Enforcing Least Privilege Cross-Cloud Resource Access for Cloud Orchestrators

Alec Li

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-61
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-61.html

May 14, 2025

Enforcing Least Privilege Cross-Cloud Resource Access for Cloud Orchestrators

by

Alec Li

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Raluca Ada Popa, Chair
Professor Ion Stoica

Spring 2025

The thesis of Alec Li, titled Enforcing Least Privilege Cross-Cloud Resource Access for Cloud Orchestrators, is approved:

Chair	Raluca Ada Popa	Date	May 12, 2025
	Ion Stoica	Date	May 14, 2025
		Date	

University of California, Berkeley

Enforcing Least Privilege Cross-Cloud Resource Access for Cloud Orchestrators

Abstract

Enforcing Least Privilege Cross-Cloud Resource Access for Cloud Orchestrators

by

Alec Li

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Raluca Ada Popa, Chair

As cloud computing systems evolve over time, there has been an increased dependency on systems that span across multiple cloud providers, leading to the increased usage of *workload orchestrator* services, to assist with the deployment and management of workloads among multiple clouds. However, the workload orchestrators that exist today all require the end user to disclose their cloud credentials—this means that an adversary that compromises a workload orchestrator can access resources in the user's cloud.

Recently, Skydentity solves one aspect of this security issue, by introducing a system that protects against orchestrator compromise, ensuring that workload orchestrators never hold any cloud credentials, and utilizing proxies that enforce fine-grained user-specified authorization policies. However, VMs created through Skydentity do not have the ability to request resources *across clouds*, limiting the scope of workloads that can utilize Skydentity.

We introduce an extension of Skydentity that allows for VMs created by workload orchestrators to access resources across clouds, while maintaining the security guarantees of Skydentity, protecting against orchestrator compromise. Our prototype introduces an added latency of at most 3% during VM creation, and has negligible effect on subsequent cross-cloud resource requests.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor, Professor Raluca Ada Popa, for her advice and guidance throughout this project. I would also like to thank my mentor, Samyu Yagati, for welcoming me into SkyLab, and for her mentorship throughout the entire research process. Thank you both for your continual support throughout my research journey.

Additionally, I would also like to thank my family for being a constant presence in my life, always supporting me through all of my endeavors and dreams.

Lastly, thank you to all of my professors, colleagues, peers, students, and friends for supporting me during my time at Berkeley. Your unwavering support has made my experience at Berkeley truly memorable and unforgettable—I will forever cherish these years.

Chapter 1

Introduction

1.1 Preface

Cloud computing systems have continually evolved over time; more recently, there has been an increased dependency on systems that span across *multiple* cloud providers—this is usually done to optimize for latency, cost, and resources available for each cloud environment. This has brought forth new kinds of services that manage these cross-cloud workloads, and with it, the need for security in these new kinds of systems.

These *workload orchestrator* services, like SkyPilot [23], Terraform [15], Astran [5], among others [6, 7, 20] assist with the deployment and management of workloads within a single or among multiple clouds. For example, in Terraform [15], developers create a infrastructure specification document, which is used by Terraform to provision and deploy resources in their cloud environments. In SkyPilot [23], developers specify virtual machine (VM) configurations, along with a specification for the workload that should be run; SkyPilot automatically provisions the VM cluster in the cloud and runs the workload from the specification, optimizing for cost and resource availability.

However, the workload orchestrators that exist today all require the end user to provide their cloud credentials, which are stored and used by the workload orchestrator in its operation. For example, in Terraform [15] (more specifically, HCP Terraform, which is a hosted service for Terraform), users must upload their cloud credentials/API tokens, and in SkyPilot [23], users grant the orchestrator a wide range of permissions to compute and IAM resources.

All of this means that the orchestrator is a primary target of attack for adversaries; compromising the orchestrator would result in the compromise of the user's entire cloud and all of the resources within it.

Skydentity [22] solves one aspect of this security issue: it introduces a system that protects against orchestrator compromise, by ensuring that workload orchestrators never hold any cloud credentials. Skydentity guarantees that even if a workload orchestrator is compromised, user resources are kept secure, aside from the fine-grained authorizations given to the orchestrator through a user-specified authorization policy.

One aspect not solved by Skydentity is resource access for VMs created through this system: the VMs that the orchestrator creates only hold credentials for resources in the cloud it is created in. In many workloads, it can be natural for machines to access resources in different clouds (especially if the workload itself is being deployed across multiple clouds); Skydentity does not natively allow for resource access *across clouds*, because the VMs do not hold credentials for other cloud environments.

Our proposed system maintains the security guarantees of Skydentity, while also allowing for VMs created by workload orchestrators to access resources across clouds. We build on top of the infrastructure introduced by Skydentity—at a high level (shown in Fig. 1.1), VMs created by the orchestrator can request credentials from authorizers in other clouds; these credentials can be used in any typical resource request in the cloud provider as usual. Each authorizer is responsible for validating requests to enforce least privilege, ensuring that each VM can only acquire credentials for resources that they are authorized for.

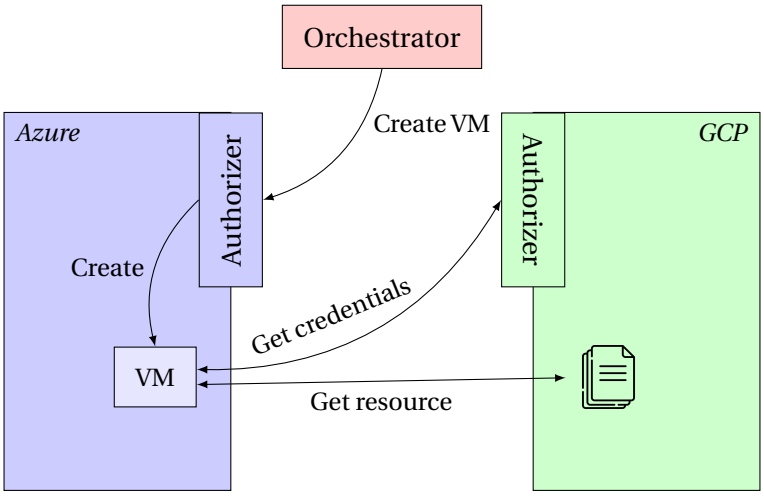

Figure 1.1: High level diagram of our system design

1.2 Related work

We'd like to highlight some related work in this section, discussing some of their shortcomings that we address in our extension of Skydentity.

1.2.1 Skydentity

Our solution builds off of existing work from the Skydentity project [22]. With Skydentity, developers deploy an *authorizer* in each cloud, and upload a *Skydentity policy* specification for each cloud. This policy specifies the resources that the orchestrator can access, create, or modify on the developer's behalf in each cloud.

Each third-party orchestrator deploys a *redirector*, which is a proxy that redirects all cloud requests to the appropriate authorizer. This ensures that the orchestrator application does not need to directly integrate with Skydentity; the redirector handles the necessary communication and request handling.

Under this setup, the orchestrator does not hold any cloud credentials. When the orchestrator is asked to provision resources in the cloud, it still makes cloud API requests as usual, but these requests will be intercepted by the redirector. The redirector then forwards the requests to the authorizer, which checks the request against the provided cloud policy. If the request is permitted by the policy, the authorizer attaches valid credentials and completes the request by forwarding it to the cloud. Otherwise, the request is rejected, and reports an error to the redirector, which in turn reports the error to the orchestrator.

The main issue with Skydentity's design that we intend to address is that cloud environments under Skydentity are completely isolated from each other; authorizers in one cloud never communicate with other authorizers in other clouds. This means that natively, there is no mechanism to support communicating credentials across clouds, and there is no policy specification for access control across clouds.

In this work, we will focus on the authorizer proxies, and extend their usage to allow for additional capabilities to intercept and forward requests made by newly provisioned VMs. In particular, we will omit specific mentions of the redirector proxy for the orchestrator service—we assume that the orchestrator makes requests through the redirector, and we omit the redirector from any sequence diagrams.

1.2.2 SPIFFE and SPIRE

SPIFFE [8] is a framework for identifying software systems in dynamic environments, and SPIRE [21] is an implementation of SPIFFE APIs for deployment in production environments.

The SPIFFE framework allows developers to identify and authenticate dynamic workloads in cloud environments, assigning SPIFFE IDs to each workload and generating SPIFFE Verifiable Identity Documents (SVIDs) for verifying identities. However, it is explicitly out of scope for SPIFFE to manage *authorization* of workloads [8, Chapter 4].

In the situation where VMs (i.e. workloads) are created by an untrusted workload orchestrator, there already needs to be a system in place to securely process and attach credentials to VM creation requests (ex. Skydentity). While SPIRE can be utilized to handle VM identity verification after the VMs are created, this still leaves a lot of work for the developer to integrate the process of validating SVIDs with each cloud provider, in order to authorize workloads for resource access across clouds.

In particular, it becomes difficult to deploy cross-cloud systems to handle resource requests made from VMs—each cloud provider has their own APIs and their own unique authorization systems, which adds to the burden of the developer to handle each of these cases.

Our goal with this work is to simplify this process of authorization; while it is possible to utilize SPIRE to handle identity verification, we show that it is also possible to utilize the existing

systems used in Skydentity to provide this functionality, while maintaining security with an untrusted workload orchestrator.

1.2.3 Identity Federation

Some cloud providers provide native methods of *identity federation*—a system that link workload identities across multiple identity management systems (i.e. across clouds). With identity federation, workloads in one cloud can be registered in the identity management system for another cloud, and gain access to resources through this identity translation layer.

