角色擁有的最大許可。如需詳細資訊，請參閱 IAM 文件中的許可界限。

個人識別資訊 (PII)

直接查看或與其他相關數據配對時，可用於合理推斷個人身份的信息。PII 的範例包括姓名、地址和聯絡資訊。

PII

請參閱個人識別資訊。

手冊

一組預先定義的步驟，可擷取與遷移關聯的工作，例如在雲端中提供核心操作功能。手冊可以採用指令碼、自動化執行手冊或操作現代化環境所需的程序或步驟摘要的形式。

公司

請參閱可編程邏輯控制器

PLM

查看產品生命週期管理。

政策

可以定義權限 (請參閱以身分識別為基礎的策略)、指定存取條件 (請參閱以資源為基礎的策略) 或定義組織中所有帳戶的最大權限的物件 AWS Organizations (請參閱服務控制策略)。

混合持久性

根據資料存取模式和其他需求，獨立選擇微服務的資料儲存技術。如果您的微服務具有相同的資料儲存技術，則其可能會遇到實作挑戰或效能不佳。如果微服務使用最適合其需求的資料儲存，則可以更輕鬆地實作並達到更好的效能和可擴展性。如需詳細資訊，請參閱在微服務中啟用資料持久性。

組合評定

探索、分析應用程式組合並排定其優先順序以規劃遷移的程序。如需詳細資訊，請參閱評估遷移準備程度。

述詞

傳回`true`或的查詢條件`false`，通常位於子`WHERE`句中。

謂詞下推

一種資料庫查詢最佳化技術，可在傳輸前篩選查詢中的資料。這樣可減少必須從關聯式資料庫擷取和處理的資料量，並改善查詢效能。

預防性控制

旨在防止事件發生的安全控制。這些控制是第一道防線，可協助防止對網路的未經授權存取或不必要變更。如需詳細資訊，請參閱在 AWS上實作安全控制中的預防性控制。

委託人

中 AWS 可執行動作和存取資源的實體。此實體通常是 IAM 角色或使用者的根使用者。 AWS 帳戶如需詳細資訊，請參閱 IAM 文件中角色術語和概念中的主體。

隱私設計

一種系統工程方法，在整個工程過程中將隱私權納入考量。

私有託管區域

一種容器，它包含有關您希望 Amazon Route 53 如何回應一個或多個 VPC 內的域及其子域之 DNS 查詢的資訊。如需詳細資訊，請參閱 Route 53 文件中的使用私有託管區域。

主動控制

一種安全控制項，旨在防止部署不符合規範的資源。這些控制項會在資源佈建之前進行掃描。如果資源不符合控制項，則不會佈建該資源。如需詳細資訊，請參閱 AWS Control Tower 文件中的控制項參考指南，並參閱實作安全性控制中的主動控制 AWS。

產品生命週期管理 (PLM)

在產品的整個生命週期中管理資料和流程，從設計、開發、上市到成長與成熟度，再到下降和移除。

生產環境

請參閱環境。

可編程邏輯控制器 (PLC)

在製造業中，一台高度可靠且適應性強的計算機，可監控機器並自動化製造過程。

化名化

以預留位置值取代資料集中的個人識別碼的程序。化名化有助於保護個人隱私。假名化數據仍被認為是個人數據。

發布/訂閱 (發布/訂閱)

一種模式，可在微服務之間實現非同步通訊，以提高延展性和回應能力 例如，在微服務型 MES 中，微服務可以將事件訊息發佈到其他微服務可訂閱的通道。系統可以在不變更發佈服務的情況下新增微服務。

Q

查詢計劃

一系列步驟，如指示，用來存取 SQL 關聯式資料庫系統中的資料。

查詢計劃迴歸

在資料庫服務優化工具選擇的計畫比對資料庫環境進行指定的變更之前的計畫不太理想時。這可能因為對統計資料、限制條件、環境設定、查詢參數繫結的變更以及資料庫引擎的更新所導致。

