Understanding allow policies

Archived: 2026-04-05 17:52:46 UTC

You can grant access to Google Cloud resources by using *allow policies*, also known as *Identity and Access Management (IAM) policies*, which are attached to resources. You can attach only one allow policy to each resource. The allow policy controls access to the resource itself, as well as any descendants of that resource that inherit the allow policy.

This page shows allow policies in JSON format. You can also use the Google Cloud CLI to retrieve allow policies in YAML format.

Policy structure

An allow policy is a collection of *role bindings* and metadata. A role binding specifies what access should be granted to a resource. It associates, or *binds*, one or more *principals* with a single IAM *role* and any context-specific *conditions* that change how and when the role is granted. The metadata includes additional information about the allow policy, such as an *etag* and *version* to facilitate policy management.

Each role binding can include the following fields:

- One or more principals, also known as *members* or *identities*. There are multiple types of principals, including single users, groups of users, and service accounts. For a full list of supported principal types, see Principal types.
- A role, which is a named collection of permissions that provide the ability to perform actions on Google Cloud resources.
- A condition, which is an optional logic expression that further constrains the role binding based on attributes about the request, such as its origin or the target resource. Conditions are typically used to control whether access is granted based on the context for a request.

  If a role binding contains a condition, it is referred to as a *conditional role binding*.

  Some Google Cloud services don't accept conditions in allow policies. For a list of services and resource types that accept conditions, see Resource types that accept conditional role bindings.

Changes to a principal's access are eventually consistent. This means that it takes time for access changes to propagate through the system. To learn how long it takes, on average, for access changes to propagate, see Access change propagation.

Limits on all principals

Each allow policy can contain up to 1,500 principals. For the purposes of this limit, IAM counts *all* appearances of each principal in the allow policy's role bindings, as well as the principals that the allow policy exempts from

Data Access audit logging. It does *not* deduplicate principals that appear in more than one role binding. For example, if an allow policy contains only role bindings for the principal `user:my-user@example.com`, and this principal appears in 50 role bindings, then you can add another 1,450 principals to the role bindings in the allow policy.

Also, for the purposes of this limit, each appearance of a domain or Google group is counted as a single principal, regardless of the number of individual members in the domain or group.

If you use IAM Conditions, or if you grant roles to many principals with unusually long identifiers, then IAM might allow fewer principals in the allow policy.

Limits on groups and domains

Up to 250 of the principals in an allow policy can be Google groups, Cloud Identity domains, or Google Workspace accounts.

For the purposes of this limit, Cloud Identity domains, Google Workspace accounts, and Google groups are counted as follows:

- For Google groups, each unique group is counted only once, regardless of how many times the group appears in the allow policy. This is different from how groups are counted for the limit on the total number of principals in an allow policy—for that limit, each appearance of a group counts towards the limit.
- For Cloud Identity domains or Google Workspace accounts, IAM counts *all* appearances of each domain or account in the allow policy's role bindings. It does *not* deduplicate domains or accounts that appear in more than one role binding.

For example, if your allow policy contains only one group, `group:my-group@example.com`, and the group appears in the allow policy 10 times, then you can add another 249 Cloud Identity domains, Google Workspace accounts, or unique groups before you reach the limit.

Alternatively, if your allow policy contains only one domain, `domain:example.com`, and the domain appears in the allow policy 10 times, then you can add another 240 Cloud Identity domains, Google Workspace accounts, or unique groups before you reach the limit.

Policy metadata

The metadata for an allow policy includes the following fields:

- An `etag` field, which is used for concurrency control, and ensures that allow policies are updated consistently. For details, see Using etags in a policy on this page.
- A `version` field, which specifies the schema version for a given allow policy. For details, see Policy versions on this page.

For organizations, folders, projects, and billing accounts, the allow policy can also contain an `auditConfig` field, which specifies the types of activity that generate audit logs for each service. To learn how to configure this part of an allow policy, see Configuring Data Access audit logs.

