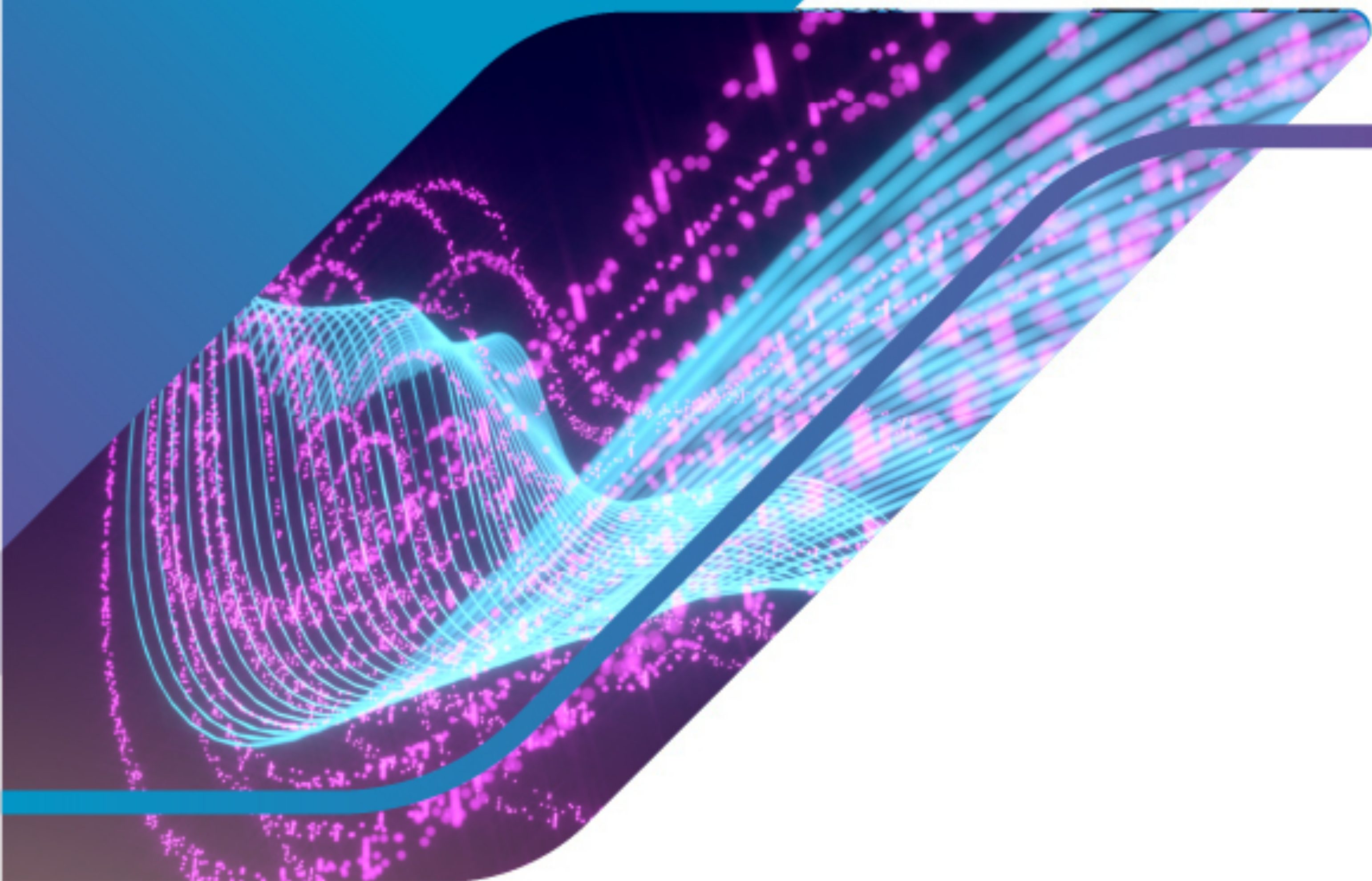
PERFORMANCE TECHNICAL WHITE PAPER
January 2025



VMware vCenter 8.0 U3 Tagging

Performance Best Practices

vmware by Broadcom

Table of contents

1 Executive summary 4
2 Introduction 4
3 Terminology and scope of paper 5
4 Tagging APIs 5
4.1 Tag and category creation APIs 6
4.2 Tag association APIs 7
4.2.1 Single tag to single VM 8
4.2.2 Single tag to multiple VMs 9
4.2.3 Multiple tags to a single VM 9
4.2.4 Multiple tags to multiple VMs 11
4.2.4.1 10 tags associated with 1,000 VMs (10,000 tag associations) 11
4.2.4.2 15 tags associated with 5,000 VMs (75,000 tag associations) 12
4.2.4.2.1 attachTagToMultipleObjects() 13
4.2.4.2.2 attachMultipleTagsToObject() 14
4.3 Associate tags with other objects 16
4.4 Query VMs associated with tags 16
5 Tagging APIs: PowerShell 20
6 Linked mode 21
6.1 Replication performance 21
6.2 Comparison to standalone vCenter tagging performance 25
7 Tags vs. custom attributes 28
8 Scale numbers for good performance 28
9 Conclusion 29
10 References 31
11 Appendix 32
11.1 Testbed setup and software versions 32
11.2 Java code examples 32
11.2.1 Example J0: VM DynamicID and tag TagID 32
11.2.2 Example J1: Retrieving category/tag IDs from category/tag names 33
11.2.3 Example J2: Create a category 35

11.2.4 Example J3: Create a tag 36

11.2.5 Example J4: Associating a tag with a VM (attach()) 36

11.2.6 Example J5: Associating a tag with multiple VMs (attachTagToMultipleObjects()) 37

11.2.7 Example J6: Associating multiple tags with multiple VMs (attachMultipleTagsToObject()) 38

11.2.8 Example J7: List tags associated with a VM 38

11.2.9 Example J8: List VMs associated with a tag 39

11.2.10 Example J9: List tags associated with a group of VMs 39

11.2.11 Example J10: List VMs associated with a group of tags 39

11.2.12 Example J11: Retrieving tag information (for example, name and description) from the tag ID 39

11.2.13 Example J12: Retrieving all associations using the pagination API 40

11.3 PowerShell examples 41

11.3.1 Example P0: Creating VMID objects and retrieving tag IDs 41

11.3.2 Example P1: List tags associated with a VM 43

11.3.3 Example P2: List VMs associated with a tag 43

11.3.4 Example P3: List tags associated with a group of VMs 43

11.3.5 Example P4: List VMs associated with a group of tags 44

11.3.6 Example P5: Retrieve associations using the pagination API 45

About the authors 46

Acknowledgments 46

1 Executive summary

Writing code to use VMware vCenter® tags can be challenging in large-scale environments. In this technical paper, we discuss the scalability limits of tags and give some tips and tricks for writing performant tagging code in Java or PowerShell.

This paper updates older versions of the vCenter tagging technical paper to 8.0 U3. Compared to the 7.0 U3 release, there are performance **speedups**[1] in tag association APIs due to optimized privilege checks. Specifically:

- The `attach()` call, which attaches a tag to an inventory item like a host or virtual machine (VM), shows a 40% speedup. (Shown in sections 4.2.1 and 4.2.2.)
- The `attachTagToMultipleObjects()` call, which attaches a tag to a group of inventory items, shows a 200% speedup. (Shown in sections 4.2.2 and 4.2.4.2.1.)
- The `attachMultipleTagsToObject()` call, which attaches a group of tags to an inventory item, shows a 31%–36% speedup. (Shown in sections 4.2.3 and 4.2.4.2.2.)

Note: These last two calls look similar, but if you look closely, you will see they are quite different. The first one is `attachTagToMultipleObjects()`; that is, one tag to many VMs. The second one is `attachMultipleTagsToObject()`; that is, many tags to only one VM.

In addition to tag association, we also show data regarding querying VMs associated with tags and Enhanced Linked Mode performance. vCenter 8.0 U3 can support the same limits as 7.0 U3 in terms of the number of tags, categories, and tag associations while offering improved performance.

2 Introduction

VMware vCenter 5.1 introduced a new feature: **inventory tagging**. **Tags** let you organize different vSphere objects like VMs, datastores, hosts, and others and makes it easier to sort and search for objects that share a tag, among other things.

Typically, you use a category for a high-level description. For example, a category might be **OS type** or **application name**. Within a category, the tags are the distinct values for that category. For example, if the category is **OS type**, then the tags might be **Linux** or **Windows 11**. If the category is **application name**, the tags might be **DB**, **Middleware**, or **vCenter**. If the category is **CPU type**, the tag values might be **AMD EPYC**, **Intel Broadwell**, or **Intel Raptor Lake**.

In this paper, we discuss the scalability limits for tags in vCenter 8.0 U3. We then provide best practices for writing code for tagging.

Note: The performance results in this paper are specific to the hardware configuration and vCenter Server Appliance (VCSA) configuration in our lab. However, the basic trends should be the same in your environment.

[1] For all percentage improvements that appear here and in section 4, we use the following formula based on [14]: `%speedup = 100(slow-fast)/fast`. We therefore refer to each improvement as showing a "speedup."

3 Terminology and scope of paper

Before describing the scale limits for tagging, let's first define some terms:

- **Categories:** A category is a group of tags, for example, **OS type** or **application name**.
- **Tag names:** Based on the previous category names, example tag names might be **Windows** for the category **OS type** and **SQL Server** for the category **application name**.
- **Tag associations:** A tag association is the mapping between a tag and an object (like a VM or datastore). For example, if you attach the tag **Windows** to the VM **vm-111**, that is 1 association. If tag **Windows** and tag **SQL Server** are both attached to **vm-111**, that is 2 associations. If you attached 25 tags (**Windows**, **SQL Server**, **Pasadena Datacenter**, and so on) with 1,000 VMs, that would be 25,000 tag associations.

There are no hardcoded limits to the number of tags, categories, and tag associations. However, increasing the numbers of tags, categories, and/or associations can impact performance.

In this paper, we have tried to provide information that you can use to understand the performance limits in your environment. We give best practices for how to write scripts to create tag associations and retrieve association information from vCenter.

We focus on Java and PowerCLI programs, although the same principles apply to other languages as well. Furthermore, our primary focus is on the operations listed below:

1. Creating tag definitions
2. Associating (attaching) tags to entities like VMs or datastores
3. Retrieving tag associations

These are the most common tasks performed by scripts or plugins.

4 Tagging APIs

There are three types of tagging APIs in vCenter: **category APIs**, **tag APIs**, and **tag association APIs**. Functionally, category APIs and tag APIs are similar, dealing with the creation, deletion, updating, and listing of categories or tags, respectively. We will, therefore, discuss them together. You use tag association APIs to create, delete, and list associations between tags and objects.

This section includes two main parts:

- Tag and category APIs
- Tag association APIs

4.1 Tag and category creation APIs

In vCenter, you can only create a tag as part of an existing category. Therefore, you must create a category before adding a tag to it. The performance considerations for creating categories are similar to those of creating tags, so we discuss only tag creation in this section. For examples of creating a category, please refer to section 11 Appendix.

Note: Also see the VMware vSphere Automation SDKs Programming Guide for VMware vCenter 8.0 [1].

When you create a tag, you must give it a name, description, and cardinality (described below). The list of associable types (that is, what types of objects, like VMs or hosts, can be associated with this tag) is optional. If you don't specify a list, the tag can be associated with any object. The tag creation API only lets you specify a single tag at a time.

The performance team tagging blog [2] describes cardinality. In brief:

- Cardinality refers to whether multiple tags or only one tag from a category can be attached to an object.
- Multiple cardinality means that you can attach multiple tags (for example, **Alice** and **Bob**) from a category (for example, **Owners**) to a given object (like a VM).
- If the cardinality is one, then you can only attach one tag within a category to an entity. For example, if the category is **OS**, you can attach only one tag from that category (for example, **Ubuntu Linux** or **Windows 11**) to the VM.

Figure 1 shows how long it takes (latency) to create 8,000 tag definitions under a single category. For example, if the category name were **Application**, the tag definitions might be **SQL Server**, **SAP HANA**, **Oracle**, **Kafka Messaging**, etc.

While the creation time from tag to tag varies, as seen in the figure, the time to create 8,000 tags under 1 category stays consistent from the 1st to 8,000th tag, averaging around 12 milliseconds (ms). The performance of this operation in 8.0 U3 is similar to that of 7.0 U3.