R

拉齐矩阵

請參閱負責任，負責，諮詢，通知（RACI）。

勒索軟體

一種惡意軟體，旨在阻止對計算機系統或資料的存取，直到付款為止。

拉西矩陣

請參閱負責任，負責，諮詢，通知（RACI）。

RCAC

請參閱列與欄存取控制。

僅供讀取複本

用於唯讀用途的資料庫複本。您可以將查詢路由至僅供讀取複本以減少主資料庫的負載。

重新建築師

見 7 盧比

復原點目標 (RPO)

自上次資料復原點以來可接受的時間上限。這決定了最後一個恢復點和服務中斷之間可接受的數據丟失。

復原時間目標 (RTO)

服務中斷與恢復服務之間的最大可接受延遲。

重構

見 7 盧比

區域

地理區域中的 AWS 資源集合。每個都 AWS 區域 是隔離和獨立於其他的，以提供容錯能力，穩定性和彈性。如需詳細資訊，請參閱指定 AWS 區域 您的帳戶可以使用的項目。

迴歸

預測數值的 ML 技術。例如，為了解決「這房子會賣什麼價格？」的問題 ML 模型可以使用線性迴歸模型，根據已知的房屋事實 (例如，平方英尺) 來預測房屋的銷售價格。

重新主持

見 7 盧比

版本

在部署程序中，它是將變更提升至生產環境的動作。

重新定位

見 7 盧比

再平台

見 7 盧比

買回

見 7 盧比

彈性

應用程式抵抗或從中斷中復原的能力。在規劃備援時，高可用性和災難復原是常見的考量因素。AWS 雲端如需詳細資訊，請參閱AWS 雲端 復原力。

資源型政策

附接至資源的政策，例如 Amazon S3 儲存貯體、端點或加密金鑰。這種類型的政策會指定允許存取哪些主體、支援的動作以及必須滿足的任何其他條件。

負責者、當責者、事先諮詢者和事後告知者 (RACI) 矩陣

定義移轉活動和雲端作業所涉及之所有各方的角色與責任的矩陣。矩陣名稱衍生自矩陣中定義的責任型別：負責 (R)、負責 (A)、諮詢 (C) 及通知 (I)。支撐（S）類型是可選的。如果您包含支援，則該矩陣稱為 RASCI 矩陣，如果您將其排除，則稱為 R ACI 矩陣。

回應性控制

一種安全控制，旨在驅動不良事件或偏離安全基準的補救措施。如需詳細資訊，請參閱在 AWS上實作安全控制中的回應性控制。

保留

見 7 盧比

退休

見 7 盧比

旋轉

定期更新密碼以使攻擊者更難以存取認證的程序。

資料列與資料行存取控制 (RCAC)

使用已定義存取規則的基本、彈性 SQL 運算式。RCAC 由資料列權限和資料行遮罩所組成。

RPO

請參閱復原點目標。

RTO

請參閱復原時間目標。

執行手冊

執行特定任務所需的一組手動或自動程序。這些通常是為了簡化重複性操作或錯誤率較高的程序而建置。

S

SAML 2.0

許多身份提供者（IdPs）使用的開放標準。此功能可啟用聯合單一登入 (SSO)，因此使用者可以登入 AWS Management Console 或呼叫 AWS API 作業，而不必為組織中的每個人在 IAM 中建立使用者。如需有關以 SAML 2.0 為基礎的聯合詳細資訊，請參閱 IAM 文件中的關於以 SAML 2.0 為基礎的聯合。

斯卡達

請參閱監督控制和資料擷取。

SCP

請參閱服務控制策略。

秘密

您以加密形式儲存的機密或受限制資訊，例如密碼或使用者認證。 AWS Secrets Manager它由秘密值及其中繼資料組成。密碼值可以是二進位、單一字串或多個字串。如需詳細資訊，請參閱「Secrets Manager 碼中有什麼內容？ 在 Secrets Manager 文檔中。

安全控制

一種技術或管理防護機制，它可預防、偵測或降低威脅行為者利用安全漏洞的能力。安全性控制有四種主要類型：預防性、偵測、回應式和主動式。

安全強化

減少受攻擊面以使其更能抵抗攻擊的過程。這可能包括一些動作，例如移除不再需要的資源、實作授予最低權限的安全最佳實務、或停用組態檔案中不必要的功能。

安全資訊與事件管理 (SIEM) 系統

結合安全資訊管理 (SIM) 和安全事件管理 (SEM) 系統的工具與服務。SIEM 系統會收集、監控和分析來自伺服器、網路、裝置和其他來源的資料，以偵測威脅和安全漏洞，並產生提醒。