However, an issue arises when workloads are created dynamically—most identity federation systems require the workload identity to be known and uniquely identified beforehand. For example, to grant permissions in GCP to a workload in Azure [10], the Azure workload needs to be registered with a Microsoft Entra ID application, and a managed identity must be created for the workload. In GCP, a *workload identity pool* needs to be set up, and IAM bindings need to be created to map permissions to unique identities. In order to set up these IAM permissions, the identity of the workload in Azure must already be known—this corresponds to the managed identity in Azure. In a dynamic workload environment, this managed identity will likely not be created until the workload is first submitted—this makes it impossible to assign IAM policies in GCP beforehand.

In the setting where workload orchestrators are untrusted, this becomes an even larger issue—the workload orchestrator should not hold any credentials to any of the user's cloud environments, so it cannot register any VMs with other clouds for identity federation in the first place. We discuss in Section 2.4.4 a potential method of integrating identity federation with Skydentity, including discussion on its benefits and drawbacks.

Chapter 2

Design

2.1 Threat Model

Our threat model mirrors that from Skydentity [22]. In particular, the workload orchestrator is untrusted: it never holds user credentials for any clouds. Through Skydentity, workload orchestrators can only communicate with the cloud provider through a user-specific and cloud-specific authorizer proxy, which interposes on these cloud requests to enforce explicit access control policies specified by the developer.

Authorizers are trusted, but we ensure that authorizers operate with least-privilege access. If an adversary compromises an authorizer in a particular cloud, we guarantee that the compromise is limited to only one user. However, one difference in security guarantee from native Skydentity is that here, with the addition of cross-cloud resource access, authorizers now have privilege in other clouds for the user as well. This means that the compromise of an authorizer in a particular cloud can lead to additional resource compromise in other clouds.

However, we make the security guarantee that cross-cloud resource compromise is limited to the scope of the user's explicit authorization policies—even if an authorizer for one cloud is compromised, the adversary can never access any resources in other clouds that are not explicitly included in the cross-cloud authorization policy for some orchestrator.

Since the compromise of a workload orchestrator (or an authorizer) inevitably leads to the ability of an adversary to impersonate the orchestrator (or the authorizer, respectively), denial of service attacks and resource wasting attacks (ex. creating more VMs than what is requested) are not in scope for our threat model. Further, we trust that each cloud provider appropriately enforces their own cloud authorization policies. (These assumptions are mirrored from Skydentity.)

2.2 Motivation

Our main approach is to utilize the authorizer proxies in Skydentity to request for credentials in other clouds.

A naive approach is to build directly on top of Skydentity. Here, the orchestrator creates a VM through the authorizer in the source cloud as usual, and the VM attempts to access resources in the destination cloud directly. A naive implementation fails here, since the VM does not have any credentials for the destination cloud. One solution is to simply provide these cloud credentials upon creation of the VM—the VM was created through an authorizer, which can pass along these additional credentials to the VM when it is created.

The main issue with this approach is the management of these cloud credentials among the authorizers. It is difficult to handle fine-grained access control with this model. We would like to enforce least-privilege access control, which means that ideally the newly provisioned VM should only have access to the resources that it needs.

This means that the authorizer must be in possession of multiple sets of credentials, for all the various permissions that are required for newly provisioned VMs. Further, if we'd like to support multiple different destination clouds (ex. if the VM in GCP requires access to both Azure and AWS), the amount of credentials that each authorizer needs to handle can increase drastically, and even worse, each set of credentials must be duplicated across all authorizers in all clouds.

A more sensible solution is to utilize the other existing cloud authorizers. In Skydentity, an authorizer is deployed to every cloud that may be in use—this means that access requests can be sent through the authorizer in the destination cloud instead, bypassing the need for every authorizer to store credentials for every cloud. A sequence diagram of this process is shown in Fig. 2.1.

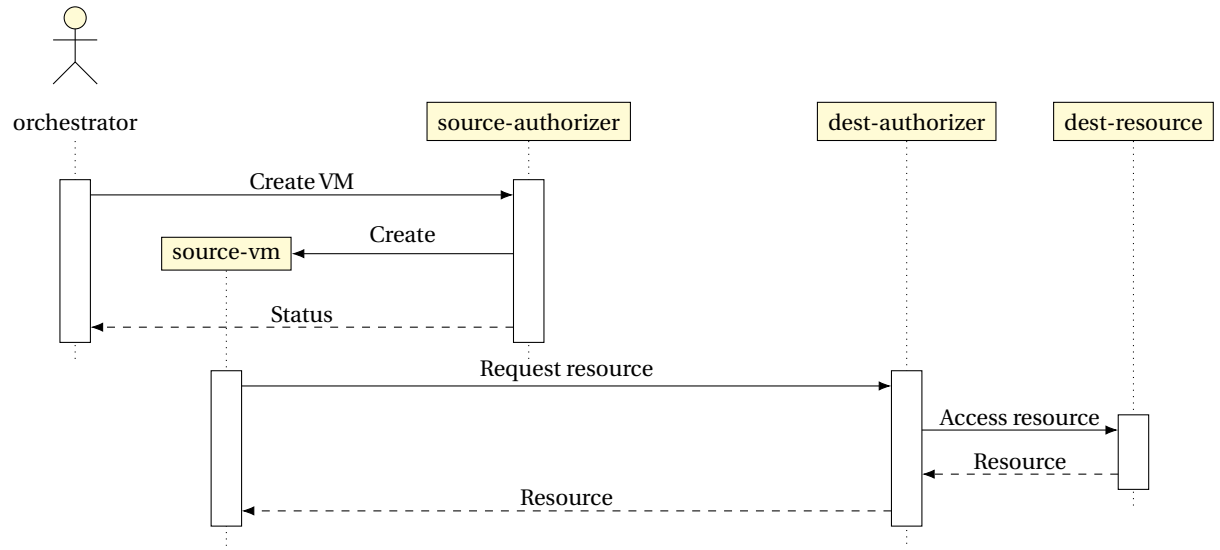

Figure 2.1: Sequence diagram for a VM in the source cloud attempting to access resources in the destination cloud, naively built on top of Skydentity

However, this solution also has its own set of issues, which will serve as the main issues that

we will focus on in our system design discussion that follows in Section 2.4.

1. **Policy specification**. With this new form of resource access, the existing policies in Skydentity do not cover the set of possible actions made by the newly provisioned VM (the existing policies only restrict actions that the *orchestrator* can perform)—we'd need to have another way of specifying an access control policy for VMs.
2. **Identity validation**. In Skydentity, all requests made to the authorizer must be signed with a valid signature corresponding to the orchestrator (this signature is added by the redirector, and the public key for this signature is usually registered with the authorizer during initial setup). This signature validates the identity of the sender, and ensures the integrity of the request during transit.

   Because of this, a newly created VM is unable to send requests to an authorizer: it cannot sign its requests, and it has not registered its public key with the authorizer. This means that the authorizer will deny all of its requests for resource access.

   If we wanted to send requests from a newly created VM to an existing authorizer, there would need to be a system in place to dynamically create key pairs for new VMs, register them with the authorizer, and persist them across interactions—Skydentity is unable to handle this case as-is.
3. **Latency**. We'd like to minimize overhead and latency as much as possible. This communication pattern requires the proxy to be on the critical path of all resource accesses—every request for a resource must pass through the authorizer, which can add a significant amount of additional latency to every request, especially if the authorizer has its own validations to perform before it can forward the request forward.

An immediate improvement that could be made regarding latency is to separate the request for *authorization* and the request for the *resource* itself. Instead of intercepting every request from the VM to add credentials, we can instead split up the process into two phases. In the first phase (usually as one of the first actions when a VM is created), the VM sends a request to the authorizer to request for *credentials*; these credentials are cached in the VM, and used to directly request for a resource from the cloud provider. This way, any latency involved in validating requests, checking permissions, fetching credentials, etc. is separated completely with the critical path of requesting resources. A sequence diagram for this design is outlined in Fig. 2.2.

We'll be focusing on this improved design (and variations on it) in the following discussion, where we request credentials before requesting for the resource. The next few sections discuss the details involved in the policy design (Section 2.3) and system design (Section 2.4).

2.3 Policy Design

One of the first issues that must be addressed is the policy specification. Skydentity policies only handle access control for the orchestrator; in order to impose access control on dynamically

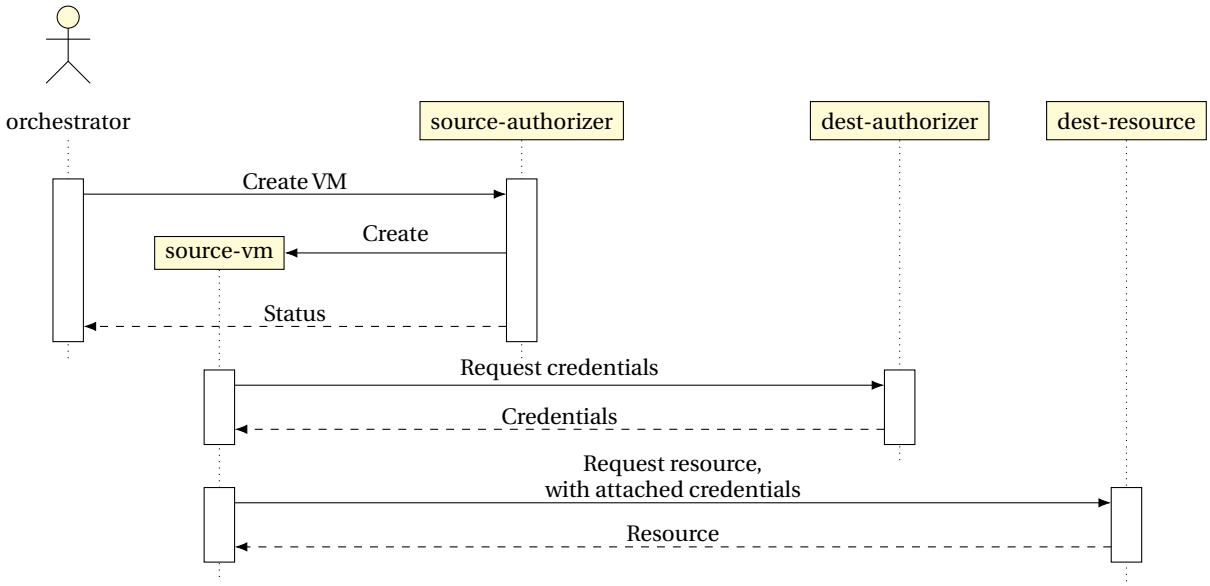

Figure 2.2: Sequence diagram for a VM in the source cloud attempting to access resources in the destination cloud, split into two phases, requesting credentials to be used in future requests

provisioned VMs, we need a new policy specification that captures the many possible actions that a worker VM can take.