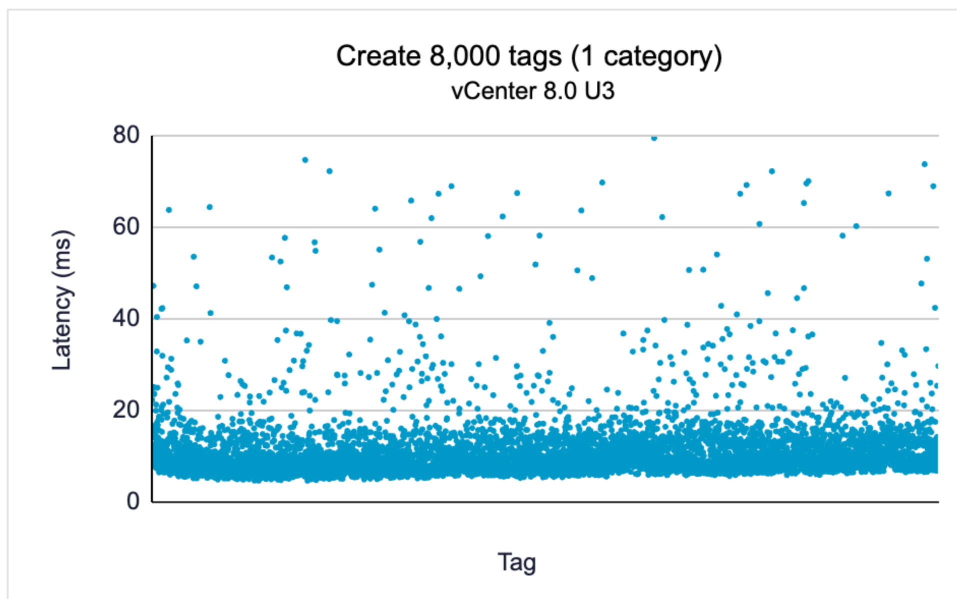


Figure 1: Creating 8,000 tags under 1 category. In 8.0 U3, the time to create a tag stays constant with the number of tags.

Figure 2 shows the time to create 8,000 tag definitions under 200 categories, evenly distributed so there are 40 tag definitions per category. As with the previous figure, the latencies show some variation, but the average latency to create tags is roughly constant: the time to create 40 tags per category is around 11 milliseconds per tag. This number is on par with creating all tags within a single category.

As described in the 7.0 U2 technical paper [3], if you want to create many tags at once and speed up the end-to-end time for tag creation, you might consider using multiple clients. While performance in 8.0 U3 is similar with one category vs. many categories, multiple categories allow for better parallelism when using multiple clients.

Section 11 Appendix includes sample code for creating a category and a tag within that category.

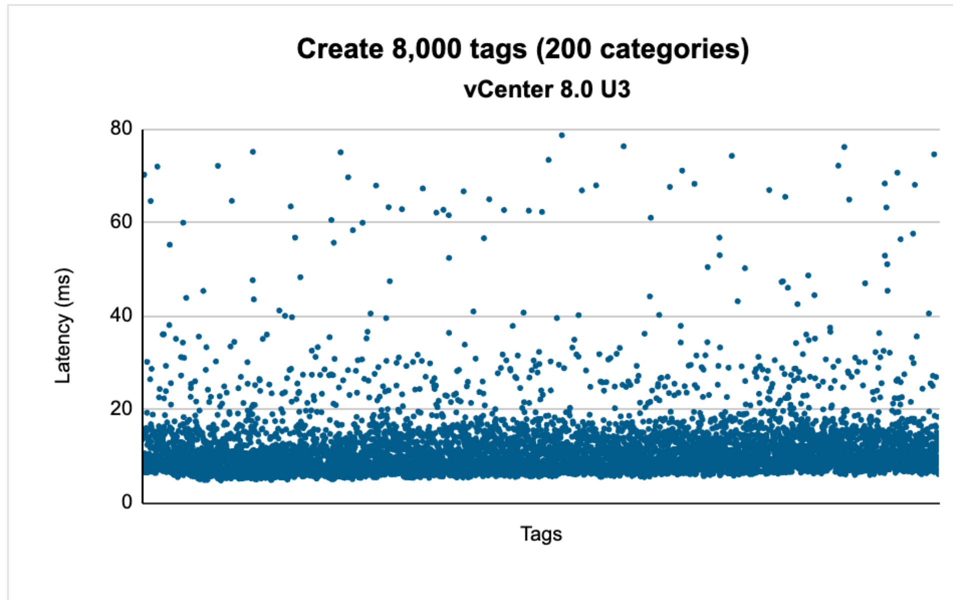


Figure 2: Creating 8,000 tags under 200 categories. Spreading tags among categories speeds up tag creation. This makes it roughly constant for small numbers of tags per category.

4.2 Tag association APIs

Now that we have discussed creating tags and categories, we next describe associating tags and categories with objects like VMs or hosts.

There are four ways to attach tags to one or more VMs:

- Option 1: Attach a single tag to a single VM using `attach(tagID, VM_ID)`, as described in section 4.2.1.
- Option 2: Attach a single tag to multiple VMs using `attachTagToMultipleObjects(tagID, list of VM_IDs)`, as described in section 4.2.2.
- Option 3: Attach multiple tags to a single VM using `attachMultipleTagsToObject(list of tagIDs, VMID)`, as described in section 4.2.3.
- Option 4: Various calls to attach multiple tags to multiple VMs, as described in section 4.2.4.

Each of these calls is useful in different scenarios. The main difference is how many calls to the tagging service are required.

4.2.1 Single tag to single VM

When attaching a single tag to a single VM, it is easiest to use a simple `attach()` call (option 1 above). Sample code for `attach()` is given in section 11 Appendix. We recommend keeping a map of tag ID to tag name and VM ID to VM name, if possible, to avoid having to call vCenter or the tagging service each time such a mapping is needed. The pseudocode for using `attach()` is given here:

```
1. Get the tag ID (tagID) from the tag name.
2. Get the VM ID (VM_ID) from the VM name.
3. attach(tagID, VM_ID)
```

Figure 3 shows the time to attach a tag to a VM as we increase the number of VMs from 1 to 10,000. The latency varies, but on average attaching a single tag to a single VM takes around 10 ms, with the time staying roughly constant from the first to last `attach()` operation. The total end-to-end time is 1.7 minutes. **The 10 ms average time for 8.0 U3 is a 40% performance speedup[1] over 7.0 U3, which was 14 ms.** This is due to optimized privilege checks. [2]

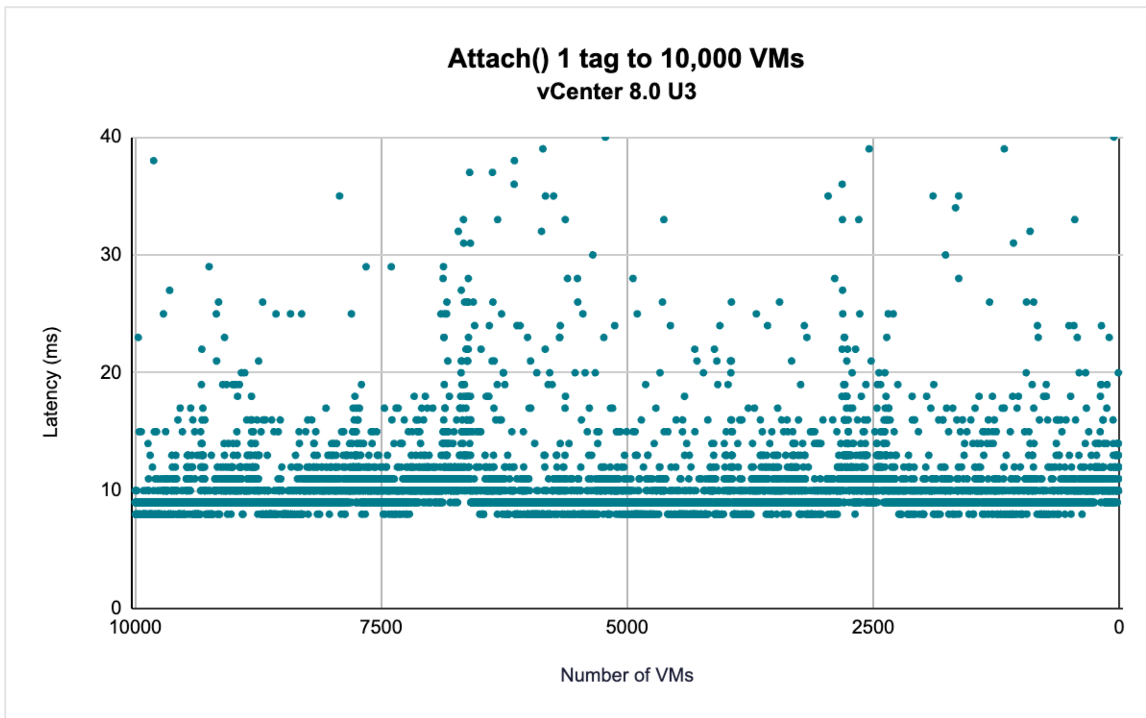


Figure 3: Using attach() to associate 1 tag with each of the 10,000 VMs. The time for an individual attach() is roughly constant even as we increase the number of VMs.

[2] All performance speedups for tag association calls shown in section 4.2 are due to optimized privilege checks.

4.2.2 Single tag to multiple VMs

When attaching a single tag to multiple VMs, it is easiest to use option 2, `attachTagToMultipleObjects()`. This call amortizes the cost of attaching the tag over multiple VMs. For attaching the same tag to 10 VMs (VMID_0-VMID_9), only one call to the tagging service is required, as opposed to 10 if we had used an `attach(tagID,VMID_N)` call in a loop over each VM. See section 11 Appendix for an example of attaching a single tag to multiple VMs using `attachTagToMultipleObjects()`. We show the pseudocode here:

```
1. Get tag ID (String tagID)
2. Get list of VM Dynamic IDs (List<DynamicID> VM_IDs)
3. attachTagToMultipleObjects(tagID,VM_IDs)
```

As you might expect, attaching a single tag to multiple VMs using `attachTagToMultipleObjects()` is more efficient than looping over every VM and calling `attach()`. Table 1 compares the latency of attaching a single tag to multiple VMs using `attachTagToMultipleObjects()` vs. using `attach()`. For `attachTagToMultipleObjects()`, a single call attaches the tag to 20,000 VMs. In the case of `attach()`, we sequentially attach the tag to each VM. **Both calls perform better in 8.0 U3 compared to 7.0 U3, with `attach()` having a 40% speedup[1], and `attachTagToMultipleObjects()` having about a 200% speedup[1]**, as shown in table 1.

Operation	Number of iterations	Latency for 1 tag & 20,000 VMs vSphere 8.0 U3	Latency for 1 tag & 20,000 VMs vSphere 7.0 U3
attach()	20,000 (1 per VM)	205 seconds	277 seconds
attachTagToMultipleObjects()	1 (20,000 VMs at a time)	25 seconds	83 seconds

Table 1: attach() vs. attachTagToMultipleObjects()

4.2.3 Multiple tags to a single VM

When attaching multiple tags to a single VM, it is best to use option 3, `attachMultipleTagsToObject()`. We can attach 10 tags (tag_0-tag_9) to a single VM with one tagging service call instead of the 10 that would be required if we used `attach(tag_N,VMID)` in a loop over each tag. See section 11 Appendix for an example of attaching multiple tags to VMs.

Here is the pseudocode to accomplish this:

```
1. Get tag IDs and put them into a list (List<String> tagIDs)
2. Get VM ID (VM_ID)
3. attachMultipleTagsToObject(VM_ID,tagIDs)
```

Figure 4 shows the latency of attaching 1 through 200 tags to a single VM using `attachMultipleTagsToObject()`. Operations with smaller tag counts are more susceptible to variability across the test runs, which is why the result with fewer tags performs slightly worse than with larger tags. As the number of tags increases, the incremental overhead is linear but with a slight slope. Compared to 7.0 U3, which was 41 seconds for the 200 tag datapoint, **8.0 U3 shows a 36% speedup[1] at the same datapoint of 30 seconds.** As the data stabilized for each version of vSphere, 8.0 U3 generally showed better performance across all datapoints (number of tags).