安全回應自動化

預先定義且程式化的動作，其設計用來自動回應或修復安全性事件。這些自動化作業可做為偵探或回應式安全控制項，協助您實作 AWS 安全性最佳實務。自動回應動作的範例包括修改 VPC 安全群組、修補 Amazon EC2 執行個體或輪換登入資料。

伺服器端加密

在其目的地的數據加密，通 AWS 服務 過接收它。

服務控制政策 (SCP)

為 AWS Organizations中的組織的所有帳戶提供集中控制許可的政策。SCP 會定義防護機制或設定管理員可委派給使用者或角色的動作限制。您可以使用 SCP 作為允許清單或拒絕清單，以指定允許或禁止哪些服務或動作。如需詳細資訊，請參閱 AWS Organizations 文件中的服務控制原則。

服務端點

的進入點的 URL AWS 服務。您可以使用端點，透過程式設計方式連接至目標服務。如需詳細資訊，請參閱 AWS 一般參考 中的 AWS 服務 端點。

服務水準協議 (SLA)

一份協議，闡明 IT 團隊承諾向客戶提供的服務，例如服務正常執行時間和效能。

服務等級指示器 (SLI)

對服務效能層面的測量，例如錯誤率、可用性或輸送量。

服務等級目標 (SLO)

代表服務狀況的目標測量結果，由服務層次指示器測量。

共同責任模式

描述您在雲端安全性和合規方面共享的責任的模型。 AWS AWS 負責雲端的安全性，而您則負責雲端的安全性。如需詳細資訊，請參閱共同責任模式。

暹

請參閱安全性資訊和事件管理系統。

單點故障 (SPF)

應用程式的單一重要元件發生故障，可能會中斷系統。

SLA

請參閱服務等級協議。

SLI

請參閱服務層級指示器。

SLO

請參閱服務等級目標。

split-and-seed 模型

擴展和加速現代化專案的模式。定義新功能和產品版本時，核心團隊會進行拆分以建立新的產品團隊。這有助於擴展組織的能力和服務，提高開發人員生產力，並支援快速創新。如需詳細資訊，請參閱中的應用程式現代化的階段化方法。 AWS 雲端

痙攣

請參閱單一故障點。

星型綱要

使用一個大型事實資料表來儲存交易或測量資料，並使用一或多個較小的維度表格來儲存資料屬性的資料庫組織結構。這種結構是專為在數據倉庫中使用或用於商業智能目的。

Strangler Fig 模式

一種現代化單一系統的方法，它會逐步重寫和取代系統功能，直到舊式系統停止使用為止。此模式源自無花果藤，它長成一棵馴化樹並最終戰勝且取代了其宿主。該模式由 Martin Fowler 引入，作為重寫單一系統時管理風險的方式。如需有關如何套用此模式的範例，請參閱使用容器和 Amazon API Gateway 逐步現代化舊版 Microsoft ASP.NET (ASMX) Web 服務。

子網

您 VPC 中的 IP 地址範圍。子網必須位於單一可用區域。

監督控制與資料擷取 (SCADA)

在製造業中，使用硬體與軟體來監控實體資產與生產作業的系統。

對稱加密

使用相同金鑰來加密及解密資料的加密演算法。

合成測試

以模擬使用者互動以偵測潛在問題或監控效能的方式測試系統。您可以使用 Amazon CloudWatch Synthetics 來創建這些測試。

T

標籤

作為組織 AWS 資源的中繼資料的索引鍵值配對。標籤可協助您管理、識別、組織、搜尋及篩選資源。如需詳細資訊，請參閱標記您的 AWS 資源。

目標變數

您嘗試在受監督的 ML 中預測的值。這也被稱為結果變數。例如，在製造設定中，目標變數可能是產品瑕疵。

任務清單

用於透過執行手冊追蹤進度的工具。任務清單包含執行手冊的概觀以及要完成的一般任務清單。對於每個一般任務，它包括所需的預估時間量、擁有者和進度。

測試環境

請參閱環境。

訓練

為 ML 模型提供資料以供學習。訓練資料必須包含正確答案。學習演算法會在訓練資料中尋找將輸入資料屬性映射至目標的模式 (您想要預測的答案)。它會輸出擷取這些模式的 ML 模型。可以使用 ML 模型，來預測您不知道的目標新資料。