Using etags in a policy

When multiple systems try to write to the same allow policy at the same time, there is a risk that those systems might overwrite each other's changes. This risk exists because allow policies are updated using the *read-modify-write* pattern, which involves multiple operations:

1. **Reading** the existing allow policy
2. **Modifying** the allow policy
3. **Writing** the entire allow policy

If System A reads an allow policy, and System B immediately writes an updated version of that allow policy, then System A will not be aware of the changes from System B. When System A writes its own changes to the allow policy, System B's changes could be lost.

To help prevent this issue, Identity and Access Management (IAM) supports concurrency control through the use of an `etag` field in the allow policy. Every allow policy contains an `etag` field, and the value of this field changes each time an allow policy is updated. If an allow policy contains an `etag` field, but no role bindings, then the allow policy does not grant any IAM roles.

The `etag` field contains a value such as `BwUjMhCsNvY=`. When you update the allow policy, be sure to include the `etag` field in the updated allow policy. If the allow policy has been modified since you retrieved it, the `etag` value will not match, and the update will fail. For the REST API, you receive the HTTP status code `409 Conflict`, and the response body is similar to the following:

```
{
 "error": {
   "code": 409,
   "message": "There were concurrent policy changes. Please retry the whole read-modify-write with exponential
   "status": "ABORTED"
 }
}
```

If you receive this error, retry the entire series of operations: read the allow policy again, modify it as needed, and write the updated allow policy. You should perform retries automatically, with exponential backoff, in any tools that you use to manage allow policies.

Example: Simple policy

Consider the following allow policy that binds a principal to a role:

```
{
 "bindings": [
   {
     "members": [
```

```
        "user:jie@example.com"
      ],
      "role": "roles/owner"
    }
  ],
  "etag": "BwUjMhCsNvY=",
  "version": 1
}
```

In the preceding example, Jie is granted the Owner basic role without any conditions. This role gives Jie almost unlimited access.

Example: Policy with multiple role bindings

Consider the following allow policy that contains more than one role binding. Each role binding grants a different role:

```
{
 "bindings": [
   {
     "members": [
       "user:jie@example.com"
     ],
     "role": "roles/resourcemanager.organizationAdmin"
   },
   {
     "members": [
       "user:raha@example.com",
       "user:jie@example.com"
     ],
     "role": "roles/resourcemanager.projectCreator"
   }
 ],
 "etag": "BwUjMhCsNvY=",
 "version": 1
}
```

In the preceding example, Jie is granted the Organization Admin predefined role (`roles/resourcemanager.organizationAdmin`) in the first role binding. This role contains permissions for organizations, folders, and limited projects operations. In the second role binding, both Jie and Raha are granted the ability to create projects by the Project Creator role (`roles/resourcemanager.projectCreator`). Together, these role bindings grant fine-grained access to both Jie and Raha, and Jie is granted more access than Raha.

Example: Policy with conditional role binding

Consider the following allow policy, which binds principals to a predefined role and uses a condition expression to constrain the role binding:

```
{
 "bindings": [
   {
     "members": [
       "group:prod-dev@example.com",
       "serviceAccount:prod-dev-example@appspot.gserviceaccount.com"
     ],
     "role": "roles/appengine.deployer",
     "condition": {
         "title": "Expires_July_1_2022",
         "description": "Expires on July 1, 2022",
         "expression":
           "request.time < timestamp('2022-07-01T00:00:00.000Z')"
     }
   }
 ],
 "etag": "BwWKmjvelug=",
 "version": 3
}
```

In this example, the `version` field is set to `3`, because the allow policy contains a condition expression. The role binding in the allow policy is conditional; it grants the role to the `prod-dev` group and the service account `prod-dev-example@appspot.gserviceaccount.com`, but only until July 1, 2022.

For details about the features that each allow policy version supports, see Policy versions on this page.

Example: Policy with conditional and unconditional role bindings

Consider the following allow policy, which contains both conditional and unconditional role bindings for the same role:

```
{
 "bindings": [
   {
     "members": [
       "serviceAccount:prod-dev-example@appspot.gserviceaccount.com"
      ],
      "role": "roles/appengine.deployer"
   },
   {
     "members": [
```

```
        "group:prod-dev@example.com",
        "serviceAccount:prod-dev-example@appspot.gserviceaccount.com"
      ],
      "role": "roles/appengine.deployer",
      "condition": {
        "title": "Expires_July_1_2022",
        "description": "Expires on July 1, 2022",
        "expression":
          "request.time < timestamp('2022-07-01T00:00:00.000Z')"
      }
    }
  ],
  "etag": "BwWKmjvelug=",
  "version": 3
}
```

In this example, the service account `serviceAccount:prod-dev-example@appspot.gserviceaccount.com` is included in two role bindings for the same role. The first role binding does not have a condition. The second role binding has a condition that only grants the role until July 1, 2022.