The major hurdle here is that there is a large variety of workloads that an end user may want to support, and each workload can require a vastly different set of permissions and actions to be given to the worker VMs. Our policy design must be flexible enough to handle the variety of workloads (even across multiple cloud environments), while maintaining ease of use for the developer.

The most fine-grained policy specification associates each individual VM with an action it can take on a specific resource in a single target cloud. This gives the highest possible level of control over a VM's resource access, but it comes with some major drawbacks. The biggest issue is that this kind of policy requires knowledge of a VM's identity. Ideally, we'd want to create these policy documents *prior* to the creation of the VM, but at this point the VM's identity is not known, so it is impossible to define such a policy. Another issue is that this policy can be overly specific: oftentimes, developers want to group VMs together and give multiple workers the same set of permissions—it would be tedious to specify each VM's identical permission set individually.

In the other direction, the most coarse-grained policy specification associates the exact same set of permissions to every VM that an orchestrator creates. This universal policy is perhaps the easiest to understand and create, but it has a lot of immediate issues. In most workloads, different VMs would naturally have different roles, each requiring a different set of

permissions. To impose least-privilege access control on these VMs, we do not want to give every single VM the exact same permissions; each VM (or group of VMs) should have its own set of permissions.

In the ideal scenario, we'd like a policy specification somewhere in the middle of these two extremes. We want to avoid specifying policies for individual VMs (since we do not have the identity of VMs before they are created), but we also do not want to assign the same policy to every VM that an orchestrator creates.

A common design pattern in authorization systems is to group entities together, ex. by giving them all a common *role* [4, 11, 19]. We can take inspiration here and do something similar. Each VM can be assigned a role within the Skydentity ecosystem, and this role can be used to assign permissions to each VM through the policy document. This way, we can have as much granularity as we wish in assigning permissions, without needing to know a VM's identity beforehand.

Further, with the usage of roles to group VMs together, it is natural to include the ability to specify *hierarchical* relationships. For example, suppose a developer has a workload that consists of a controller VM and many worker VMs, where the controller is responsible for delegating tasks to each worker. Here, it may be the case that every VM requires some set of base-level permissions (ex. sending and receiving messages from a queue), while the controller requires additional permissions for administrative tasks (ex. creating and modifying worker VMs), and the workers require additional permissions for their own specific tasks (ex. reading/writing to a database or cloud storage).

Under this policy specification, the controllers can be assigned a `workload/controller` role, and the workers can be assigned a `workload/worker` role. Permissions can be specified at the base level under the `workload` role, which will be applied to both controller and worker VMs. Additional permissions can then be granted for the `workload/controller` and `workload/worker` roles individually, each of which already inherit permissions from the base `workload` role. This avoids the need to re-specify the same permissions for multiple groups of VMs.

2.3.1 VM roles

With the notion of VM roles, the next question is: how does a user specify these roles when the orchestrator creates VMs? It turns out that most cloud providers already allow for VMs to be associated with *tags* (also called *labels* in some clouds) [3, 13, 18]. We can utilize this to attach some extra metadata information to the VM upon creation, which can be read by the authorizer when the creation request is sent.

One consideration is whether we should allow a single VM to be associated with *multiple* roles. This may be useful in scenarios where there is a subset of common permissions among some VMs, but another intersecting subset of VMs may require a different subset of permissions. For example, developers may choose to separate permissions by resource—there could be individual roles for reading from/writing to cloud storage, querying databases, pushing/pulling

from queues, etc. Assigning multiple roles to a given VM can simplify the policy design process for the developer.

However, this causes in issue in implementation—in the end, we want to translate each of these roles specified in the policy into a set of cloud-specific permissions. These cloud-specific permissions generally must also be attached to an access identity (called *service accounts* or *managed identities* in some clouds), so that a user (or a VM in our case) can exercise these permissions [4, 11, 19].

These access identities must then be created prior to a VM attempting to access resources. However,the process of creating these identities can take a while, in order for the changes to propagate globally [4, 11, 19]. If we create these identities dynamically when receiving requests from the VM, latency becomes a huge issue—we'd need to wait for the access identity to be created and propagated, which can take time on the order of minutes.

The solution would be to create all the necessary identities upon setup, far before any resource requests are made. However, if we allow each VM to be associated with multiple roles, it becomes difficult to determine what identity to assume when requesting the resource. We cannot merge multiple roles into a single identity, since there is an exponential number of possible subsets of roles that can be assigned to VMs. Inferring the roles based on the request can also get complicated quickly, since native cloud-specific permissions can provide access to a large variety of resources.

For our proof of concept implementation, we decided to disallow assigning multiple roles to a single VM, for simplicity—this means that every VM can only be associated with a single role (corresponding to a single access identity). This assumption aligns with the behavior in existing cloud providers: generally, VMs are only allowed to be attached to a single identity [1, 14][1]. With more sophisticated heuristics or inference, it may be possible to perform static analysis on a policy document to create service accounts for a subset of the possible roles, or it may be possible to infer the correct role to assume given the request made from the VM.

2.3.2 Policy document format

Our discussion gives rise to the following policy document in Listing 2.1.

After uploading this policy via the authorizer proxy, the actual stored policy document will contain information about the newly created identity that has the specific permissions for each cloud, as shown in Listing 2.2.

Here, note that each cloud has their specific ways of managing users and policies; this policy document attempts to combine the common elements of identity management across multiple clouds into one flexible format.

All cloud providers have some mechanism to specify a permission for a given resource:

[1]It should be noted that Azure actually allows for multiple managed identities to be associated with a single resource [16, 17]; in particular, this is most commonly used so that a resource is attached to a *system-assigned* managed identity alongside a *user-assigned* managed identity.

```yaml
- role: bucket-reader
  clouds:
    - cloud: gcp
      permissions:
        - permission: roles/storage.objectViewer
          resource:
            type: bucket
            name: storage-bucket-name
        - role: ...
          resource: ...
    - cloud: aws
      permissions:
        - permission: arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess
        - ...
- role: ...
  clouds: ...
```

Listing 2.1: Example policy document prior to upload

- In GCP, the most common method is to assign a predefined *role* to a given GCP *scope* (ex. a specific resource, or the entire project, etc.) [11].
- In Azure, the most common method is to assign *roles* to a given Azure *scope* (ex. a specific resource, or the entire project, etc.) [19].
- In AWS, fine-grained access control is performed through custom *policies*, which associate *actions* with certain *resources* [4].

One notable difference here is that AWS handles policies slightly differently from GCP and Azure—AWS bundles together the permission with the resource, so a single AWS IAM policy already has restrictions on which resources the permissions apply to. Other clouds may have different authorization schemes, and thus the policy document in our application should be flexible enough to support these variations.

(As an aside, in the case of AWS, the support for fine-grained association of permissions to resources does allow us to actually *create* custom AWS IAM policies based on this policy document—this means that it is ultimately a design decision whether to support dynamic policy creation in AWS. In the ideal case, we would like to support as much customization as possible with our policy document format, while still maintaining readability for the end user.)

Regardless of the cloud though, there is always some form of *permission* given to an identity; some clouds (ex. AWS) just don't have an independent resource specification applied to the policy. This means that the `permissions` list will always include some kind of `permission` value, but everything else is cloud-specific.

A set of permissions is then given to a specific identity:

```yaml
- role: bucket-reader
  clouds:
    - cloud: gcp
      permissions:
        - permission: roles/storage.objectViewer
          resource:
            type: bucket
            name: storage-bucket-name
      # newly added
      access_identity: service-account@project-name.iam.gserviceaccount.com
    - cloud: aws
      permissions:
        - permission: arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess
      # newly added
      access_identity: arn:aws:iam::123456789012:role/sky-identity-custom-role
- role: ...
  clouds: ...
```

Listing 2.2: Example policy document after uploading, transformed to include newly created access identities

- In GCP, a *service account* is usually used for programmatic access; one can assume a service account to perform actions on its behalf [11].
- In Azure, *managed identities* are used for programmatic access; one can assume a managed identity to perform actions with the appropriate permissions [19].
- In AWS, *roles* are usually used for programmatic access; one can assume a role to perform actions with the appropriate permissions [4].

This identity is specified in the policy document through the `access_identity` key; this is added by the authorizer proxy after the respective identity has been created in the cloud.

2.4 System Architecture Design

We will now discuss a few potential architecture designs, looking at the strengths and weaknesses of each.

2.4.1 Routing requests through the destination authorizer

The initial design described in Section 2.2 has the VM request credentials from the destination authorizer. The full sequence diagram is shown in Fig. 2.3; we'll now discuss each aspect of this flow.