傳輸閘道

可以用於互連 VPC 和內部部署網路的網路傳輸中樞。如需詳細資訊，請參閱 AWS Transit Gateway 文件中的傳輸閘道是什麼。

主幹型工作流程

這是一種方法，開發人員可在功能分支中本地建置和測試功能，然後將這些變更合併到主要分支中。然後，主要分支會依序建置到開發環境、生產前環境和生產環境中。

受信任的存取權

授與權限給您指定的服務，以代表您在組織內 AWS Organizations 及其帳戶中執行工作。受信任的服務會在需要該角色時，在每個帳戶中建立服務連結角色，以便為您執行管理工作。如需詳細資訊，請參閱 AWS Organizations 文件中的AWS Organizations 與其他 AWS 服務搭配使用。

調校

變更訓練程序的各個層面，以提高 ML 模型的準確性。例如，可以透過產生標籤集、新增標籤、然後在不同的設定下多次重複這些步驟來訓練 ML 模型，以優化模型。

雙比薩團隊

一個小 DevOps 團隊，你可以餵兩個比薩餅。雙披薩團隊規模可確保軟體開發中的最佳協作。

U

不確定性

這是一個概念，指的是不精確、不完整或未知的資訊，其可能會破壞預測性 ML 模型的可靠性。有兩種類型的不確定性：認知不確定性是由有限的、不完整的資料引起的，而隨機不確定性是由資料中固有的噪聲和隨機性引起的。如需詳細資訊，請參閱量化深度學習系統的不確定性指南。

無差別的任務

也稱為繁重工作，是創建和操作應用程序所必需的工作，但不能為最終用戶提供直接價值或提供競爭優勢。無差異化作業的範例包括採購、維護和容量規劃。

較高的環境

請參閱環境。

V

清空

一種資料庫維護操作，涉及增量更新後的清理工作，以回收儲存並提升效能。

版本控制

追蹤變更的程序和工具，例如儲存庫中原始程式碼的變更。

VPC 對等互連

兩個 VPC 之間的連線，可讓您使用私有 IP 地址路由流量。如需詳細資訊，請參閱 Amazon VPC 文件中的什麼是 VPC 對等互連。

漏洞

會危及系統安全性的軟體或硬體瑕疵。

W

暖快取

包含經常存取的目前相關資料的緩衝快取。資料庫執行個體可以從緩衝快取讀取，這比從主記憶體或磁碟讀取更快。

溫暖的數據

不常存取的資料。查詢此類資料時，通常可以接受中度緩慢的查詢。

視窗功能

一種 SQL 函數，可對以某種方式與當前記錄相關的一組行執行計算。視窗函數對於處理工作非常有用，例如計算移動平均值或根據目前列的相對位置存取列的值。

工作負載

提供商業價值的資源和程式碼集合，例如面向客戶的應用程式或後端流程。

工作串流

遷移專案中負責一組特定任務的功能群組。每個工作串流都是獨立的，但支援專案中的其他工作串流。例如，組合工作串流負責排定應用程式、波次規劃和收集遷移中繼資料的優先順序。組合工作串流將這些資產交付至遷移工作串流，然後再遷移伺服器和應用程式。

蠕蟲

看到寫一次，多讀。

WQF

請參閱 AWS 工作負載鑑定架構。

寫一次，多讀（WORM）

一種儲存模型，可單次寫入資料並防止資料遭到刪除或修改。授權用戶可以根據需要多次讀取數據，但無法更改數據。這種數據存儲基礎設施被認為是不可變的。

Z

零日漏洞

一種利用零時差漏洞的攻擊，通常是惡意軟件。

零時差漏洞

生產系統中未緩解的瑕疵或弱點。威脅參與者可以利用這種類型的漏洞攻擊系統。由於攻擊，開發人員經常意識到該漏洞。

殭屍應用程式

CPU 和記憶體平均使用率低於 5% 的應用程式。在遷移專案中，通常會淘汰這些應用程式。

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。