Effectively, this allow policy always grants the role to the service account. In IAM, conditional role bindings don't override role bindings with no conditions. If a principal is bound to a role, and the role binding does not have a condition, then the principal always has that role. Adding the principal to a conditional role binding for the same role has no effect.

In contrast, the `prod-dev` group is included only in the conditional role binding. Therefore, it has the role only before July 1, 2022.

Example: Policy that binds a role to a deleted principal

Consider the following allow policy. This allow policy binds a role to a service account, `serviceAccount:my-service-account@my-project.iam.gserviceaccount.com`, which was deleted. As a result, the service account's identifier now has a `deleted:` prefix:

```
{
 "bindings": [
   {
     "members": [
       "deleted:serviceAccount:my-service-account@my-project.iam.gserviceaccount.com?uid=123456789012345678901"
     ],
     "role": "roles/owner"
   }
 ],
 "etag": "BwUjMhCsNvY=",
 "version": 1
```

```
}
```

If you create a new service account with the same name, the allow policy's role bindings for the deleted service account don't apply to the new service account. This behavior applies for all types of deleted principals.

This behavior prevents new principals from inheriting roles that were granted to deleted principals. If you want to grant roles to the new principal, add the new principal to the allow policy's role bindings, as shown in Policies with deleted principals on this page.

All deleted principals have the prefix `deleted:`. Some types of deleted principals, like service accounts, also have the suffix `?uid=numeric-id`, where *`numeric-id`* is the deleted principal's unique numeric ID. In this example, instead of `serviceAccount:serviceAccount:my-service-account@my-project.iam.gserviceaccount.com`, the allow policy shows the identifier `deleted:serviceAccount:my-service-account@my-project.iam.gserviceaccount.com?uid=123456789012345678901`.

Default policies

All resources that accept allow policies are created with default allow policies. Most resources' default allow policies are empty. However, some resources' default allow policies automatically contain certain role bindings. For example, when you create a new project, the allow policy for the project automatically has a role binding that grants you the Owner role (`roles/owner`) on the project.

These role bindings are created by the system, so the user doesn't need `getIamPolicy` or `setIamPolicy` permissions on the resource for the role bindings to be created.

To learn if a resource is created with an allow policy, refer to the resource's documentation.

Policy inheritance and the resource hierarchy

Google Cloud resources are organized hierarchically, where the organization node is the root node in the hierarchy, then optionally folders, then projects. Most of other resources are created and managed under a project. Each resource has exactly one parent, except the organization. The organization, as the root node in the hierarchy, has no parent. See the Resource Hierarchy topic for more information.

The resource hierarchy is important to consider when setting an allow policy. When setting an allow policy at a higher level in the hierarchy, such as at the organization level, folder level, or project level, the granted access scope includes the resource level where this allow policy is attached to and all resources under that level. For example, an allow policy set at the organization level applies to the organization and all resources under the organization. Similarly, an allow policy set at the project level applies to the project and all resources in the project.

Policy inheritance is the term that describes how allow policies apply to resources beneath their level in the resource hierarchy. *Effective policy* is the term that describes how all parent allow policies in the resource hierarchy are inherited for a resource. It is the union of the following:

- The allow policy set on the resource
- The allow policies set on all of resource's ancestry resource levels in the hierarchy

Each new role binding (inherited from parent resources) that affect the resource's effective allow policy are evaluated independently. A specific access request to the resource is granted if any of the higher-level role bindings grant access to the request.

If a new role binding is introduced to any level of a resource's inherited allow policy, the access grant scope increases.

Example: Policy inheritance

To understand allow policy inheritance, consider a scenario where you grant a user, Raha, two different IAM roles at two different levels in the resource hierarchy.