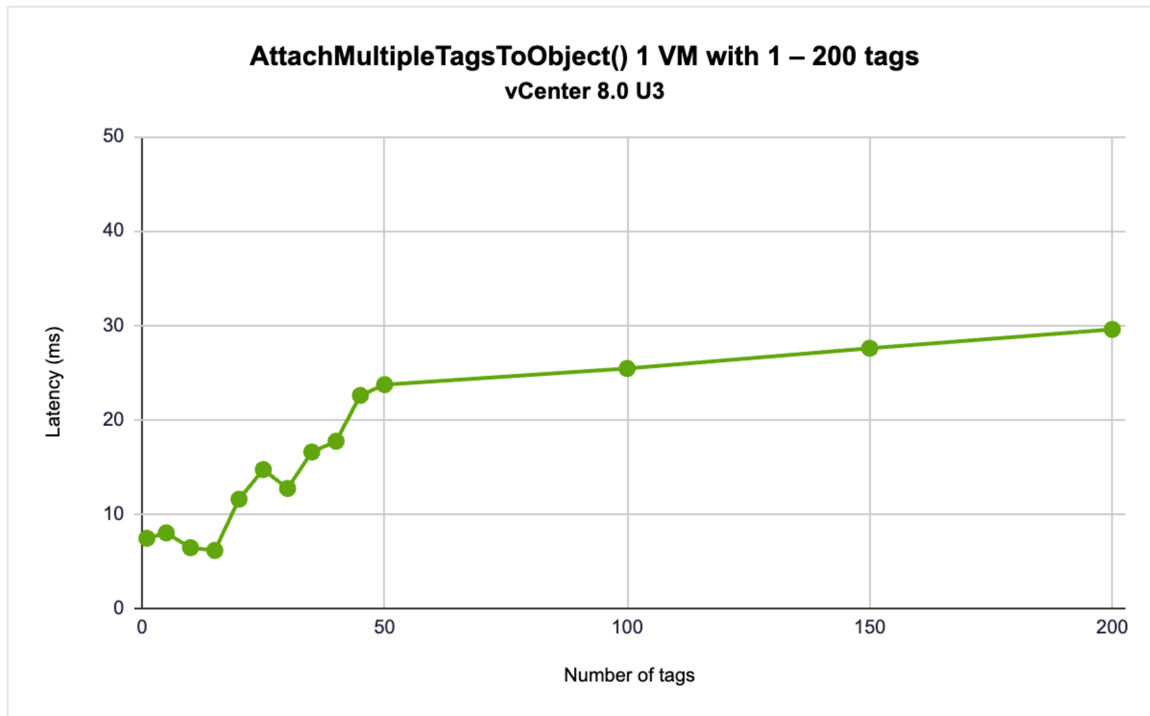


Figure 4: attachMultipleTagsToObject() for 1-200 tags and 1 VM.

For comparison purposes, figure 5 compares the latency of attaching multiple tags (1 up to 2,000) to a single VM using all three methods. Notice that they have similar latencies at small numbers of tags, but `attachMultipleTagsToObject()` scales substantially better as the number of tags increases. Additionally, the latency for a single tag using `attachMultipleTagsToObject()` is around 25 ms, while the latency to attach 2,000 tags is around 200 ms. In contrast, the latency for `attach()` grows from 25 ms with 1 tag to nearly 40 seconds for 2,000 tags, and `attachTagToMultipleObjects()` shows a similar latency increase. Because `attachMultipleTagsToObject()` is significantly faster, it is the preferred API for this use case.

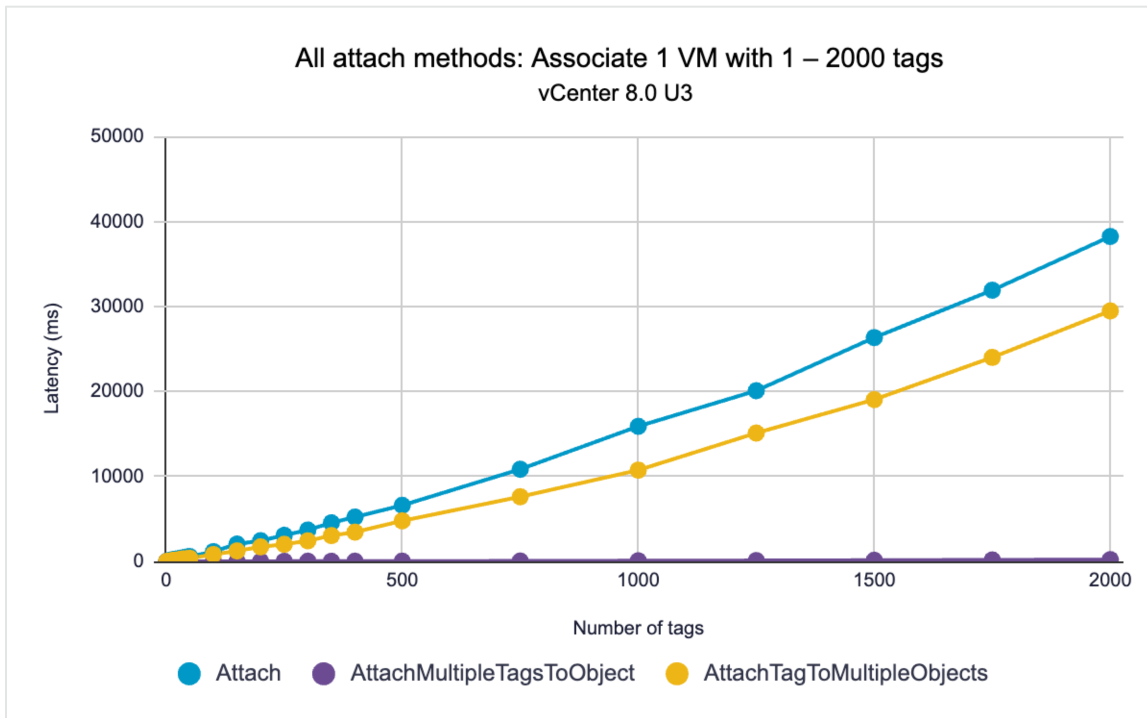


Figure 5: Attaching 1 through 2,000 tags to 1 VM. attachMultipleTagsToObject() is well suited for this use case.

4.2.4 Multiple tags to multiple VMs

In this section, we discuss assigning multiple tags to multiple VMs and provide two examples illustrating how you would carry out each operation. The examples are:

• 10 tags associated with 1,000 VMs for a total of 10,000 tag associations (section 4.2.4.1)

• 15 tags associated with 5,000 VMs for a total of 75,000 tag associations (section 4.2.4.2)

4.2.4.1 10 tags associated with 1,000 VMs (10,000 tag associations)

Suppose we have a group of 10 tags (for example, **Windows 11**, **DB app**, etc.), and we want to attach them to a group of 1,000 VMs (for example, VM_0, VM_1, …, VM_999).

As the preceding discussion suggests, there are two ways to accomplish this:

1. `attachTagToMultipleObjects()` pseudocode:

```
Get tag IDs ➔ List<String> tagIDs

Get VM IDs ➔ List<DynamicId> VM_IDs

for (String tagID : tagIDs) {

     attachTagToMultipleObjects(tagID, VM_IDs)

}
```

2. `attachMultipleTagsToObject()` pseudocode:

```
Get tag IDs ➔ List<String> tagIDs

Get VM IDs ➔ List<DynamicId> VM_IDs

for (DynamicID VM_ID : VM_IDs) {

     attachMultipleTagsToObject(VM_ID, tagIDs)

}
```

We give more complete code snippets in section 11 Appendix.

In table 2 below, we show the relative performance of these two calls for assigning 10 tags to 1,000 VMs from our internal testing labs.

Operation	Number of iterations	End-to-end Latency	Time per iteration
attachTagToMultipleObjects()	10	14 seconds	1.4 seconds
attachMultipleTagsToObject()	1,000	10.8 seconds	10.8 milliseconds

Table 2: Latency for assigning 10 tags to 1,000 VMs (a total of 10,000 associations). The end-to-end latency depends on the per-iteration cost.

As the table indicates, for this modest number of tags and VMs, `attachMultipleTagsToObject()` is faster than `attachTagToMultipleObjects()`, even though `attachMultipleTagsToObject()`runs more iterations. In the first case, each iteration is creating 1,000 tag associations (that is, 1 tag is being associated with 1,000 VMs). In the second case, each iteration is creating 10 associations (that is, associating 10 tags to 1 VM).

4.2.4.2 15 tags associated with 5,000 VMs (75,000 tag associations)

In the previous example, the number of associations (10,000) was small enough that the latency to create each association was constant. However, the latency to attach tags to VMs increases as the number of associations increases. Therefore, we offer two ways to implement this operation:

- Using `attachTagToMultipleObjects()`, shown in section 4.2.4.2.1
- Using `attachMultipleTagsToObject()`, shown in section 4.2.4.2.2.

As noted in section 1 Executive summary, these two calls look similar:

• The first one is `attachTagToMultipleObjects()`; that is, one tag to many VMs.

• The second one is `attachMultipleTagsToObject()`; that is, many tags to only one VM.

We show how you can attach many tags to many VMs using both calls. The second example uses a script with a loop as a workaround.

4.2.4.2.1 attachTagToMultipleObjects()

Figure 6 shows the latency of attaching multiple (1, 5, 10, 15) tags to up to 5,000 VMs using `attachTagToMultipleObjects()`. This creates up to 75,000 tag associations—that is 15 x 5,000 = 75,000.

In the graph below, the time taken to attach the first tag to all VMs is around 7 seconds (compared to 21 seconds for 7.0 U3), and the time to attach the 15th tag to all VMs is about the same**. There is a 200% performance speedup[1] for 8.0 U3 compared to 7.0 U3.** The end-to-end time to attach all 15 tags is the sum of the time for each iteration, around 100 seconds compared to around 320 seconds for 7.0 U3.

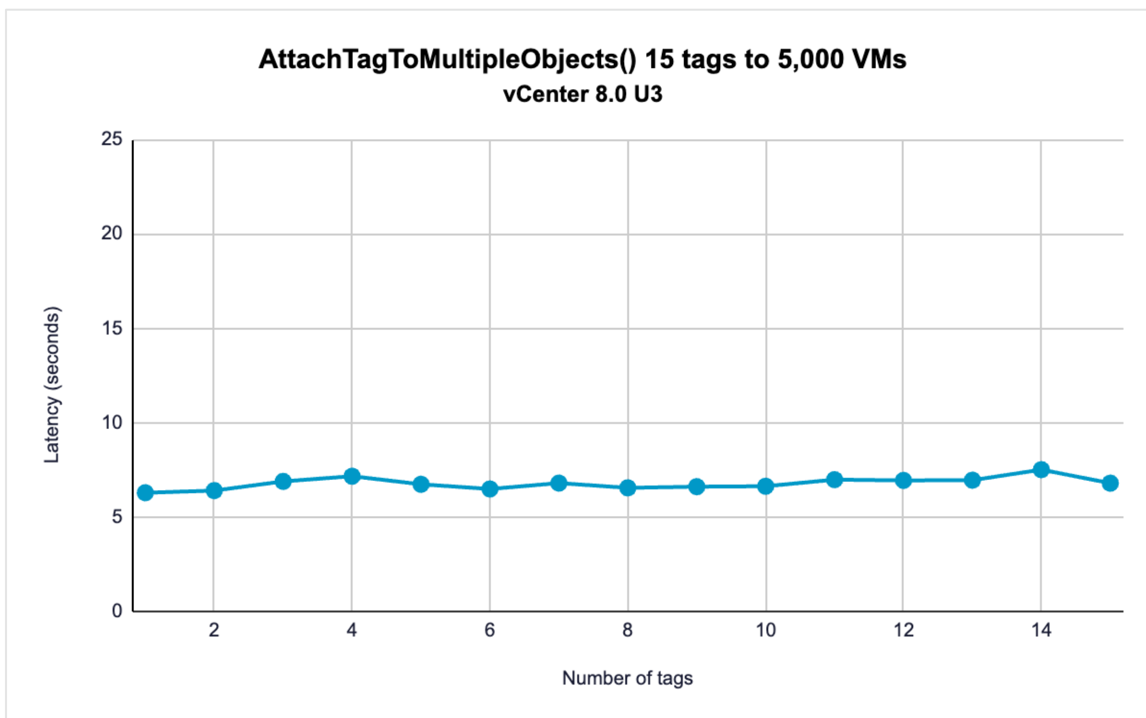


Figure 6: attachTagToMultipleObjects(): 15 tags to 5,000 VMs.