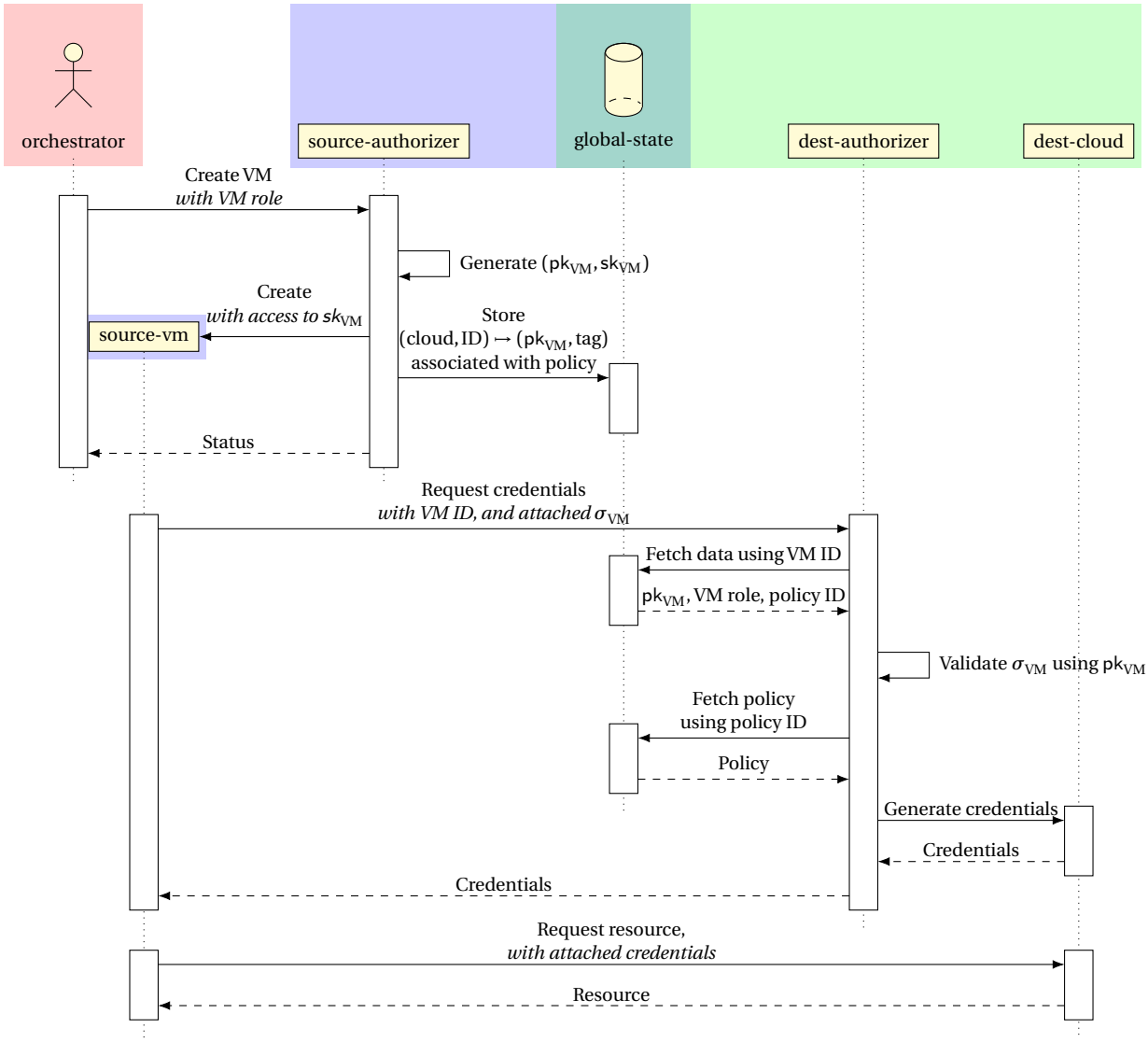

Figure 2.3: Detailed sequence diagram for VM creation and a subsequent credentials request utilizing custom signatures, where the VM contacts the target cloud authorizer. The colored regions indicate the entities present in each cloud; the global state is shared across clouds.

At the high level, when receiving a request to create a VM, the source authorizer must first log the VM's identity, so that future requests from the VM can be validated, and associated with the correct policy. Later on, after the VM is created, it sends a request to the destination authorizer for credentials in order to access resources. The destination authorizer proceeds to validate that the request actually came from the correct VM, and generates credentials for the corresponding role.

The biggest issue that is addressed here is that the destination authorizer must be able to validate the identity of the requester, to ensure that the request was made by a VM that was given permission through a policy, rather than another untrusted source. One way of solving this problem is by introducing custom signatures to requests made by the VM.

When the authorizer creates the VM, it generates a fresh pair of keys $(\mathsf{pk}_{\text{VM}}, \mathsf{sk}_{\text{VM}})$ for the VM. The VM will be given access to the secret key (ex. through additional permissions to access a secret store), and the public key will be stored in persistent state.

The persistent state is organized as a global key-value store. (Note that this state needs to be globally accessible: the source authorizer writes to this state, and the destination authorizer reads from this state.) The key will consist of the source cloud name, alongside the VM's resource ID. The VM's resource ID, assigned by the source cloud, is usually guaranteed to be globally unique *within* the cloud, so the source cloud name must be included in order for this ID to be globally unique across *all* clouds. This tuple is then used to map to any information needed for the VM, including the public key pk_{VM} and the VM's role.

This means that when the VM sends the request for credentials, it will send its own resource ID, the source cloud name, the target cloud name, and a timestamp, alongside a signature σ_{VM} over all of these fields, using its private key sk_{VM}.

The destination cloud uses the VM resource ID and source cloud name to fetch pk_{VM}, the VM's role, and the appropriate policy ID. The public key pk_{VM} is used to validate the attached signature σ_{VM}; this ensures that the request was sent by the desired VM, and not any other untrusted party.

The next step for the destination authorizer is to fetch the policy, and extract the correct access identity for the destination cloud corresponding to the VM's role. A final request is made to the destination cloud to generate fresh credentials for the access identity, and the credentials are returned to the VM. The VM caches these credentials for future resource requests; when the credentials expire, new ones can be fetched through this same process.

Security

MITM attacks on the requests Throughout this request flow, there is only one main point of attack on the requests: the request made from the VM to the destination authorizer. Here, we will walk through a few scenarios to show that the signature used for VM identities (i.e. σ_{VM}) is sufficient to guarantee the integrity of the request, guaranteeing that the request originated from the expected VM.

Recall that the request from the VM to the destination authorizer includes a signature σ_{VM} over the VM's resource ID, the source cloud name, the target cloud name, and a timestamp.

Let's focus on a man-in-the-middle (MITM) adversary, which can intercept messages between the VM and the target cloud. Here, the adversary will inevitably gain access to the credentials produced by the authorizer, and thus will be able to access the same resources that the VM can while the credentials are valid. However, we'll show that the adversary will not be able to gain access to any other credentials, nor will it be able to refresh its credentials after they expire.

The inclusion of the VM's resource ID and source cloud name ensures that the adversary is unable to request credentials while pretending to be another VM. This ensures that it will not be able to retrieve credentials for another role (associated with another VM), because modifying the ID or the source cloud will cause the signature check to fail.

The inclusion of the target cloud ensures that the adversary is unable to request credentials for access to a different target cloud; modifying the target cloud name will cause the signature check to fail.

The inclusion of the timestamp prevents replay attacks; the adversary cannot simply send the exact same request at a later time to retrieve fresh credentials, since the timestamp will not be valid, and the request will fail.

It should be noted that typically these requests will be sent through HTTPS, which prevents MITM attackers from decrypting request contents; the above analysis shows that even with requests sent under HTTP, the system is still relatively robust to MITM attackers.

Orchestrator compromise Another adversary in our threat model that we would like to ensure security against is an adversary that compromises the orchestrator. Here, the security of the system is automatically guaranteed, since the orchestrator holds no credentials to the user's cloud. This is a guarantee provided by the architecture of Skydentity, and it is preserved with this design. In particular, the adversary is unable to grant themselves additional access to the user's cloud, beyond the authorizations explicitly given to the orchestrator. Under Skydentity, this is generally only going to be permissions to *create* resources, but no permissions to *read* resources—this means that the adversary will only be able to execute resource wasting attacks, but no user data will be leaked.

Authorizer compromise Here, we also consider an adversary that compromises one of the authorizers. If the source authorizer is compromised, the adversary will inevitably gain access to resources within the source cloud, and it will also gain access to some resources in the target cloud. However, this design guarantees that this adversary will not gain access to any resource that is not part of the policy specification.

This adversary will be able to make requests on behalf of the source authorizer, so it will be able to create new VM key pairs, and it will be able to modify the global state. In particular, this means that the adversary can send valid credential requests, impersonating a new VM in the source cloud. However, due to the policy checks made by the destination authorizer—the destination authorizer ensures that the VM role is present in the policy specification, and fetches the appropriate access identity—this adversary will not be able to access any other resources that do not appear as part of the policy specification.

The only caveat is that we require each authorizer to only be granted *read-only* access to the global policy store. This aligns with the principle of least-privilege—read-only permissions to the global policy store are sufficient for the operation of the authorizers, since there is never any need to modify the policy documents. Only the end-user may have a need to create new policies or edit the policy documents—not any of the authorizers.

As such, under the assumption that the authorizers have no write access to the global policy store, any adversary that compromises one of the authorizers will not gain access to any resources in other clouds that are not part of a policy document.

2.4.2 Routing requests through the source authorizer

An alternative is for the VM to send credentials requests to the source authorizer; the source authorizer is then responsible for forwarding the request along to the appropriate cloud. The full sequence diagram is shown in Fig. 2.4; we'll now discuss each aspect of this flow.

The VM creation process is identical to the flow when directly contacting the destination authorizer: when the source authorizer receives a request to create the VM, it generates a new key pair, and stores this metadata information in state.