To grant Raha a role at the **organization level**, you set the following allow policy on your organization:

```
{
 "bindings": [
   {
     "members": [
       "user:raha@example.com"
     ],
     "role": "roles/storage.objectViewer"
   }
 ],
 "etag": "BwUjMhCsNvY=",
 "version": 1
}
```

This allow policy grants Raha the Storage Object Viewer role (`roles/storage.objectViewer`), which contains `get` and `list` permissions for projects and Cloud Storage objects. Because you set the allow policy on your organization, Raha can use these permissions for all projects and all Cloud Storage objects in your organization.

To grant Raha a role at the **project level**, you set the following allow policy on the project `myproject-123` :

```
{
 "bindings": [
   {
     "members": [
       "user:raha@example.com"
     ],
     "role": "roles/storage.objectCreator"
   }
 ],
```

```
  "etag": "BwUjMhCsNvY=",
  "version": 1
}
```

This allow policy grants Raha the Storage Object Creator role (`roles/storage.objectCreator`), which lets them create Cloud Storage objects. Because you set the allow policy on `myproject-123` , Raha can create Cloud Storage objects only in `myproject-123` .

Now that there are two role bindings that grant Raha access to the target Cloud Storage objects under `myproject-123` , the following allow policies apply:

- An allow policy at the organization level grants the ability to list and get all Cloud Storage objects under this organization.
- An allow policy at the project level, for the project `myproject-123` , grants the ability to create objects within that project.

The following table summarizes Raha's effective policy:

Permissions from the Storage Object Viewer role on the organization	**Permissions from the Storage Object Creator role on `myproject-123`**	**Effective permissions for Raha on `myproject-123`**
resourcemanager.projects.get resourcemanager.projects.list storage.objects.get storage.objects.list	resourcemanager.projects.get resourcemanager.projects.list storage.objects.create	resourcemanager.projects.get resourcemanager.projects.list storage.objects.get storage.objects.list storage.objects.create

Policy versions

Over time, IAM might add new features that significantly add or change fields in the allow policy schema. To avoid breaking your existing integrations that rely on consistency in the allow policy structure, such changes are introduced in new allow policy schema versions.

If you are integrating with IAM for the first time, we recommend using the most recent allow policy schema version available at that time. The following section discusses the different versions available and how to use each. It also describes how to specify a policy version and discusses some troubleshooting scenarios.

Every existing allow policy specifies a `version` field as part of the allow policy's metadata. Consider the highlighted portion of the following sample:

```
{
  "bindings": [
    {
```

```
      "members": [
        "user:jie@example.com"
      ],
      "role": "roles/owner"
    }
  ],
  "etag": "BwUjMhCsNvY=",
  "version": 1
}
```

This field specifies the *syntax schema version* of the allow policy. Each version of the allow policy contains a specific syntax schema that can be used by role bindings. The newer version can contain role bindings with the newer syntax schema that is unsupported by earlier versions. This field is not intended to be used for any purposes other than controlling the syntax schema for the allow policy.

Valid policy versions

Allow policies can use the following allow policy versions:

Version	Description
`1`	The first version of the IAM syntax schema for policies. Supports binding one role to one or more principals. Does not support conditional role bindings.
`2`	Reserved for internal use.
`3`	Introduces the `condition` field in the role binding, which constrains the role binding using context-based and attribute-based rules. For more information, see the overview of IAM Conditions.

Specifying a policy version when getting a policy

For the REST API and client libraries, when you get an allow policy, we recommend that you specify an allow policy version in the request. When a request specifies an allow policy version, IAM assumes that the caller is aware of the features in that allow policy version and can handle them correctly.

If the request does not specify an allow policy version, IAM assumes that the caller wants a version `1` allow policy.

When you get an allow policy, IAM checks the allow policy version in the request, or the default version if the request did not specify a version. IAM also checks the allow policy for fields that are not supported in a version `1` allow policy. It uses this information to decide what type of response to send:

- If the allow policy does not contain any conditions, then IAM always returns a version `1` allow policy, regardless of the version number in the request.

- If the allow policy contains conditions, and the caller requested a version `3` allow policy, then IAM returns a version `3` allow policy that includes the conditions. For an example, see scenario 1 on this page.
- If the allow policy contains conditions, and the caller requested a version `1` allow policy or did not specify a version, then IAM returns a version `1` allow policy.