4.2.4.2.2 attachMultipleTagsToObject()

We now look at attaching 15 tags to 5,000 VMs using `attachMultipleTagsToObject()` instead. The results are in figure 7. To use `attachMultipleTagsToObject()`, we loop over VMs one at a time and attach 15 tags at once, as shown in the following pseudocode:

```
Get tag IDs ➜ List<String> tagIDs
Get VM IDs ➜ List<DynamicId> VM_IDs

for (DynamicID VM_ID : VM_IDs) {
   attachMultipleTagsToObject(VM_ID, tagIDs)
}
```

As figure 7 indicates, the time to attach 15 tags to a VM is typically between 5 and 15 milliseconds. The total time to attach 15 tags to 5,000 VMs is the sum of the latency of each iteration, around 51 ms. **The performance speedup[1] for 8.0 U3 is 31% faster than that of 7.0 U3, which was 67 ms.**

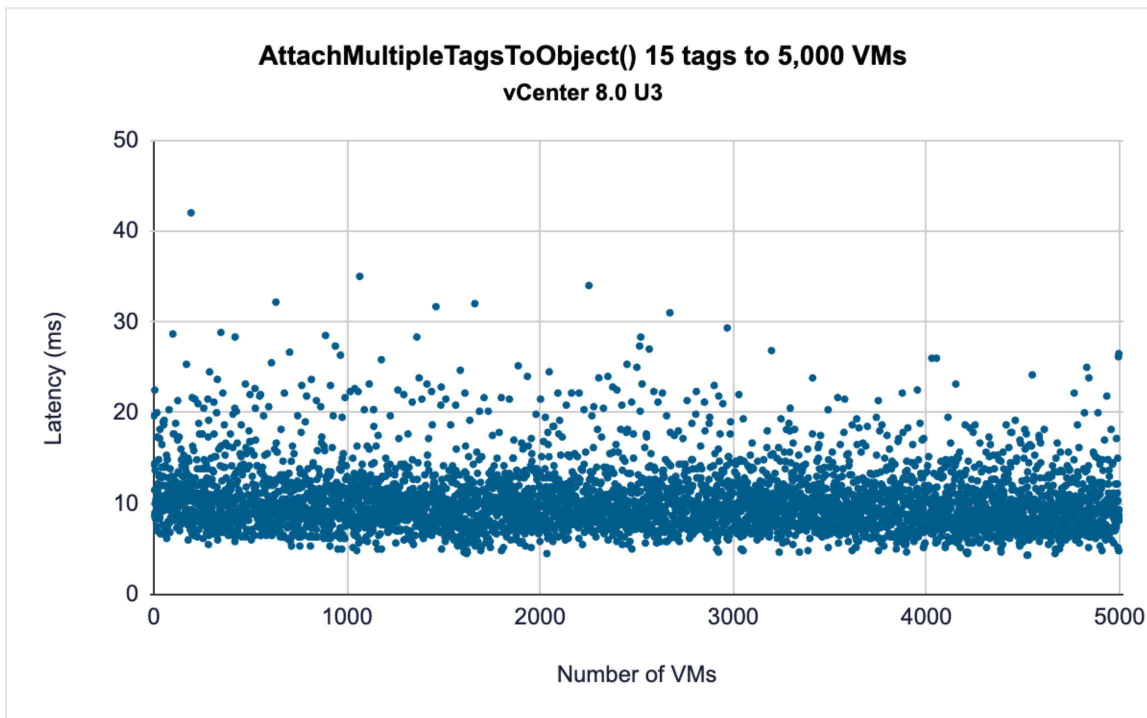


Figure 7: attachMultipleTagsToObject(). This call is roughly constant with the number of objects.

In table 3, we summarize the performance of these API calls when attaching 15 tags to 5,000 VMs (75,000 tag associations). For end-to-end latency, lower is better.

Operation	Number of iterations	End-to-end latency vCenter 8.0 U3	End-to-end latency vCenter 7.0 U3
attachTagToMultipleObjects()	15 (one per tag)	1.7 minutes	5.3 minutes
attachMultipleTagsToObject()	5,000 (one per VM)	0.9 minutes	1.1 minutes

Table 3: Latency for assigning 15 tags to 5,000 VMs (a total of 75,000 associations). The end-to-end latency depends on the per-association cost for attachTagToMultipleObjects() and the per-iteration cost for attachMultipleTagsToObject().

When attaching multiple tags to multiple objects, we recommend using `attachMultipleTagsToObject()`.

Table 4 shows a few more data points: 1, 5, 10, 15, or 20 (hereafter referred to as {1,5,10,15,20}) tags associated with 5,000 VMs. We also performed a test where we attached 20 tags to 10,000 VMs. For large numbers of VMs and **more than one tag per VM**, `attachMultipleTagsToObject()` scales the best. With **one tag and many VMs**, `attachTagToMultipleObjects()` scales the best.

Operation	1 tag to 5,000 VMs (5,000 associations)	5 tags to 5,000 VMs (25,000 associations)	10 tags to 5,000 VMs (50,000 associations)	15 tags to 5,000 VMs (75,000 associations)	20 tags to 5,000 VMs (100,000 associations)	20 tags to 10,000 VMs (200,000 associations)
attach()	52 seconds	4.3 minutes	8.5 minutes	12.8 minutes	17 minutes	34 minutes
attachTagToMultipleObjects()	7 seconds	33.5 seconds	1.1 minutes	1.7 minutes	2.3 minutes	4.6 minutes
attachMultipleTagsToObject()	32 seconds	43 seconds	47 seconds	51 seconds	1.4 minutes	2.7 minutes

Table 4: End-to-end latency to attach tags to VMs. We show {1,5,10,15,20} tags associated with 5,000 VMs, and 20 tags associated with 10,000 VMs. `attachMultipleTagsToObject()` scales the best when the number of VMs is large.

The following are the key takeaways from the table:

- The time to perform a single association increases as more tag associations are added.
- Chunking using `attachTagToMultipleObjects()` or `attachMultipleTagsToObject()` is preferable to using `attach()` to loop over each tag/VM pair.

As an additional consideration, the tagging service is subject to a limit of 1,500 tagging-related operations per second to prevent denial-of-service attacks. To detect if you are hitting this limit, look for messages in the vapi-endpoint log file such as `Request rejected due to high request rate. Try again later.` If your creation code exceeds this limit, we recommend inserting a pause of 2 milliseconds or more between tag-creation calls.

4.3 Associate tags with other objects

Associating tags with other objects (for example, hosts, datastores, or content library items) has the same tradeoffs as associating tags with VMs, so the preceding discussion applies. As with the discussion on VMs, we have four cases to consider:

1. To associate 1 tag with 1 object (for example, a host), we recommend using `attach(tagID, HostID)`.
2. To associate 1 tag with multiple hosts, we recommend using `attachTagToMultipleObjects(tagID, HostID [])`.
3. To associate multiple tags with 1 host, we recommend using `attachMultipleTagsToObject(tagID[], HostID)`.
4. To associate multiple tags with multiple hosts, we recommend looping over hosts and calling `attachMultipleTagsToObject(tagID[], HostID)` for each host.

The tagging APIs for VM and host associations have been optimized. Therefore, most operations might be slower with datastores or other objects vs. VMs and hosts. For example, attaching a single datastore to a single tag might take up to 30 milliseconds, while attaching a single VM to a single tag is typically around 10 milliseconds. The performance difference depends on inventory size and the number of tags, and for most customer setups, the difference will be much lower than what we have indicated above.

4.4 Query VMs associated with tags

To find the VMs associated with a tag, there are five APIs:

1. `List<TagAssociation.objectToTags> listAttachedObjectsOnTags(List<String> tagIDs)`: Given a list of tagIDs, this call returns the list of objects that are associated with these tags. The return value is a tuple that shows the object ID (as a DynamicID) and tag ID. A DynamicID is an object that contains an identifier and a type. You can view examples in section 11 Appendix.
2. `List<DynamicId> listAttachedObjects(tagID)`: Given a single tagID, this call returns a list of all objects (as Dynamic IDs) associated with this tag ID.
3. `List<TagAssocation.tagToObjects> listAttachedTagsOnObjects(List<DynamicID> objectIDs)`: Given a list of VM_IDs, this function returns all tags associated with those VMs. The return value is a tuple that shows the VM ID and its associated tag IDs.
4. `List<String> listAttachedTags(objectID)`: Given a VM, this call lists all tags associated with the VM.
5. `List<ListResult> list(iterationSpec)` (Pagination API): This call lists all the tag associations that are present in vCenter, provided a page at a time, with a maximum limit per page (default: 5,000). In addition to returning a page of data, this call also returns a marker, which is used to get the next page of data. Please note that this pagination-based API is accessed via `https://{api-host}/api/vcenter/tagging/associations`, while the other APIs are accessed via `https://{api-host}/api/cis/tagging/tag-association`.

One important point to note when performing query calls is that the call will fail if the response exceeds the maximum limit of 7MB in vCenter 8.0 U3. If the operation fails due to exceeding the limit, you'll see an error in the log file for the vapi-endpoint service `/var/log/vmware/vapi/endpoint/endpoint.log` such as this: `Error: Response size is greater than allowed 7000000b)`.

Figure 8 below shows the latency to retrieve associated tags or associated objects for 1, 5, 10, or 15 ({1,5,10,15}) tags associated with 20,000 VMs for 4 APIs: `listAttachedObjects()`, `listAttachedObjectsOnTags()`, `listAttachedTagsOnObjects()`, and list using pagination, which is `list()`. In figure 9, we do the same experiment using the `listAttachedTags()` API. We display it separately due to its slower performance compared to other methods, which would result in the other plot lines clustering closely and making them difficult to view.

In figure 8 below, we see the following:

- The `listAttachedObjects()` call takes a single tag as an input. When we have 15 tags, we loop over the 15 tags. Each call returns 20,000 VMs. We have at most 15 iterations (one for each tag). The latency goes from around 1 second to 15 seconds.
- In contrast, the `listAttachedObjectsOnTags()` call takes a list of tags and returns the objects attached to them. Each call again returns 20,000 VMs. In our experiments, we used 1 tag per call, rather than providing a list of tags, because when we provided too many tags, the call failed due to exceeding response size limits (7MB). We have at most 15 iterations. Because we request tags using the same technique as `listAttachedObjects()`, the latencies are very similar.
- The `listAttachedTagsOnObjects()` call allows us to specify a list of objects. To prevent the call from failing due to exceeding response size limits (7MB), we do not retrieve the tags for all 20,000 VMs at once. Instead, we break down the list of 20,000 VMs into chunks of 2,000 VMs. For 1 tag, we provide 1 tag and iterate over the VMs in chunks of 2,000, for a total of 10 iterations. For 15 tags, we provide an array of 15 tags and again iterate over VMs in groups of 2,000, for a total of 10 iterations to get associations for 20,000 VMs. This method scales the best, finishing nearly 2x faster than the other approaches.
- The `list()` (pagination API) call returns associations in a paged manner (for example, if we have 50,000 tag associations in vCenter, the pagination API will get all the tag associations in chunks of 5,000 per iteration). In this case, it will take 10 iterations to get all 50,000 tag associations. Using this API, a client does not need to manually break down the objects into chunks.

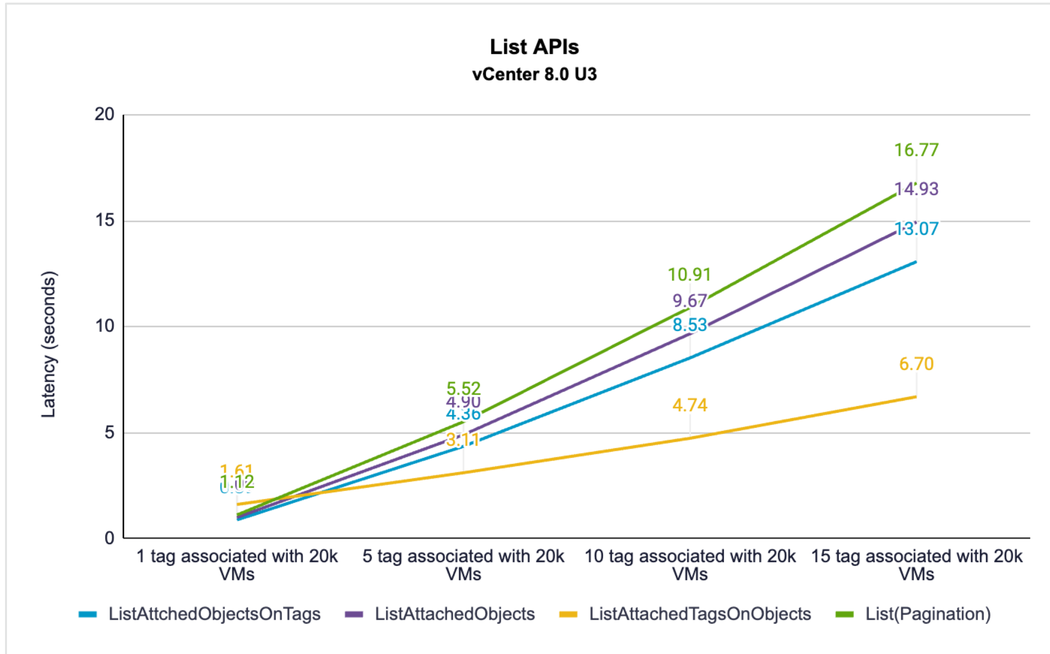


Figure 8: End-to-end time to retrieve associations for {1,5,10,15} tags attached to 20,000 VMs using various List APIs.

In figure 9 below, we see that `listAttachedTags()` is much slower than the other approaches, taking minutes instead of seconds. This call takes only 1 object as an input and returns the appropriate number of attached tags. For 20,000 VMs, we need 20,000 iterations, which is substantially higher than for the other approaches.