After the VM is created, it will then send a request to the *source authorizer* for access identity credentials; this is a request within the VM's cloud. The source authorizer checks the VM's identity, checks the policy, and forwards the credential request to the destination authorizer. The destination authorizer does not need to perform any policy checking, and is only responsible for generating the credentials.

This design allows for the resource requests to be more isolated within each cloud; the VM is now only responsible for knowing the address of a single source authorizer, rather than the address of *all* other destination authorizers. The cross-cloud requests are all abstracted away in the intermediate jumps between the authorizers.

As a consequence of this isolation, we also no longer need a global state—each authorizer only needs to know about the VMs that reside within their own cloud. This means that information about the VM's ID, public key ($\mathsf{pk}_{\mathrm{VM}}$), role, etc. only need to be stored in a cloud-local state.

Policy documents can also be stored in cloud-local state, despite the fact that the same policy needs to be read by multiple authorizers across clouds. This is because the policy document is almost always unchanging; it is updated once upon setting up the system, and very rarely updated afterward. The policy documents can thus be uploaded individually to each cloud state when setting up or when updating. These updates also generally will not coincide with any policy reads, since generally no VMs will be active when setting up permissions—the permissions setup is done beforehand.

Another big difference in this design is that now we have an additional hop for communication; there is additional request made between the source authorizer and the destination authorizer. This means that we need additional infrastructure setup to ensure the integrity of this request.

The request between the VM and the source authorizer is almost identical to the request between the VM and the destination authorizer. When the VM makes a request for credentials, it includes its own resource ID, the target cloud name, and a timestamp, alongside a signature σ_{VM} over all of these fields, using its private key $\mathsf{sk}_{\mathrm{VM}}$. Notably here, we no longer need to include the source cloud name in these fields, since the request is always internal to the VM's cloud.

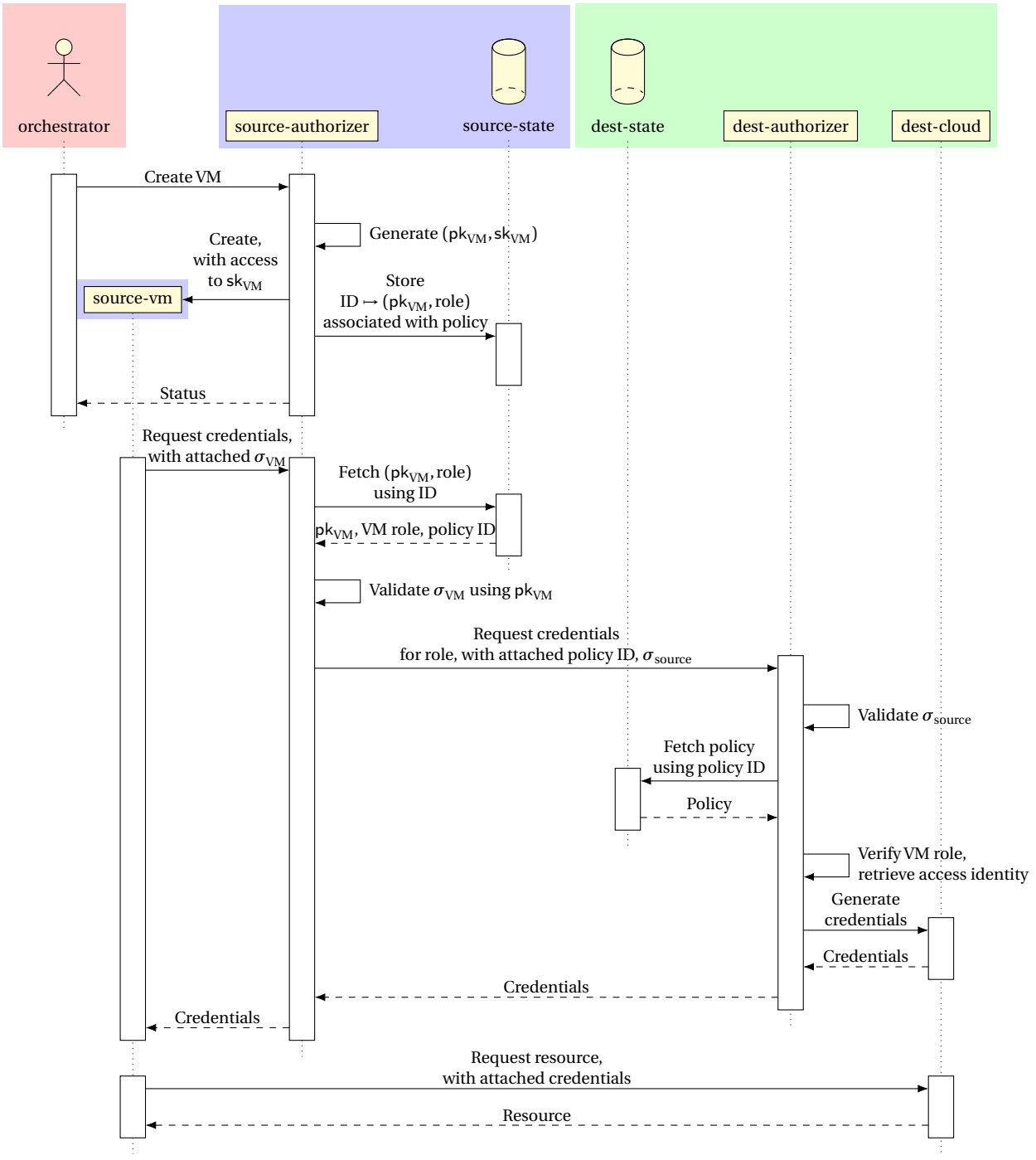

Figure 2.4: Detailed sequence diagram for VM creation and a subsequent credentials request utilizing custom signatures, where the VM contacts the source cloud authorizer. The colored regions indicate the entities present in each cloud.

The source authorizer then validates this request, fetching the VM's metadata (i.e. its public key pk_{VM}, the VM's intended role, and the policy ID) from the cloud-local state and using pk_{VM} to validate the signature. If the request is valid, the source authorizer makes a request to the destination authorizer to generate credentials, sending along the VM's role, the source cloud name, the target cloud name, the policy ID, and a signature σ_{source} over all of these fields, using its private key $\mathsf{sk}_{\text{source}}$.

When the destination authorizer receives this request, it must first validate the request; here, the destination authorizer must have access to the public key $\mathsf{pk}_{\text{source}}$ of the source authorizer. This can be communicated across authorizers during setup, and stored local to each cloud. In addition to ensuring that the signature is valid, the destination authorizer should also fetch the policy and check whether the role is valid. If the request is valid, the destination authorizer proceeds to generate credentials for the VM role, and returns it to the source authorizer, which then forwards it to the VM.

Security

MITM attacks on the requests Throughout this request flow, there are two main points of attack on the requests: the request made from the VM to the source authorizer, and the request made from the source authorizer to the destination authorizer. Here, we will show that the signatures used guarantee the integrity of the respective requests.

Recall that the request from the VM to the source authorizer includes a signature σ_{VM} over the VM's resource ID, the target cloud name, and a timestamp. We'll focus on an MITM adversary, which can intercept messages between the VM and the source authorizer. Here, the adversary will inevitably gain access to the credentials produced by the authorizer at the end of the request flow, but we'll show that the adversary will not be able to gain access to any other credentials, nor will it be able to refresh its credentials after they expire.

The security follows identically to the design discussed in the prior section (Section 2.4.1). In particular, the inclusion of the VM's resource ID ensures that the adversary is unable to impersonate another VM; the inclusion of the target cloud ensures that the adversary is unable to request credentials for a different cloud; the inclusion of the timestamp prevents replay attacks.

Notably here, we no longer need to include the source cloud—the same request will not be valid when sent to other authorizers in other clouds. In particular, it will almost always be the case that the VM ID is not valid in another cloud, and in the rare circumstance where it is valid (and is present in the local state), the corresponding public key will not match the signature.

Orchestrator compromise Similar to the other design, the orchestrator still does not hold any credentials, so compromising the orchestrator leaks no information.

Authorizer compromise If the source cloud authorizer is compromised, resources in the source cloud will also be compromised, and the adversary will gain access to some resources

in the target cloud. Here, we similarly assert that the adversary still is unable to gain access to any resource that is not part of the policy specification.

The adversary will be able to make requests on behalf of the source authorizer, so it can send valid requests to the destination authorizer. However, since the destination authorizer checks the policy for the requested role, the adversary can only request credentials for roles specified in a policy. (If the destination authorizer does not perform this check, then all cloud resources in the destination cloud would be compromised; the adversary can request credentials for *any* identity.)

2.4.3 Integrating with identity token services

Many cloud providers also provide their own form of VM identity verification; this can be used to improve the security of the scheme.

To utilize cloud-provided identity tokens, the only change required is that the VM must now request an identity token from the native cloud metadata service, and attach it to its request. As a result, the requests no longer need to have any unique identifying information about the VM (i.e. the VM ID and cloud no longer need to be included). The destination authorizer (or source authorizer, depending on whether this extends the design in Section 2.4.1 or Section 2.4.2) is responsible for validating the identity token as part of the request validation process.

However, one big caveat is that identity tokens typically only attest to the identity of the VM, it cannot attest to any additional custom information. As such, we would still need to include our own custom signatures to ensure the integrity of the request. This means that the request made by the VM now includes the identity token, the target cloud name, and a timestamp. The custom signature σ_{VM} attached to this request is created over the target cloud name and the timestamp, signed as usual with sk_{VM}.

One benefit of using existing identity token services is that we delegate part of the security to the cloud provider. We trust that the cloud provider has a secure implementation of identity verification through their identity tokens, which allows us to avoid re-inventing the wheel. However, at the same time, different cloud providers may have different APIs and guarantees for identity verification, and cloud providers may change their APIs at any point in time. This makes the integration process more challenging, and may require more maintenance effort.