  For role bindings that include a condition, IAM appends the string `_withcond_` to the role name, followed by a hash value; for example, `roles/iam.serviceAccountAdmin_withcond_2b17cc25d2cd9e2c54d8`. The condition itself is not present. For an example, see scenario 2 on this page.

Scenario 1: Policy version that fully supports IAM Conditions

Suppose you call the following REST API method to get the allow policy for a project:

```
POST https://cloudresourcemanager.googleapis.com/v1/projects/project-id:getIamPolicy
```

The request body contains the following text:

```
{
 "options": {
   "requestedPolicyVersion": 3
 }
}
```

The response contains the project's allow policy. If the allow policy contains at least one conditional role binding, its `version` field is set to `3`:

```
{
 "bindings": [
   {
     "members": [
       "user:tal@example.com"
     ],
     "role": "roles/iam.securityReviewer",
     "condition": {
         "title": "Expires_July_1_2022",
         "description": "Expires on July 1, 2022",
         "expression": "request.time < timestamp('2022-07-01T00:00:00.000Z')"
     }
   }
 ],
 "etag": "BwWKmjvelug=",
```

```
  "version": 3
}
```

If the allow policy does not contain conditional role bindings, its `version` field is set to `1`, even though the request specified version `3`:

```
{
 "bindings": [
   {
     "members": [
       "user:tal@example.com"
     ],
     "role": "roles/iam.securityReviewer",
   }
 ],
 "etag": "BwWKmjvelug=",
 "version": 1
}
```

Scenario 2: Policy version with limited support for IAM Conditions

Suppose you call the following REST API method to get the allow policy for a project:

```
POST https://cloudresourcemanager.googleapis.com/v1/projects/project-id:getIamPolicy
```

The request body is empty; it does not specify a version number. As a result, IAM uses the default allow policy version, `1`.

The allow policy contains a conditional role binding. Because the allow policy version is `1`, the condition does not appear in the response. To indicate that the role binding uses a condition, IAM appends the string `_withcond_` to the role name, followed by a hash value:

```
{
 "bindings": [
   {
     "members": [
       "user:tal@example.com"
     ],
     "role": "roles/iam.securityReviewer_withcond_58e135cabb940ad9346c"
   }
 ],
 "etag": "BwWKmjvelug=",
```

```
  "version": 1
}
```

Specifying a policy version when setting a policy

When you set an allow policy, we recommend that you specify an allow policy version in the request. When a request specifies an allow policy version, IAM assumes that the caller is aware of the features in that allow policy version and can handle them correctly.

If the request does not specify an allow policy version, IAM accepts only the fields that can appear in a version `1` allow policy. As a best practice, don't change conditional role bindings in a version `1` allow policy; because the allow policy does not show the condition for each role binding, you don't know when or where the role binding is actually granted. Instead, get the version `3` representation of the allow policy, which shows the condition for each role binding, and use that representation to update the role bindings.

Scenario: Policy versions in requests and responses

Suppose you use the REST API to get an allow policy, and you specify version `3` in the request. The response contains the following allow policy, which uses version `3`:

```
{
 "bindings": [
   {
     "members": [
       "user:raha@example.com"
     ],
     "role": "roles/storage.admin",
     "condition": {
         "title": "Weekday_access",
         "description": "Monday thru Friday access only in America/Chicago",
         "expression": "request.time.getDayOfWeek('America/Chicago') >= 1 && request.time.getDayOfWeek('America
     }
   }
 ],
 "etag": "BwUjMhCsNvY=",
 "version": 3
}
```

You decide that Raha should have the Storage Admin role (`roles/storage.admin`) throughout the week, not just on weekdays. You remove the condition from the role binding and send a REST API request to set the allow policy; once again, you specify version `3` in the request:

```
{
 "bindings": [
   {
```

```
      "members": [
        "user:raha@example.com"
      ],
      "role": "roles/storage.admin"
    }
  ],
  "etag": "BwUjMhCsNvY=",
  "version": 3
}
```

The response contains the updated allow policy:

```
{
 "bindings": [
   {
     "members": [
       "user:raha@example.com"
     ],
     "role": "roles/storage.admin"
   }
 ],
 "etag": "BwWd8I+ZUAQ=",
 "version": 1
}
```

The allow policy in the response uses version `1`, even though the request specified version `3`, because the allow policy uses only fields that are supported in a version `1` allow policy.