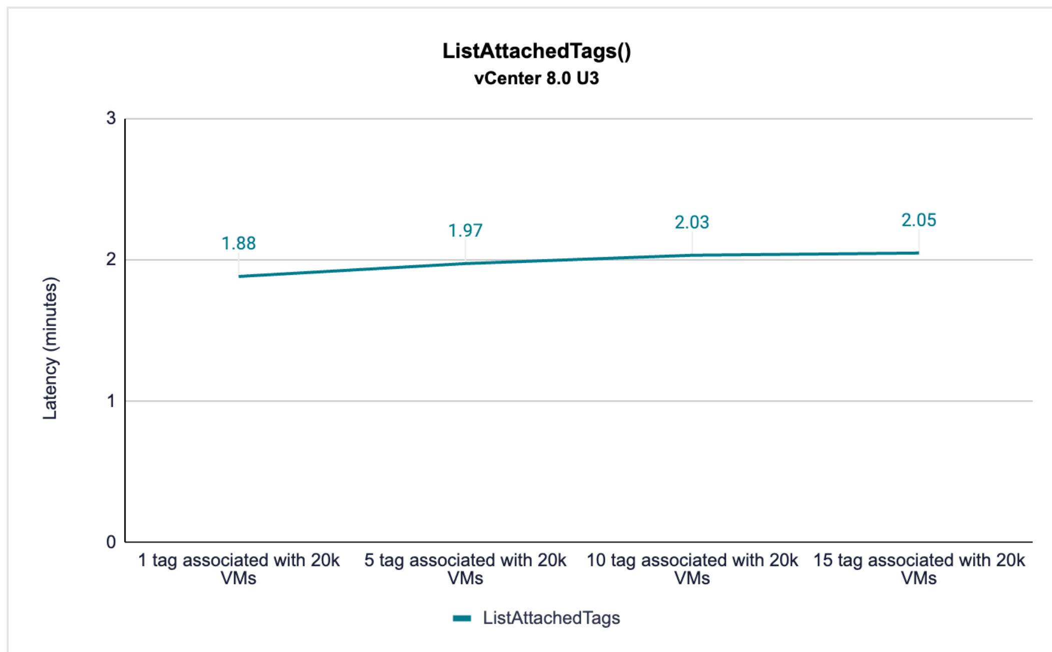


Figure 9: End-to-end time to retrieve associations for {1,5,10,15} tags attached to 20,000 VMs using listAttachedTags() API. listAttachedTags() is much slower than the other list calls, and we do not recommend using it in environments with more than 2,000 VMs.

We summarize the results in table 5 below.

Operation	1 tag associated with 20K VMs	5 tags associated with 20K VMs	10 tags associated with 20K VMs	15 tags associated with 20K VMs
listAttachedObjects()	1.0 seconds	4.9 seconds	9.7 seconds	14.9 seconds
listAttachObjectsOnTags()	0.9 second	4.4 seconds	8.5 seconds	13.1 seconds
listAttachedTagsOnObjects()	1.6 seconds	3.1 seconds	4.7 seconds	6.7 seconds
listAttachedTags()	1.88 minutes	1.97 minutes	2.03 minutes	2.05 minutes
list()	1.1 seconds	5.5 seconds	10.9 seconds	16.8 seconds

Table 5: Latency for various list APIs. It is best to use APIs like listAttachedTagsOnObjects() or the pagination API rather than listAttachedTags().

Because `listAttachedObjectsOnTags()` and `listAttachedObjects()` can return every object in the system, their performance is very sensitive to inventory size. Instead, if it is an option, we recommend using `listAttachedTagsOnObjects()`, since it allows you to break down the number of objects into smaller chunks and give more predictable performance per call. However, if you delete any of the input objects while using `listAttachedTagsOnObjects()`, the call will return errors for that object, requiring the client to handle those errors. Alternatively, you can use the pagination API `list()`, which also gives predictable performance so that you do not have to manually break the object list into chunks of 2,000 entities to account for the 7MB response size limits. In addition, the pagination API will return only valid associations, rather than returning an error if an object is no longer present in the inventory.

It is important to note that the list APIs above return IDs. Most client applications require names. To retrieve tag names from these IDs, you must use the `Get Tag ID` calls. For example, `tag_id` returns a TagModel object that has `name`, `id`, `category_id`, `description`, and `used_by` fields [4]. For performance reasons, we recommend maintaining a mapping of IDs to names on the client side, if practical. Tag names are scoped to categories, so when storing a tag name, be sure to store the category name as well.

As earlier mentioned, we recommended that chunking using `attachTagToMultipleObjects()` or `attachMultipleTagsToObject()`was preferable to using `attach()` to loop over each tag/VM pair. Similarly, we recommend chunking here too. For example, consider getting the mapping of all VMs associated with each tag. One way to do this is using `listAttachedObjects()`as shown in the following pseudocode:

```
Get all categories
Get tags for each category
Get VMs associated with Tag using listAttachedObjects()
```

Because the number of objects associated with a tag may exceed 2,000 and risk hitting the response size limits, we recommend using `listAttachedTagsOnObjects()` instead. Because we don't know how many tags will be associated with each object, to avoid overflowing internal vapi-endpoint buffers, one option is to loop over each object individually, as shown in this pseudocode:

```
Get all objects using vSphere API for retrieving objects (List <DynamicId> objectList)
for (DynamicId object : objectList) {
     List<Tag.objectToTags> result = listAttachedTagsOnObjects(object)
}
```

We could have used `listAttachedTagsOnObjects()`, `listAttachedTags()`, or `list()`, but we chose `listAttachedTagsOnObjects()` (iterating over each object individually) because the return value of `listAttachedTagsOnObjects()` is a mapping of tag to object. If we had used `listAttachedTags()` or `list()`, we would have had to manually create such a mapping.

For an example on retrieving all associations using the pagination API, refer to section 11 Appendix.

The key takeaway from the discussion of the list APIs is that two APIs offer predictable performance and are the recommended ways to retrieve associations:

1. The pagination API, `list()`
2. The API: `listAttachedTagsOnObjects()`

The pagination API `list()` offers robust, predictable performance since it returns a fixed number of associations per iteration without requiring the client to shard the data into chunks. The pagination API `list()` also retrieves associations only for objects that are currently present in the inventory, so it does not return errors in case an object has been deleted after `list()` has been called but before associations have been retrieved. The `listAttachedTagsOnObjects()` API also offers predictable performance but requires you to properly chunk the list of objects into smaller groups to avoid hitting response size limits. In addition, if the list of objects includes deleted objects, then `listAttachedTagsOnObjects()` will return errors for those objects, so a client must handle those errors.

5 Tagging APIs: PowerShell

The previous examples all used Java. In this section, we discuss using PowerShell. As described in Writing Performant Tagging Code: Tips and Tricks for PowerCLI [2], there are typically two ways to write PowerShell scripts for vCenter tags: either using cmdlets or using direct tagging APIs. For performance reasons, we recommend using direct tagging APIs instead of cmdlets, as indicated in the blog article.

The direct tagging service calls in PowerShell use the same vCenter server and tagging service resources as Java. Therefore, the performance results are the same as the Java results above. In this section, we focus on the differences in the API calls themselves, which differ slightly from Java to PowerShell. Java uses camelCase, while PowerShell puts underscores in the method name. We give some examples below in table 6.

Description	Java	PowerShell (CIS service)
Attach one tag to one VM	attach()	Attach()
Attach one tag to multiple objects	attachTagToMultipleObjects()	Attach_tag_to_multiple_objects()
Attach multiple tags to one object	attachMultipleTagsToObject()	Attach_multiple_tags_to_object()
List objects attached to a tag	listAttachedObjectsOnTags()	List_attached_objects_on_tags()
List associations (pagination)	list()	List() (instead of com.vmware.cis, please use com.vmware.vcenter)

Table 6: Examples of differences between Java and PowerShell method names. Note that the pagination API for PowerShell is at a different location from the other association-based APIs, using com.vmware.vcenter instead of com.vmware.cis.

There are also some minor differences in how arrays and objects are created between PowerShell and Java. Please see the examples in section 11 Appendix for details. At the time of this writing, PowerCLI version 13.3 is compatible with vCenter 8.0 U3. Also refer to the VMware PowerCLI documentation [4].

6 Linked mode

In an on-premises environment, Enhanced Linked Mode (ELM) [5] offers a unified view for all linked vCenter servers.

In ELM, linked vCenter servers replicate tag and category information between peer nodes periodically. For vCenter servers connected in linked mode, there are two main considerations:

• How long does it take to replicate tag and category information across vCenter servers?

• What is the performance of tag and category operations when carried out against an individual vCenter?

6.1 Replication performance

Each vCenter server in linked mode replicates data with its peer every 30 seconds, so changes to categories and tag definitions on one node may take some time before they propagate to the other nodes in the ELM setup. The choice of topology can impact this replication time.

For example, figure 10 shows the recommended ring topology [6] of 15 linked vCenter servers.

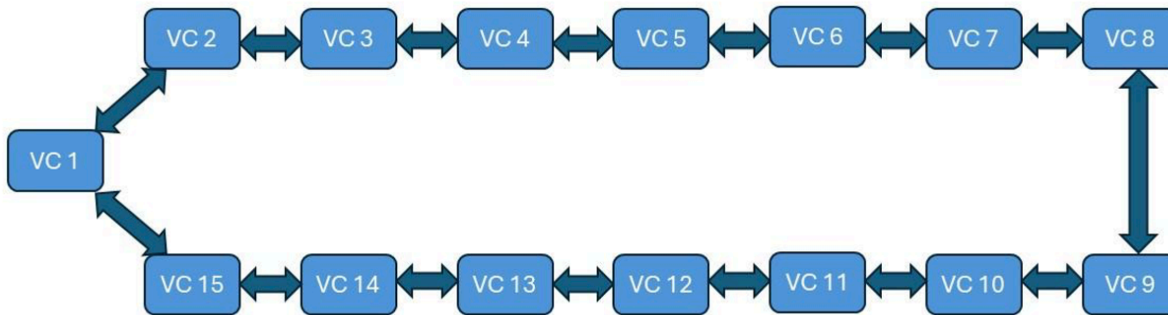


Figure 10: Ring topology for linked mode of 15 vCenters

In the figure above, the replication partners are as follows:

- VC 1: VC 2 and VC 15
- VC 2: VC 1 and VC 3
- VC 3: VC 2 and VC 4
- …
- VC 13: VC 12 and VC 14
- VC 14: VC 13 and VC 15
- VC 15: VC 14 and VC 1

When a new tag is added on VC 1, here is a rough timeline of replication:

- 0 - Time 0: The tag is added to VC 1.
- 1 - Up to 30 seconds later: VC 1 replicates to VC 2 and VC 15.
- 2 - Up to 30 seconds later: VC 2 replicates to VC 3; VC 15 replicates to VC 14.
- 3 - Up to 30 seconds later: VC 3 replicates to VC 4; VC 13 also replicates to VC 12.
- 4 - Up to 30 seconds later: VC 4 replicates to VC 5; VC 12 also replicates to VC 11.
- 5 - Up to 30 seconds later: VC 5 replicates to VC 6; VC 11 also replicates to VC 10.
- 6 - Up to 30 seconds later: VC 6 replicates to VC 7; VC 10 also replicates to VC 9.
- 7 - Up to 30 seconds later: VC 7 replicates to VC 8; VC 9 also replicates to VC 8.
- 8 - Up to 30 seconds later: VC 8 and VC 9 try to replicate to each other, but that change is already synced. ELM performs conflict resolution to ensure only one new tag definition is created on any VC.

As the timeline above suggests, in this 15-node ring topology, a tag may take up to 210 seconds to replicate throughout the system. Because each node pushes changes at most 30-second intervals, this is the worst-case latency. Since nodes may not simultaneously push changes to each other, it is possible for a tag change to occur just before a scheduled node pushes changes. This scenario would result in a faster reflection of the changes.

In our experiments using a ring topology of 15 linked mode vCenter servers, we observed that the replication process takes approximately 120 seconds to complete when a single vCenter server creates a tag. When we run a workload creating tags on all vCenter servers simultaneously, it takes close to 180 seconds for all nodes to converge to the same tagging view. Both are quicker than the worst-case scenario explained above.

Different topologies will have different delays. For example, suppose VC 1 is not connected to VC 15, as shown below in figure 11, and VC 1 is creating a tag.

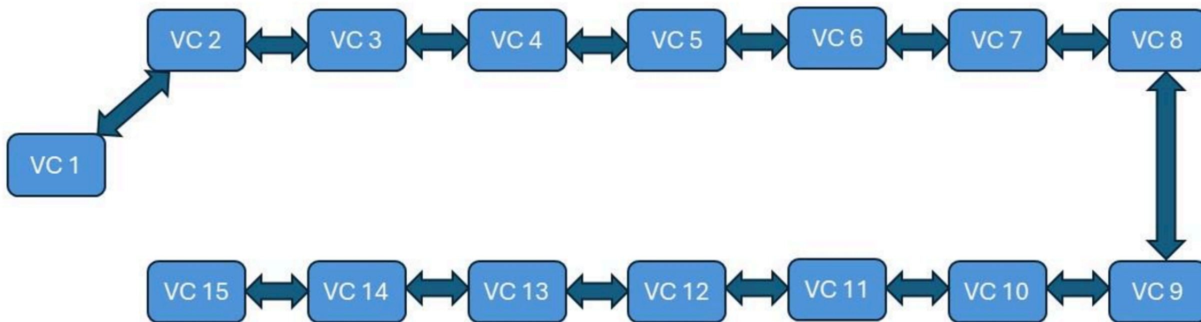


Figure 11: Linear topology for linked mode of 15 vCenter servers. We do not recommend this approach due to the latency of tag/category replication.