2.4.4 Integrating with identity federation

As mentioned in Section 1.2.3, many cloud providers provide identity federation as a system of linking workload identities across multiple clouds. In this section, we discuss one potential method of utilizing existing identity federation services with Skydentity, in the setting where workload orchestrators are untrusted, and hold no user credentials.

Every cloud has its own unique identity federation system, so there is no single request flow across all identity federation systems. The overall outline is similar though: when a VM is created, the source authorizer registers the VM with the identity federation service in the destination

cloud; when a VM attempts to access a resource, it first requests a short-lived authentication token directly from the destination cloud, and uses the token to access resources.

One example of this flow is shown in Fig. 2.5, where a VM in GCP accesses resources in AWS [2]. Here, a policy is set up within AWS IAM to allow GCP VM principals (validated through the use of JWT tokens) to request short-lived authentication tokens for AWS access. In particular, when the VM is first created, it requests a JWT token and forwards it to the GCP authorizer, so that the authorizer can use the token to set up the identity federation policies in AWS. Once the policies are set up, the VM can then request an STS token from AWS, using its JWT token as a proof of its identity. The STS token is used for subsequent resource requests in AWS.

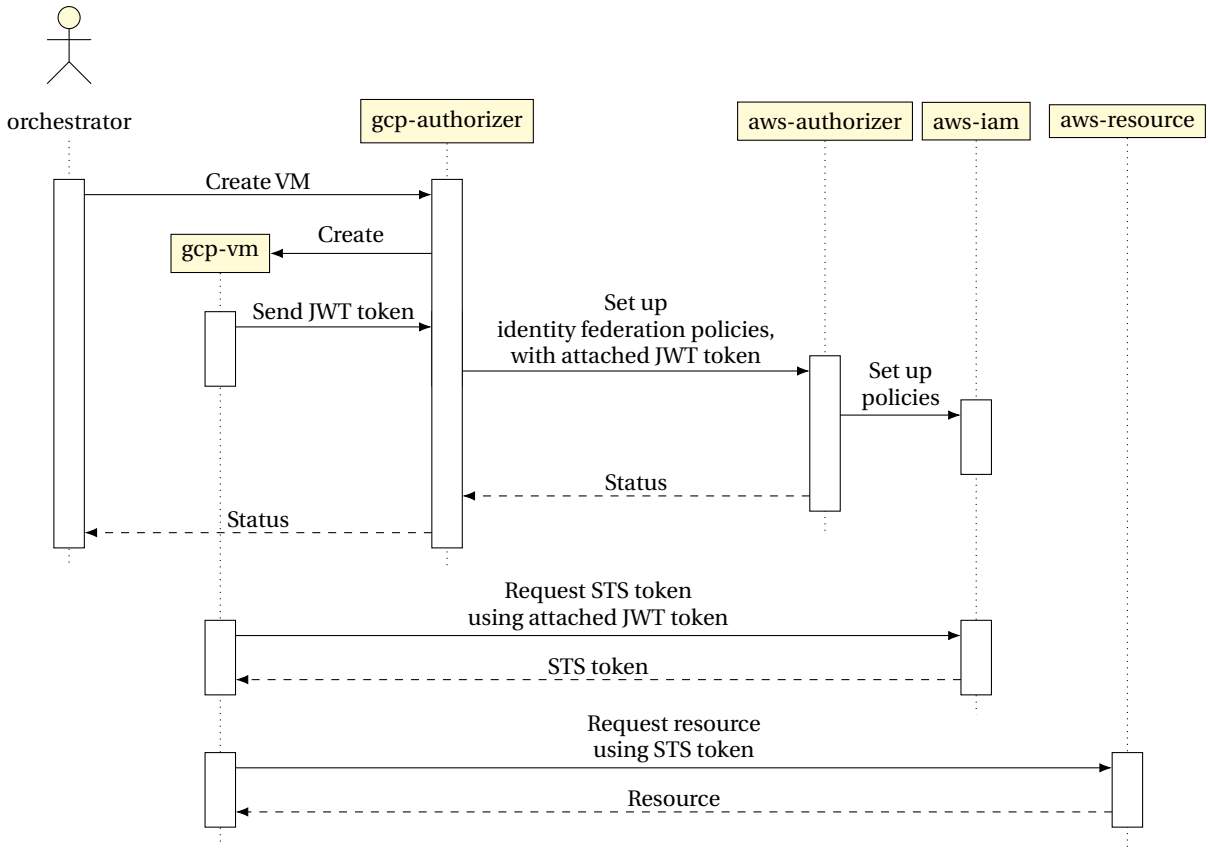

Figure 2.5: Detailed sequence diagram for VM creation and subsequent resource request, utilizing existing identity federation in AWS

In GCP [10], the flow is slightly different, since the GCP client library handles a lot more of the communication; Fig. 2.6 details the sequence diagram for a VM in Azure accessing resources in GCP. Here, when the VM is created in Azure, the Azure authorizer sends a request to the GCP authorizer to set up the identity federation policies, passing along the newly created VM's object ID. This allows the GCP authorizer to create a workload identity pool, associating the

VM (using its ID) with a particular access control policy. The result of this setup is a *credential configuration file*, which is sent back to the Azure VM. Note that this credential configuration file is a public document—it does not contain any private information [9, 10].

When the Azure VM attempts to access resources in GCP, the GCP client library automatically performs an initial credentials request—this request contains an identity token (much like JWT tokens from GCP VMs) that verifies the VM's identity. The response from GCP is a short-lived authentication token, which is then used for subsequent resource access in GCP.

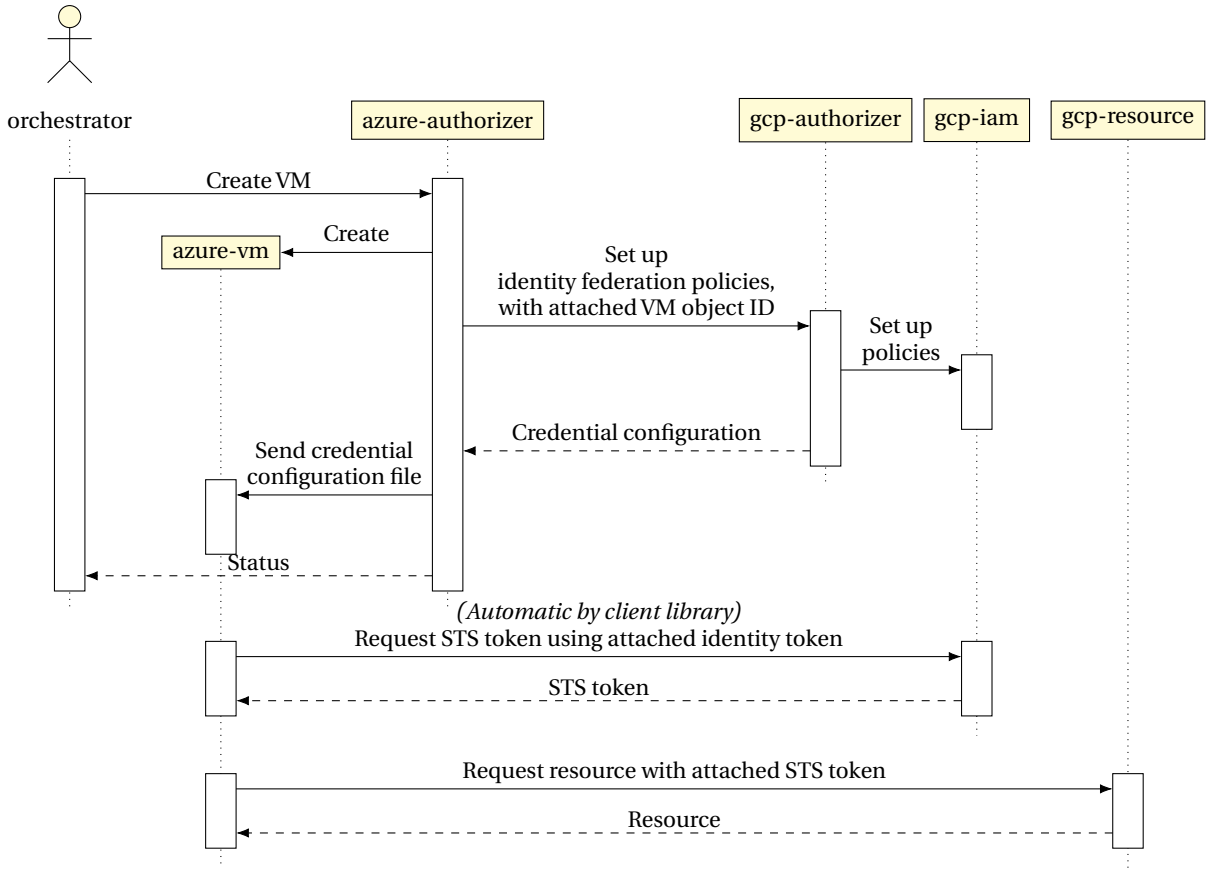

Figure 2.6: Detailed sequence diagram for VM creation and subsequent resource request, utilizing existing identity federation in GCP

Many cloud providers already implement some form of identity federation—this is the main benefit of utilizing it for cross-cloud resource access. Additionally, if we are able to utilize existing identity federation solutions, we do not need any custom security schemes; we rely on the functionality of the existing systems.

The major drawback is the lack of standardization across clouds. Even among two large cloud providers like GCP and AWS, their identity federation systems require slightly different setup steps, and slightly different access patterns. Further, identity federation would look

slightly different for different *pairs* of clouds as well; even if we fix GCP as the destination cloud, the setup looks slightly different for VMs originating in Azure compared to VMs originating in AWS, due to their unique identity verification processes.