Policies with deleted principals

If a role binding in an allow policy includes a deleted principal, and you add a role binding for a new principal with the same name, the role binding is always applied to the new principal.

For example, consider this allow policy that includes a role binding for a deleted service account, `my-service-account@project-id.iam.gserviceaccount.com`. As a result, the identifier for each the service account has a `deleted:` prefix:

```
{
 "bindings": [
   {
     "members": [
       "deleted:serviceAccount:my-service-account@project-id.iam.gserviceaccount.com?uid=123456789012345678901"
     ],
     "role": "roles/owner"
```

```
    }
  ],
  "etag": "BwUjMhCsNvY=",
  "version": 1
}
```

Suppose you create a new service account that is also named `my-service-account@project-id.iam.gserviceaccount.com`, and you want to grant it the Project Creator role (`roles/resourcemanager.projectCreator`). To grant the role to the new service account, update the allow policy as shown in this example:

```
{
  "bindings": [
    {
      "members": [
        "deleted:serviceAccount:my-service-account@project-id.iam.gserviceaccount.com?uid=1234567890
      ],
      "role": "roles/owner"
    },
    {
      "members": [
        "serviceAccount:my-service-account@project-id.iam.gserviceaccount.com"
      ],
      "role": "roles/resourcemanager.projectCreator"
    }
  ],
  "etag": "BwUjMhCsNvY=",
  "version": 1
}
```

To make it easier to audit your IAM allow policies, you can also remove the deleted user from the role binding to the Owner role:

```
{
  "bindings": [
    {
      "members": [
        "deleted:serviceAccount:my-service-account@project-id.iam.gserviceaccount.com?uid=1234567890
      ],
      "role": "roles/owner"
    },
    {
      "members": [
        "user:donald@example.com"
```

```
      ],
      "role": "roles/resourcemanager.projectCreator"
    }
  ],
  "etag": "BwUjMhCsNvY=",
  "version": 1
}
```

Policy best practices

The following best practices apply to organizations with many Google Cloud users:

- When managing multiple principals with the same access configurations, use groups instead. Put each individual principal into the group, and grant the intended roles to the group instead of individual user accounts principals.
- **Roles granted at the organization level**: Carefully consider which principals are granted roles at the organization level. For most organizations, only a few specific teams, such as Security and Network teams, should be granted access at this level of the resource hierarchy.
- **Roles granted at the folder levels**: Consider reflecting your organization's operation structure by using tiers of folders, where each folder can be configured with different sets of access grants that are aligned with business and operation needs. For example, a parent folder might reflect a department, one of its child folder might reflect resource access and operation by a group, and another child folder might reflect a small team. Both of these two folders might contain projects for their team's operation needs. Using folders in this way can ensure proper access separation, while respecting allow policies inherited from parent folders and the organization. This practice requires less maintenance of allow policies when creating and managing Google Cloud resources.
- **Roles granted at the project level**: Grant role bindings at the project level when necessary to follow the principle of least privilege. For example, if a principal needs access to 3 of the 10 projects in a folder, you should grant access to each of the 3 projects individually; in contrast, if you granted a role on the folder, the principal would gain access that they don't need to another 7 projects.

  Alternatively, you can use IAM Conditions to grant roles at the organization or folder level, but only for a subset of folders or projects.
- **Only grant access to principals within your domain**: To improve your organization's security, don't grant roles to principals outside of your domain. You can enforce this best practice by enforcing the `iam.allowedPolicyMemberDomains` organization policy constraint.

What's next

- Learn how to troubleshoot allow policies that contain the string `withcond` in role names.
- Find out how to manage the role bindings in an allow policy.

- Get an overview of IAM Conditions, which use version `3` allow policies.
- Explore the Policy Intelligence tools, which help you understand and manage your allow policies to proactively improve your security configuration.
- Use the Cloud Asset API to search allow policies.
- Use the Cloud Asset API to view effective allow policies.