- 0 - Time 0: The tag is added to VC 1.
- 1 - Up to 30 seconds later: VC 1 replicates to VC 2.
- 2 - Up to 30 seconds later: VC 2 replicates to VC 3.
- 3 - Up to 30 seconds later: VC 3 replicates to VC 4.
- …
- 13 - Up to 30 seconds later: VC 13 replicates to VC 14.
- 14 - Up to 30 seconds later: VC 14 replicates to VC 15.

The timeline indicates that the worst-case latency for the last node to receive a tag update is 30 seconds * 14 = 420 seconds after the tag's addition. The average case is much quicker since the sync between neighboring nodes does not wait for the whole 30 seconds.

To summarize, the choice of ELM topology impacts the time it takes to propagate tag and category changes. A ring topology is the suggested configuration for ELM, with a worst-case replication latency of (N/2)*30s, while a linear topology has a worst-case replication latency of N*30s, where N is the number of vCenter servers in linked mode.

In practice, we observe much faster performance as neighboring vCenter servers sync at different times without waiting for a full 30-second replication cycle before pushing changes to the replication peers. Figure 12 is an example in which vCenter servers do not wait 30 seconds to synchronize with replication peers. We can track how a change in tags/categories propagates among linked vCenter servers:

- We create a single tag on VC 1 at 17:44:10.
- Every 5 seconds, we have an external agent that probes all vCenter servers to see when the newly created tag shows up.

In figure 12, we see how the newly created tag on VC1 is propagated across all linked vCenter servers.

- The new tag is available at VC 1 at 17:44:15.
- By 17:44:40, the new tag is already propagated to not only VC 2 and VC 15, which are the two replication partners of VC 1, but also to VC 13 and VC 14.
- In a ring topology, the new tag is sequentially propagated to neighboring linked VCs.
- By 17:45:50, the new tag has reached VC8, which is the farthest node in the ring topology, and sync is completed across all linked vCenter servers.

It took **1 minute and 40 seconds to complete the sync process** and bring all the linked mode vCenter servers to the same state in this case.

Time	VC1	VC2	VC3	VC4	VC5	VC6	VC7	VC8	VC9	VC10	VC11	VC12	VC13	VC14	VC15
17:44:00	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17:44:05	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17:44:10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17:44:15	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17:44:20	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17:44:25	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17:44:30	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17:44:35	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17:44:40	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1
17:44:45	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1
17:44:50	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1
17:44:55	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
17:45:00	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
17:45:05	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1
17:45:10	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1
17:45:15	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1
17:45:20	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1
17:45:25	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1
17:45:30	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
17:45:35	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
17:45:40	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
17:45:45	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
17:45:50	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
17:45:55	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
17:46:00	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 12: We create a single tag on VC1. This is the timeline showing how the newly created tag on VC1 was propagated across all linked vCenter servers. 0 means the tag wasn't visible at the specified time, while 1 indicates when the tag was visible.

We ran various workloads on VC1 as shown in figure 13 and measured how long it took for all linked vCenter servers in the ring topology to reach the same state. We observed that it took between 100 to 130 seconds in practice, while the theoretical worst-case limit was 210 seconds.

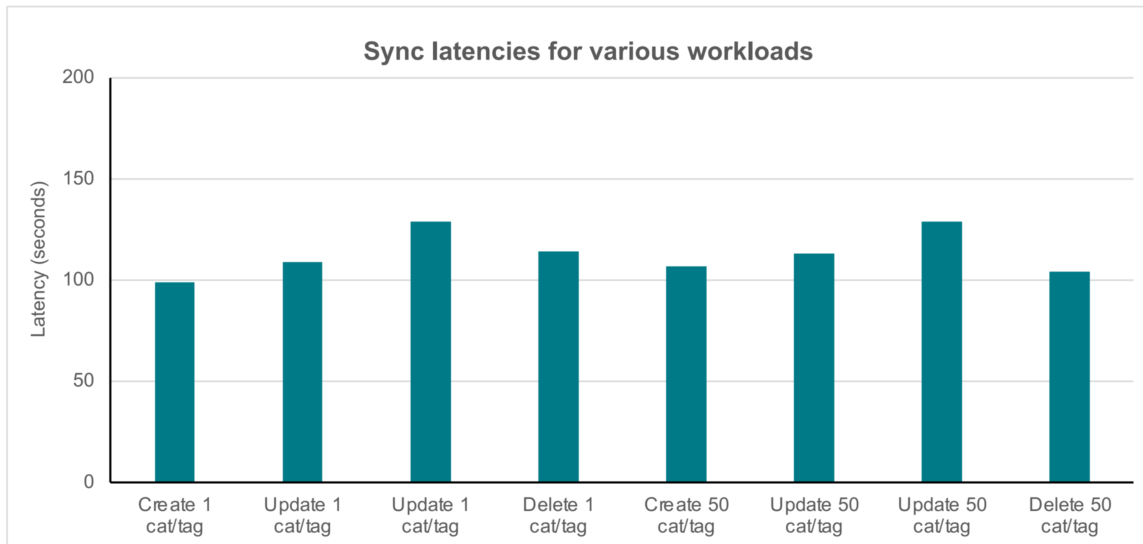


Figure 13: Sync completion time for all linked vCenter servers to reach the same state for various workloads

6.2 Comparison to standalone vCenter tagging performance

In embedded linked mode, tag and category definitions (for example, category **VM OS** or tag **Linux**) are global to all vCenter servers, while tag associations (for example, VM-1 is a **Linux** VM) are local to each vCenter server, since a given VM, host, datastore, or other entity can reside only on one vCenter server. Because the associations do not need to be replicated, most tagging operations issued to multiple vCenter servers simultaneously can proceed in parallel. Therefore, we expect the performance of tagging operations to match that of vCenter servers not in linked mode.

To illustrate this point, we evaluate two important operations:

- Attaching tags to objects.
- Listing tag associations.

We constructed a 15-node ring topology, as shown in figure 14. Each vCenter had 6,144 VMs, so for the 15 vCenter servers, we had a total of 92,160 VMs.

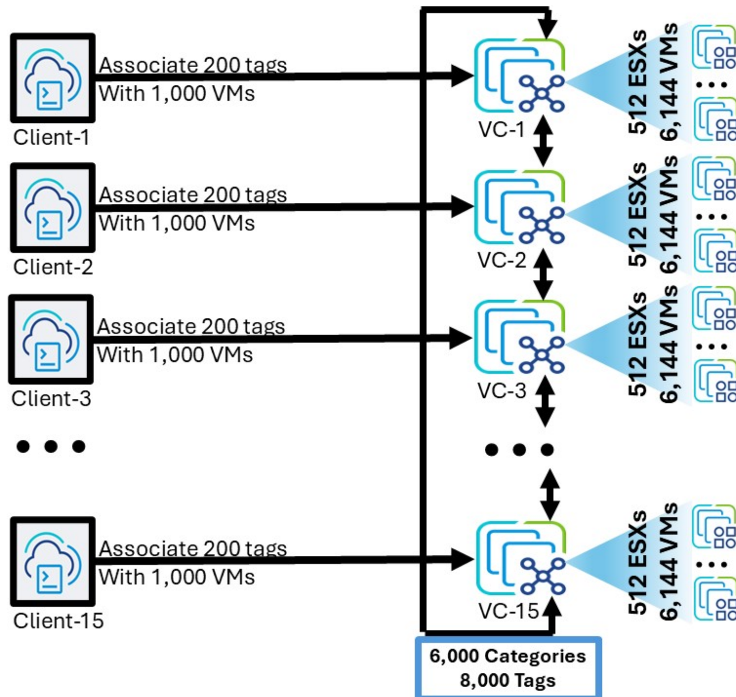


Figure 14: A total of 15 linked VCs. Each VC has 512 ESX servers and 6,144 VMs with a total of 6,000 categories and 8,000 tags. Each client runs a workload associating 200 tags with 1,000 VMs using API calls.

First, using the `AttachMultipleTagsToObject()` API, we associated 200 tags with 1,000 VMs. This means that on each vCenter server, we created **200 tags * 1,000 VMs = 200,000** associations for a total of 3,000,000 associations overall across the entire linked mode installation.

We also deployed a standalone vCenter server with 6,144 VMs, as shown in figure 15. We associated 200 tags with 1,000 VMs on the single vCenter server using the same API, which created **200 tags * 1,000 VMs = 200,000** associations.

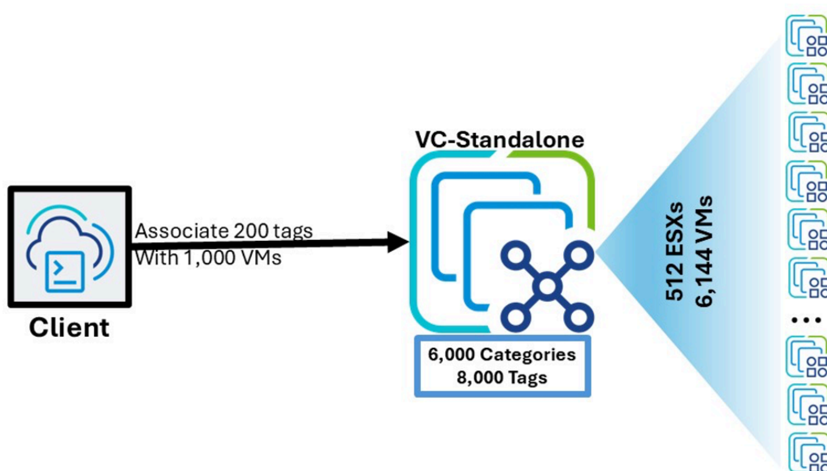


Figure 15: The standalone VC with 512 ESX servers and 6,144 VMs with a total of 6,000 categories and 8,000 tags. One client runs a workload associating 200 tags with 1,000 VMs using API calls.

As shown below in figure 16, for a standalone vCenter server, the time to attach 200 tags to 1,000 VMs is 43.3 seconds. When we run the same workload to attach 200 tags to 1,000 VMs against each of the vCenter servers in a 15-vCenter linked mode topology, with each vCenter server managing 6,144 VMs, we expect each vCenter server to complete in the same amount of time as the standalone vCenter server time, since there are no dependencies between vCenter servers, and they can run the tagging operations in parallel. As table 6 indicates, the overall end-to-end time for each vCenter server is indeed between 42.7 and 44.9 seconds, with an average of 44 seconds. In this case, with 15 vCenter servers, each with 200,000 associations, a 15-vCenter linked mode topology can handle up to 3,000,000 associations.