Another big drawback is that identity federation systems can have additional limitations, and APIs can have a more limited and/or restricted scope when using identity federation [12]. As such, the end user may be completely blocked from using some aspects of cloud APIs if this design is used.

2.5 Final design

Our final design is the one described in Section 2.4.1, building on top of Skydentity to route requests through the destination authorizer. We'll now discuss some of the comparisons across the designs in the last section.

Across all of the designs discussed in the last section (Section 2.4), each has their own benefits and drawbacks. However, the two designs we'll mainly be considering here are the designs routing requests through the destination authorizer (Section 2.4.1) and routing requests through the source authorizer (Section 2.4.2). This is mainly because these designs offer the best balance of ease of integration and flexibility. Identity federation is completely cloud-specific, which makes it very difficult to integrate across multiple clouds (and it's not guaranteed to be available), while the use of JWT tokens adds an extra dependency on the cloud-specific implementations of VM identity verification (which also is not guaranteed to be available).

On the other hand, using our own custom signatures to facilitate communication and credentials transfer allows us to have a consistent interface across clouds, and further allows us to customize and consolidate access control policies across various clouds.

Routing requests through the destination authorizer and routing requests through the source authorizer are fairly equivalent; they both follow a similar high-level flow, while providing the same security guarantees. The main difference is in the latency: contacting the destination authorizer directly requires only two hops to get to the destination cloud, while contacting the source authorizer requires three hops. In the perspective of the VM, this added latency is the only observed difference between the two strategies; the process of making the credentials request can be completely abstracted away from the user scripts on the VM through a custom client library.

In the perspective of the developer implementing this Skydentity system, the two methods are slightly different, with each having their own benefits. Routing requests through the source authorizer requires the least amount of configuration information to be transferred to the VM—the VM only needs to know how to contact one authorizer in its own cloud. On the other hand, in order to route requests through the destination authorizer, the VM would need to know the address of *every* other authorizer (though it should be noted that this is still not much information to store).

Additionally, routing requests through the source authorizer allows us to use only cloud-local state; there is no longer any need for global state storage. This reduces the need for

extensive database scaling and availability at a global level, which can reduce costs of deployment.

However, since our main focus is on the end user, our final design routes requests through the destination authorizer—we'd like to prioritize reducing the latency of requests from the VM.

2.5.1 Policy uploading

With the final design settled, one piece left to discuss is the process of uploading policies and setting them up on each cloud. This process is typically done only once, by the end user, at the very beginning of the system setup. Since this is done by the user, we do not need any special handling of credentials—the end user holds all of their own cloud credentials.

This means that this process is relatively straightforward—the process is depicted in Fig. 2.7.

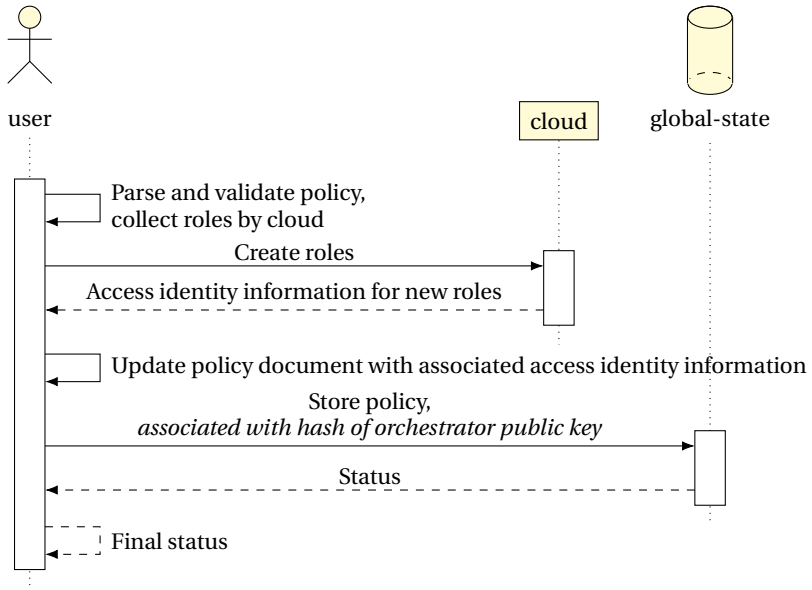

Figure 2.7: Detailed flow for policy upload and role creation. Note that if the policy allows for resource access in other clouds, then additional requests will be sent to those respective clouds to create additional roles.

In particular, when the end user wants to upload a policy, it runs a script that first parses and validates the policy, aggregating roles by destination cloud. A request is then made directly to the destination cloud to create the appropriate roles and access identities, and the corresponding information about the access identities are collected to update the policy. This is repeated for every cloud present in the policy document.

As mentioned in Section 2.4.1, the authorizers in each cloud should only have read-only access to the database storing these policies—this means that it is important for the policy

documents to be stored separately from other global state information for VMs. As such, after all of the roles are created, the final policy document (with all of the access identity information included) is then uploaded to this separate global file store, associated with the hash of the orchestrator's public key. This hash links the policy with the orchestrator it applies to, allowing the authorizers to find the appropriate policy for requests it receives.

2.5.2 State cleanup

A final consideration is the clean-up of the stored VM metadata information (this applies to both designs discussed in Sections 2.4.1 and 2.4.2). As-is, our design stores information about every VM that is created through Skydentity, but does not handle the deletion of this information. This means that when VMs are stopped or deleted, the metadata information will persist forever.

Deleting this metadata from the database is also nontrivial—we do not always know when VMs are deleted. One approach is to intercept all deletion requests made by the workload orchestrator, and remove the corresponding metadata information from the state. However, VM deletions may not always come from the orchestrator—the user may choose to delete the VM manually themselves. This means that it is impossible to intercept *every* possible VM deletion request.

A solution to this problem is to schedule the state clean-up to run occasionally at a fixed time interval. This state clean-up will iterate through all of the VMs currently present in the shared state, and checks to see whether the VM is still active and running. If the VM was deleted, the corresponding metadata information will also be deleted from the state.

In addition to this scheduled clean-up job, it is generally beneficial to also intercept any deletion requests from the orchestrator, so that we can delete metadata information whenever we can guarantee that the VM has been deleted. The clean-up job will cover any other edge cases, where the VM is deleted through other means.

Chapter 3

Evaluation

In this section, we evaluate our implementation of our extensions to Skydentity, assessing the development cost and added latency overheads incurred by integrating with our system.

3.1 Setup

Our implementation of the final design is built off of the prototype Skydentity implementation. In particular, all of the proxies are written in Python, with Flask as the web server. Evaluation results were collected with a VM created in Azure, requesting for resources in GCP. We simulate a workload orchestrator by sending a VM creation request to the Azure authorizer using the Azure client libraries, with no credentials attached.

Policies are stored in NoSQL databases (Firestore for GCP, and Cosmos DB for Azure), with the global state in GCP (used to store cross-cloud policies and VM metadata). Authorizer proxies are deployed as serverless functions. In GCP, the authorizer is deployed using Google Cloud Run, allocated with 512 MiB and 1 vCPU. In Azure, the authorizer is deployed using Azure Container Apps, allocated with 2 GB of memory and 1 vCPU.

There are also a few nuances when implementing the final design, which we will briefly discuss next.

When handling VM creation requests from the orchestrator, the Azure authorizer needs to communicate configuration information with the newly created VM (the private key associated with the VM, the addresses of other authorizer proxies, etc.). The most generic and straightforward way of communicating this information to the VM is through cloud-init; in particular, for our prototype, we make the assumption that the user initializes the VM using cloud-init, and we modify the cloud-init script to write additional configuration files to the VM upon creation.

Further, in an ideal scenario, we would package any Skydentity API interfaces into a standalone library, which can be installed independently on the VM. Since we only have an unpublished prototype implementation, we also copy over any necessary code to handle the interfacing with Skydentity upon VM creation through cloud-init as well. In practice, a lot of

these setup scripts can be abstracted away as a standalone library, reducing the amount of modifications necessary to the orchestrator's VM creation request.

Another point to consider is that serverless functions suffer from *cold starts*. To conserve on resources, compute instances are deallocated after a period of inactivity (in GCP, instances are kept alive for 15 minutes, while in Azure, instances are kept alive for 30 minutes)—if a request is sent after the instance is deallocated, additional time must be spent reinitializing a new instance. Cold starts add around 5 s to 30 s to the request latency. Since cold starts are generally rare in practice, all requests in our evaluation are made with a warm start.

In our implementation, we also include a few optimizations to reduce latency in practice. Since policies generally do not change very much, all policies are cached at each proxy for a short time; this means that for the vast majority of requests, there is no additional latency for fetching policies from the global state (without this caching, the time spent fetching a given policy can range from 200 ms to 500 ms). Because of this, all benchmarks in our evaluation are performed under the condition that the policies have already been cached in the authorizers.

3.2 Development overhead

One of the main goals of our design is to minimize the effort needed to integrate this security system. In particular, there are three parties involved here: the end user, who is trying to deploy their workloads in a multi-cloud environment; the orchestrator, who is providing the deployment service to the developer; and the cloud provider, who is hosting the resources deployed in their cloud.

There are no modifications required to any of the existing cloud infrastructure in order to utilize our system—we build on top of the existing REST frameworks provided by cloud providers. There is also very little development overhead for workload orchestrators: Skydentity utilizes redirector proxies to intercept all cloud requests made by the orchestrator, so the orchestrator only needs to deploy the redirector proxy in their (untrusted) cloud environment.

The end user has more overhead when integrating with this system. In particular, to set up Skydentity, they must create and deploy redirector and authorizer proxies in each cloud, and they must also setup and upload policy documents for the workloads that they are planning to run. To also integrate with our additions to Skydentity, the end user must also ensure that each VM is tagged with the correct roles, recognizable by the Skydentity authorizer proxies for validation. (For our proof-of-concept implementation, we look for a tag associated with a predefined key, and match the tag value with the policy document.)