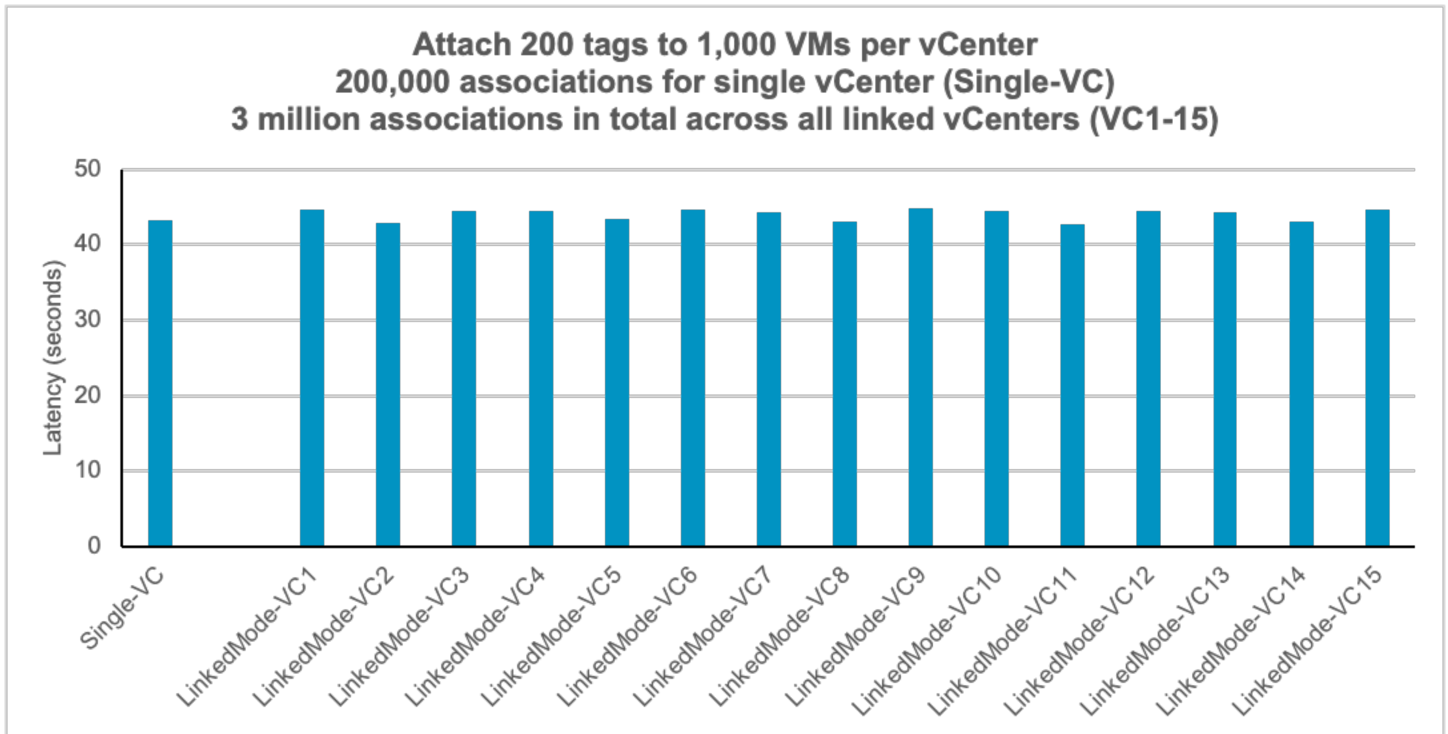


Figure 16: Latency to attach 200 tags to 1,000 VMs, standalone vCenter vs. vCenters in a linked mode configuration

Next, we look at the performance of listing associations comparing a standalone single vCenter server vs. linked vCenter servers. We deployed two testbeds:

- **Testbed 1:** Standalone single vCenter server with 120,000 tag associations.
- **Testbed 2:** Linked mode with 15 vCenter servers; each vCenter server with 120,000 tag associations.

The PowerCLI command `Get-TagAssignment` is the workload used to list all of the associations.

As expected, in figure 17, running the workload of listing 120,000 associations performs similarly on linked vCenter servers compared to a single vCenter server.

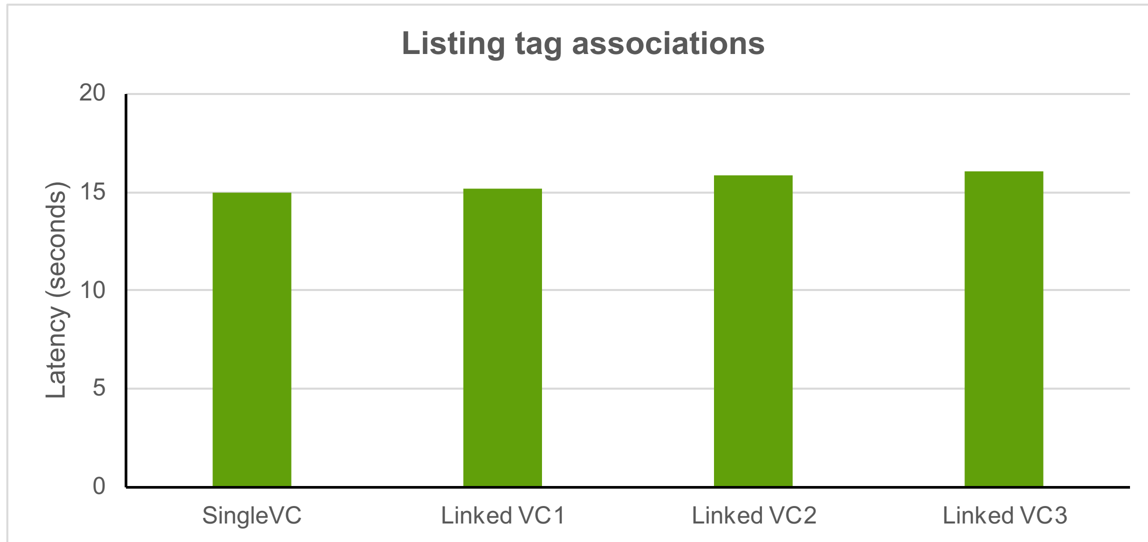


Figure 17: Time to list tag associations on a single vCenter server vs. vCenter servers in linked mode.

To summarize, for vCenter servers in linked mode, tagging-related operations like attaching objects to tags or listing tag associations takes the same amount of time as doing these operations against a standalone vCenter server that is not in linked mode, since tag associations are specific to each vCenter server and can therefore be done in parallel across all of the vCenter servers.

7 Tags vs. custom attributes

There are some situations in which tags might not be the right fit. For example, suppose you have 4,000 VMs in vSphere, and you want to assign an assetID to each VM. One option is to create a category called **assetID**, and then have each tag within that category be a different assetID. However, this approach would generate numerous tags, which can cause performance issues. Another example might be assigning a "last power-off time" to each VM. You could have a category named **last power-off time** and have each tag be a different time, but this can again cause scalability issues. For situations like these two cases, it would be better to use vSphere custom fields and attributes. A discussion of custom fields and attributes is beyond the scope of this paper; refer to the vSphere documentation topic What Are Custom Attributes in the vSphere Client? [7] or the article Custom Attributes != vSphere Tags [8].

8 Scale numbers for good performance

As mentioned in section 2 Introduction, there are no hardcoded limits for the number of categories, tags, or tag associations that the tagging service can support. However, as we have shown in this paper, the performance of the UI and APIs depends on the number of associations. In table 7, we list scale numbers that will work well out of the box for vCenter 8.0 U3. The limits are identical to those of vCenter 7.0: 150,000 tag associations per vCenter. For these numbers of tags, categories, and associations, and assuming proper hardware configuration of vCenter (see 11 Appendix), we expect out-of-the-box performance to be better than vCenter 7.0 for the UI, second- and third-party applications, and custom scripts. This performance is contingent upon the scripts adhering to our best practices, the system having the appropriate hardware, and the VCSA being properly sized (in our tests, we used

the x-large configuration). As previously stated, all nodes in a linked mode domain share tag and category definitions, while associations are exclusive to a single vCenter. Therefore, the associations below are for a single vCenter server in a linked mode domain, and the numbers of tag and category definitions are the cumulative number of definitions across all nodes in the linked mode domain. The number of associations scales with the number of nodes in linked mode, so if you have, say, 10 nodes in linked mode, you can support up to 1,500,000 total associations across those nodes, with no more than 150,000 per vCenter. In this scenario, each VCSA must be configured as x-large.

Target of Limit	Limit
Category definitions	6,000
Tag definitions	8,000
Tag Associations	150,000

Table 7: vSphere 8.0 U3 tag limits for acceptable out-of-the-box performance. The number of associations is for a single vCenter in a linked mode configuration.

The limits above are based on response latency as well as hardcoded limits within vSphere processes to prevent denial-of-service (DoS) attacks and out-of-memory exceptions for large response bodies. As mentioned above, the limits are suggested soft limits. With careful coding and proper VCSA sizing, as indicated in this paper, you can exceed these limits.

9 Conclusion

We provided information about the proper use of tagging APIs for performance and discussed the following considerations.

Coding tips:

- For faster tag or category creation time, consider using multiple threads.
- If possible, for better performance, create tags spread across many categories rather than putting them in a few categories.
- For faster assignment of tags to objects, consider using `attachMultipleTagsToObject()` or `attachTagToMultipleObjects()` instead of `attach()`.
- For faster listing of tag associations, consider `listAttachedObjectsOnTags()`, `listAttachedTagsOnObjects()`, or the pagination API `list()`, rather than `listAttachedObjects()` or `listAttachedTags()`. The first two calls allow you to specify a concrete list of tags or objects, and they allow chunking requests to avoid overwhelming the tagging service. The pagination API `list()` is especially helpful if you prefer not to manually chunk the objects on the client side. It is also easier from an error-handling perspective, as described in section 4.4 Query VMs associated with tags. Use a ring-based ELM topology for the best latency in replicating tags. For more details, refer to vCenter Server Architecture Part 1 – The Basics [9].

- Keep a local map of IDs (tag IDs, category IDs, VM_IDs) to names so that you don't have to make a call to vCenter or the tagging service every time you need an ID from a name.

Important caveats:

- For listing associations, if using `listAttachedTagsOnObjects()`, make sure you divide the list of objects into groups of 2,000 to avoid hitting response size limits.
- Tagging APIs can return a maximum response size of approximately 7MB. In practice, for 256 billion tags, this limit represents a response size of approximately 30,000 tag associations: this is, for example, 2,000 objects and 15 tags per object, or 200 objects and 150 tags per object, or 1,000 tags and 30 tags per object. When using `listAttachedTagsOnObjects()` or `listAttachedObjectsOnTags()` to return tag associations, be careful to avoid reaching this limit by chunking the request sizes by tags or by objects. If you surpass this limit, you'll see the error shown in section 4.4 Query VMs associated with tags.
- Note that the pagination API `list()` automatically breaks down responses to avoid this limit. The creation and deletion of tags might generate tombstones within the tagging service database. Some of these are fine, but an excessive number of them can negatively impact performance. Refer to Broadcom KB 318924 [10].
- There are internal vCenter limits to avoid DoS attacks. The vCenter appliance allows no more than 1,500 requests per second to tagging and other RESTful vAPI-based services within the VCSA (for example, the content library). If you exceed this limit in your environment, we recommend inserting a pause periodically to slow down the rate of requests. To know if you are hitting this limit, please check the vAPI endpoint log file in `/var/log/vmware/vapi/endpoint/endpoint.log` for errors like `Request rejected due to high request rate. Try again later.`
- In certain rare cases, you may need to increase the heap size of the vAPI endpoint service to accommodate high amounts of tagging traffic. When the vAPI endpoint approaches its heap size, it triggers an alert within vCenter, so please be careful to check for this alert periodically.

The latency of performing a tag association depends on the number of pre-existing associations. If many associations exist, it may be necessary to perform maintenance on the tag association table, which is a table within the main vCenter database. To perform such cleanup, run the following vacuum job to clean up the database engine statistics:

1. Connect to your VCSA VM via SSH as the root user.
2. Log into Postgres:

```
/opt/vmware/vpostgres/current/bin/psql -U postgres -d VCDB
```

3. Run vacuum analyze in this order:

```
vacuum analyze cis_kv_keyvalue;

vacuum analyze cis_kv_providers;
```

10 References

[1] Broadcom, "VMware vSphere Automation SDKs Programming Guide: VMware vSphere 8.0," 2024. https://vdc-download.vmware.com/vmwb-repository/dcr-public/7d0e2590-6239-4a65-b688-f6dcfd293eef/.

[2] R. Soundararajan and J. Zuk, "Writing Performant Tagging Code: Tips and Tricks for PowerCLI," 6 Jun 2019. https://blogs.vmware.com/performance/2019/06/writing-performant-tagging-code-tips-and-tricks-for-powercli.html.

[3] R. Angani, R. Soundararajan and M. Wiggers, "VMware vSphere 7.0 U2 Tagging Best Practices," 1 Mar 2023. https://www.vmware.com/docs/tagging-vsphere70u2-perf.

[4] Broadcom, "Developer Portal > Get Tag Id > vSphere 8.0.3.," 2024. https://developer.broadcom.com/xapis/vsphere-automation-api/latest/cis/api/cis/tagging/tag/tag_id/get/.

[5] Broadcom, "PowerCLI SDK," 2024. https://developer.broadcom.com/powercli.

[6] Broadcom, "vCenter Enhanced Linked Mode," 8 Jul 2022. https://docs.vmware.com/en/VMware-vSphere/8.0/vsphere-vcenter-installation/GUID-4394EA1C-0800-4A6A-ADBF-D35C41868C53.html.

[7] Broadcom, "Enhanced Linked Mode Design for the Management Domain: Recommended ring topology," 28 Mar 2023. https://docs.vmware.com/en/VMware-Cloud-Foundation/4.5/vcf-management-domain-design/GUID-21FF411C-3996-4614-A574-3C6BA29364D9.html.

[8] Broadcom, "What Are Custom Attributes in the vSphere Client," 27 Jul 2023. https://docs.vmware.com/en/VMware-vSphere/8.0/vsphere-vcenter-esxi-management/GUID-73606C4C-763C-4E27-A1DA-032E4C46219D.html.

[9] W. Lam, "Custom Attributes != vSphere Tags," 13 Jan 2015. https://www.virtuallyghetto.com/2015/01/custom-attributes-vsphere-tags.html.

[10] E. Younis, "vCenter Server Architecture Part 1 - The Basics," 7 May 2018. https://emadyounis.com/vcenter-server-architecture-part-1-the-basics/.