It should be noted that most of the deployment and setup process for the end user can be automated through scripts—in a more polished implementation, these scripts can be prepackaged and run by the end user for setup.

3.3 Time breakdown

Table 3.1 breaks down the time spent in the Azure authorizer proxy—this consists of handling the incoming VM creation request from the workload orchestrator. Compared to the baseline of the native Skydentity implementation, we can see that our implementation adds approximately 836 ms of additional latency.

Here, we can see that most of the time is spent forwarding the VM creation request over to Azure; this latency already exists as part of the baseline Skydentity implementation. The next largest chunk of time is spent generating a new 2048-bit RSA keypair for the VM—in practice, this time spent generating the new keypair has a high variance, ranging from 100 ms to 900 ms. The third highest chunk of time is spent saving the VM metadata to the global state (i.e. the VM ID, the VM's requested label, and the hash of the orchestrator's public key to identify the corresponding policy). All other categories comprise the remaining fraction of time in the authorizer—request verification and cross-cloud policy checks take negligible time due to policy caching.

Category	Time (ms)	Percent time
Verify request	*3.48*	*0.18*
Check cross-cloud policy	4.54	0.24
Generate keypair	455.83	23.93
Modify cloud-init	44.41	2.33
Send Azure request	*1065.71*	*55.94*
Save VM metadata	331.04	17.38

Table 3.1: Breakdown of time spent in the Azure authorizer proxy. Items in *gray italics* already exist from the base Skydentity implementation; other items are newly added as a result of this work.

Table 3.2 breaks down the time spent in the VM during setup—this consists of the request for GCP credentials, to be used in resource requests later on. Here, the largest chunk of time is spent waiting for the request to the GCP authorizer for the actual credentials (we break down this time next). The next highest chunk of time is spent fetching and reading the configuration written to the VM's disk upon initialization, due to the disk I/O required. In order to fetch the VM's ID, a request is sent to the internal Azure metadata service—since this is a request local to the VM's virtual network, the request does not take much time.

Table 3.3 breaks down the time spent in the GCP authorizer proxy—this consists of handling the request from the VM for resource credentials, along with the actual generation of the new cloud credentials. Generating credentials for the GCP service account takes the longest time here, closely followed by the request for the VM's metadata from the global state. Policy checks take negligible time due to policy caching.

Category	Time (ms)	Percent time
Read config	74.46	23.97
Fetch VM ID	18.19	5.86
Create signature	2.69	0.87
Request credentials	215.32	69.31

Table 3.2: Breakdown of time spent in the VM for setup

Category	Time (ms)	Percent time
Fetch VM metadata	50.07	40.00
Validate signature	0.98	0.78
Check cross-cloud policy	0.05	0.04
Generate credentials	74.07	59.18

Table 3.3: Breakdown of time spent in the GCP authorizer proxy

Considering these benchmarks together, our implementation has a negligible effect on the latency of requests in the critical path of workloads—the added latency is usually a one-time cost.

In the Azure authorizer, we incur an additional latency of about 1.9 s in total, of which approximately 836 ms is added as a result of this work. This cost is only incurred upon VM creation, which usually only occurs once during the lifetime of a workload. Further, the VM creation process usually takes on the order of minutes (generally ranging from 30 s to 2 min), so this added latency is still very small in comparison (about 2.7% of the VM creation time in the worst case).

During the VM's initial setup, we incur an additional latency of about 310 ms. Similarly, this cost is also incurred rarely in the lifetime of the workload—GCP credentials have a default lifetime of 1 hour (though it can be extended to 12 hours), so we only need to re-request credentials every hour. All other resource requests made by the VM do not incur any additional latency cost, since we already have credentials for resource access in GCP.

Chapter 4

Concluding remarks

4.1 Limitations and future work

Our final proposed design has a few limitations that can be explored in future work.

One of the major limitations of our design is in the policy and role specification for VMs. As mentioned in Section 2.3.1, we impose a restriction where each VM can only be associated with a single role at any given point in time. This simplifies the creation of access identities and the request for credentials greatly: access identities can be created far in advance (when policies are first uploaded), and requests for credentials are also straightforward, since there is only one possible access identity to retrieve credentials for.

In future work, it is worthwhile to look into ways of allowing VMs to be associated with multiple roles. This can create a few issues though; if we allow each VM to be associated with any number of roles, then it quickly becomes infeasible to create access identities entirely in advance—there is an exponential number of possible subsets of roles any given VM can be associated with. Requesting credentials also becomes trickier: it can be difficult to determine which role the VM should assume in order to fulfill a resource request.

It may be possible to design a static analysis algorithm that looks at the permissions that can be given to VMs, and rearranges the user-defined roles into a hierarchical tree, so that each VM can be assigned a unique role that encapsulates all of its designated authorizations.

Another limitation of our design is in its implementation; in order to pass configuration information from the authorizer proxies to the newly created VM, we make the assumption that the user utilizes cloud-init. This allows us to make modifications to the user-provided cloud-init configuration to include additional data. This may cause incompatibilities with existing workloads, especially if the end user is not using cloud-init in the first place, or if the cloud provider does not support cloud-init. Future work into improving the implementation to support a more general way of communicating configuration information to each VM can resolve these potential incompatibilities, reducing the barrier to adoption.

4.2 Conclusion

We introduce a novel extension to Skydentity, solving the problem of enforcing least-privilege cross-cloud resource access for VMs created under Skydentity.

Through our extension of Skydentity, we provide guarantees on the security of the end user's cloud environments, even if the orchestrator is compromised. Further, if an authorizer is compromised, we guarantee that the scope of the compromise is limited to resources and actions that are explicitly included in the user-provided authorization policies.

Through our evaluation, we show that the system incurs a small amount of additional latency compared to native Skydentity, in addition to the fact that all added latency occurs off of the critical path of resource access. In particular, individual VM resource requests are unaffected by our additions.

Bibliography

[1] Amazon Web Services, Inc. *IAM Roles for Amazon EC2 - Amazon Elastic Compute Cloud*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html. 2025.

[2] Amazon Web Services, Inc. *Identity Providers and Federation - AWS Identity and Access Management*. https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_providers.html.

[3] Amazon Web Services, Inc. *Tag Your Amazon EC2 Resources - Amazon Elastic Compute Cloud*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Using_Tags.html. 2025.

[4] Amazon Web Services, Inc. *What Is IAM? - AWS Identity and Access Management*. https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html. 2025.

[5] Astran. *Astran*. https://www.astran.io/. 2025.

[6] Astronomer. *Astronomer: The Best Place to Run Apache Airflow®*. https://www.astronomer.io/. 2025.

[7] Bodo, Inc. *Bodo Developer Documentation*. https://docs.bodo.ai/latest/. 2025.

[8] Daniel Feldman, Emily Fox, Evan Gilman, Ian Haken, Frederick Kautz, Umair Khan, Max Lambrecht, Brandon Lum, Agustín Martínez Fayó, Eli Nesterov, Andres Vega, and Michael Wardrop. *Solving the Bottom Turtle — a SPIFFE Way to Establish Trust in Your Infrastructure via Universal Identity*. 1st Edition. 2020. ISBN: 978-0-578-77737-5.

[9] Google LLC. *AIP-4117: External Account Credentials (Workload Identity Federation)*. https://google.aip.dev/auth/4117.

[10] Google LLC. *Configure Workload Identity Federation with AWS or Azure | IAM Documentation*. https://cloud.google.com/iam/docs/workload-identity-federation-with-other-clouds. 2025.

[11] Google LLC. *IAM Overview | IAM Documentation*. https://cloud.google.com/iam/docs/overview. Documentation. 2025.

[12] Google LLC. *Identity Federation: Products and Limitations | IAM Documentation*. https://cloud.google.com/iam/docs/federated-identity-supported-services. 2025.

[13] Google LLC. *Manage Tags for Resources | Compute Engine Documentation*. https://cloud.google.com/compute/docs/tag-resources. 2025.

[14] Google LLC. *Service Accounts | Compute Engine Documentation*. https://cloud.google.com/compute/docs/access/service-accounts. 2025.

[15] HashiCorp, Inc. *Terraform*. https://developer.hashicorp.com/terraform. 2025.

[16] Microsoft Corporation. *Best Practice Recommendations for Managed System Identities - Managed Identities for Azure Resources*. https://learn.microsoft.com/en-us/entra/identity/managed-identities-azure-resources/managed-identity-best-practice-recommendations. 2025.

[17] Microsoft Corporation. *Managed Identities for Azure Resources*. https://learn.microsoft.com/en-us/entra/identity/managed-identities-azure-resources/overview. 2025.

[18] Microsoft Corporation. *Use Tags to Organize Your Azure Resources and Management Hierarchy - Azure Resource Manager*. https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/tag-resources. 2025.

[19] Microsoft Corporation. *What Is Azure Role-Based Access Control (Azure RBAC)?* https://learn.microsoft.com/en-us/azure/role-based-access-control/overview. 2024.

[20] Serverless, Inc. *Serverless: Zero-Friction Serverless Apps On AWS Lambda & Beyond*. https://www.serverless.com/. 2025.

[21] *SPIRE Concepts*. https://spiffe.io/docs/latest/spire-about/spire-concepts/.

[22] Samyu Yagati, Alec Li, Karthik Dharmarajan, Romil Bhardwaj, Sam Kumar, Raluca Popa, Malte Schwarzkopf, and Ion Stoica. "Enforcing Least-Privilege for Cloud Orchestrators". Work in preparation.

[23] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. "SkyPilot: An Intercloud Broker for Sky Computing". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 437–455. ISBN: 978-1-939133-33-5.