[11] Broadcom, "Troubleshooting and addressing accumulation of tombstones in a platform services controller (318924)," 31 Oct 2024. https://knowledge.broadcom.com/external/article?articleNumber=318924.

[12] Broadcom, "VMware vSphere Automation SDK for Java," 18 Jul 2024. https://github.com/vmware/vsphere-automation-sdk-java.

[13] Broadcom, "Managed Object References," 31 May 2019. https://docs.vmware.com/en/VMware-vSphere/8.0/vsphere-vddk-programming-guide/GUID-ABA1EC0F-12C8-4149-8752-30B008C820BC.html.

[14] J. L. Hennessy and D. A. Patterson, "Chapter One: Fundamentals of Quantitative Design and Analysis," in *Computer Architecture: A quantitative approach*, Waltham, MA: Morgan Kaufmann, 2012, p. 46–48. ISBN: 978-0-12-383872-8.

11 Appendix

11.1 Testbed setup and software versions

We used Java 1.8 (1.8.0_231).

We used vCenter 8.0 U3 and 7.0 U3 running on vSphere 8.0. A proprietary internal tool simulated 2,048 hosts and 36,864 VMs in our inventory. Because of the large number of VMs, we configured the VCSA as x-large.

11.2 Java code examples

Some of these examples come from the following Github repository:
https://github.com/vmware/vsphere-automation-sdk-java [9]

For each of the Java examples below, we needed a set of imports. Here are the imports we used in our code snippets:

```
import vim25
import com.vmware.cis.tagging.Category;
import com.vmware.cis.tagging.CategoryTypes;
import com.vmware.cis.tagging.Tag;
import com.vmware.cis.tagging.TagAssociation;
import com.vmware.cis.tagging.TagModel;
import com.vmware.cis.tagging.TagTypes;
import com.vmware.cis.tagging.CategoryModel.Cardinality;
```

11.2.1 Example J0: VM DynamicID and tag TagID

Many code samples involve the use of VM DynamicID objects. A DynamicID object is similar to a managed object reference (MoRef, described in [12]), but the identifier and type are stored in separate fields. Here is a simple code snippet for constructing a DynamicID from a VM MoRef. We first retrieve the VM by name, and then we get its MoRef. Finally, we construct a DynamicID from this MoRef.

```
// Retrieve the VM MoRef from its name
this.VMMoRef = VimUtil.getVM(this.vimAuthHelper.getVimPort(),
                             this.vimAuthHelper.getServiceContent(),
                             “VM_NAME”);
assert this.VMMoRef != null;
// convert the MoRef to vAPI DynamicID
this.vmDynamicId = new DynamicID(this.VMMoRef.getType(),this.VMMoRef.getValue());
```

11.2.2 Example J1: Retrieving category/tag IDs from category/tag names

Most tagging APIs require tag IDs. A tag ID is globally unique, while a tag name is unique within a category. To find a tag ID given a tag name, you must also specify the category ID.

The following code finds a category ID given a category name:

```
Category _categoryService =
                   vapiAuthHelper.getStubFactory().createStub(Category.class,
                   sessionStubConfig);
// Get CategoryId
public String getCategoryId(String catName) {
   List<String> catIds = _categoryService.list();
   String retCatId = NULL;
   if (catIds.size() > 0) {
      for (String cat : catIds){
         if (_categoryService.get(cat).getName().equals(catName)) {
            retCatId = _categoryService.get(cat).getId();
            break;
         }
      }
   }
   return retCatId;
}
```

The following code gets a tag ID given a tag name and a category name, using the routine above to map the category name to an id.

```
Tag _taggingClient = vapiAuthHelper.getStubFactory().createStub(Tag.class,
                                                          sessionStubConfig);
// Get TagId for a given tagName and CategoryName
public String getTagId(String tagName, String catName) {
   List<String> tagIds = _tagProvider.list();
   String catId = getCategoryId(catName);
   String retTagId = "";
   if (tagIds.size() > 0) {
      for (String tag : tagIds) {
         if (_tagProvider.get(tag).getName().equals(tagName)
              &&_tagProvider.get(tag).getCategoryId().equals(catId)) {
            retTagId = _tagProvider.get(tag).getId();
            break;
         }
      }
   }
   return retTagId;
}
```

11.2.3 Example J2: Create a category

The following code shows how to create a category.

```
/**
 * Creates a tag category
 *
 */
private String createTagCategory(String name, String description,
                      Cardinality cardinality, Set<String> AssociableTypes) {
   CategoryTypes.CreateSpec createSpec = new CategoryTypes.CreateSpec();
   createSpec.setName(name);
   createSpec.setDescription(description);
   createSpec.setCardinality(cardinality);
   createSpec.setAssociableTypes(associableTypes);
   return this.categoryService.create(createSpec);
}
// Sample invocation: create category named “GuestOS”
Set<String> associableTypes = new HashSet<String>(); // empty hash set
associableTypes.add(“VirtualMachine”);
String categoryId = createTagCategory(“GuestOS”,
                                      “Guest OS”,
                                      CategoryModel.Cardinality.SINGLE,
                                      associableTypes);
```

11.2.4 Example J3: Create a tag

In this example, we create a tag using the categoryID from the previous example.

```
/**
 * Creates a tag
 *
 */
private String createTag(String name, String description,
                         String categoryId) {
   TagTypes.CreateSpec spec = new TagTypes.CreateSpec();
   spec.setName(name);
   spec.setDescription(description);
   spec.setCategoryId(categoryId);
   return this.taggingService.create(spec);
}

// Sample invocation: create tag named "Windows 11" in categoryId from above
String tagID = createTag("Windows 11","DB Operating System",categoryId);
```

11.2.5 Example J4: Associating a tag with a VM (attach())

In this example, we associate a tag with a VM. using the VM DynamicID (vmDynamicId) from "11.2.1 Example J0: VM DynamicID and tag TagID" above and the tagID from the previous example.

```
/**
 * Associates tagId from Example J3 to VM whose DynamicID is vmDynamicId
   (Example J0)
 * We assume vmDynamicId is retrieved as shown in Example J0 above.
 *
 */

// Get a handle to the tagAssociation Methods
tagAssociation =
             vapiAuthHelper.getStubFactory().createStub(TagAssociation.class,
             sessionStubConfig);

tagAssociation.attach(tagId,vmDynamicId);
```

11.2.6 Example J5: Associating a tag with multiple VMs (attachTagToMultipleObjects())

The following code shows attaching 1 tag to 1,000 objects.

```
// Attach 1 tag to first 1,000 VMs
// Assume we have a list of VM DynamicIDs List<DynamicID> vmDynamicIds

int totalVMs = 1000;
List<DynamicID> vmDynamicIdsNeeded = new LinkedList<>();
for (int k = 0; k < totalVMs; k++) {
   vmDynamicIdsNeeded.add(vmDynamicIdsNeeded.add(vmDynamicIds.get(k));
}

// Get a handle to the tagAssociation Methods
tagAssociation =
             vapiAuthHelper.getStubFactory().createStub(TagAssociation.class,
             sessionStubConfig);
tagAssociation.attachTagToMultipleObjects(tagId,vmDynamicIdsNeeded);
```

11.2.7 Example J6: Associating multiple tags with multiple VMs (attachMultipleTagsToObject())

The following code shows attaching 10 tags to 1,000 objects.

```
// Attach 10 tags to 1,000 VMs
// Assumes we have a list of all tagIDs List<String> tagIds
// Assumes we have a list of all VM_IDs List<DynamicId> vmDynamicIds
// We will assign the first 10 of these tags to 1,000 VMs

int totalVMs = 1000;
int totalTags = 10;

// Get first 10 tags from the total tagIds
List<String> associableTags = new LinkedList<>();
for (int k = 0; k < totalTags; k++) {
   associableTags.add(tagIds.get(k));
}

// Get first 1,000 VMs from the total VM_IDs
List<DynamicID> vmDynamicIdsNeeded = new LinkedList<>();
   for (int k = 0; k < totalVMs; k++) {
vmDynamicIdsNeeded.add(vmDynamicIds.get(k));
}

tagAssociation =
             vapiAuthHelper.getStubFactory().createStub(TagAssociation.class,
             sessionStubConfig);
for (int i = 0; i < totalVMs; i++) {
   tagAssociation.attachMultipleTagsToObject(vmDynamicIdsNeeded.get(i),TagIds);
}
```

11.2.8 Example J7: List tags associated with a VM

```
// List tags associated with vm vmDynamicId from above.
// Assumes tagAssociation object from above
List<String> retTagIds = tagAssociation.listAttachedTags(vmDynamicId);
```

11.2.9 Example J8: List VMs associated with a tag

```
// List tags associated with tagId from above
// Assumes tagAssociation object from above
List<DynamicID> retDynamicIds = tagAssociation.listAttachedObjects(tagId);
```

11.2.10 Example J9: List tags associated with a group of VMs

```
// Helper object for storing results
ObjectToTags {
   DynamicID objectId;
   List<String> tagId;
}

// List tags associated with 10 VMs from vmDynamicIdsUsed list above
// Assumes tagAssociation object from above
List<TagAssociationTypes.ObjectToTags> retObjectToTags =
   tagAssociation.listAttachedTagsOnObjects(vmDynamicIdsUsed.subList(0,10));
```

11.2.11 Example J10: List VMs associated with a group of tags

```
// Helper object for storing results
TagToObjects {
   String tagId;
   List<DynamicID> objectIds;
}

// Assumes we have a list of all tagIDs List<String> tagIds
List<TagAssociationTypes.TagToObjects> retTagToObjects =
   tagAssociation.listAttachedObjectsOnTags(tagIds);
```

11.2.12 Example J11: Retrieving tag information (for example, name and description) from the tag ID

```
// TagModel contains name, description, tag ID, category ID, and usedBy fields
TagModel retTagModel = _tagProvider.get(tagID);
```

11.2.13 Example J12: Retrieving all associations using the pagination API

Here is an example in which we retrieve all associations using the pagination API.

```
// Get all tag associations a page at a time.
// Construct the tag association IDs.
// Store these IDs in a set.
AssociationsTypes.ListResult page;
AssociationsTypes.IterationSpec iter = new AssociationsTypes.IterationSpec();
static Set<String> associationIds = new HashSet<String>();

iter.setMarker("");
do {
   page = associationsProvider.list(iter);
   iter.setMarker(page.getMarker());

   // Store association IDs for this page
   for (AssociationsTypes.Summary summary : page.getAssociations()) {
      String tagId = summary.getTag();
      String objectId = summary.getObject().getId();
      String associationId = tagId + objectId;
      associationIds.add(associationId);
   }
} while (page.getStatus() !=
                          AssociationsTypes.LastIterationStatus.END_OF_DATA);

// Print out the total number of associations
System.out.println("Total number of Tag Associations: " + associationIds.size());
```

11.3 PowerShell examples

In this section, we give select PowerShell examples. Examples of creating a category, creating a tag, `attach()`, `attach_tag_to_multiple_objects()`, and `attach_multiple_tags_to_object()` are given in the Performance Team blog on tagging [2] and are therefore not repeated here.

11.3.1 Example P0: Creating VMID objects and retrieving tag IDs

(See next page.)

```
# Retrieving a tag ID from its name
function Get-TagIdFromName {
   Param ($a)
   $allTagMethodSvc = Get-CisService com.vmware.cis.tagging.tag
   $allTags = @()
   $allTags = $allTagMethodSvc.list()
   foreach ($tag in $allTags) {
      if (($allTagMethodSvc.get($tag.value)).name -eq $a) {
         return $tag.value
      }
   }
   return $null
}

# Creating a single VM ID Object
$testVM = Get-VM -Name "myVM"

$VMInfo = $testVM.ExtensionData.MoRef
$vmid = New-Object PSObject -Property @{
   id = $VMInfo.value
   type = $VMInfo.Type
}

# Create an array of VM IDs
$allVMs = Get-VM
$useTheseVMIDs = @()

# Create VM objects for all VMs.
# This should be done once for all VMs, not every time
# you want to do an association.
for ($i = 0; $i -lt 1000; $i++) {
   $VMInfo = $allVMs[$i].extensiondata.moref
   $useTheseVMIDs += New-Object PSObject -Property @{
      id = $VMInfo.value
      type = $VMInfo.type
   }
}
```

11.3.2 Example P1: List tags associated with a VM

```
$allCategoryMethodSVC = Get-CisService com.vmware.cis.tagging.category
$alltagMethodSVC = Get-CisService com.vmware.cis.tagging.tag
$allTagAssociationMethodSVC = Get-CisService
                                       com.vmware.cis.tagging.tag_association
# Use first VM from above list
$useThisVMID = $useTheseVMIDs[0]
$tagList = $allTagAssociationMethodSVC.list_attached_tags($useThisVMID)
```

11.3.3 Example P2: List VMs associated with a tag

```
$allTagMethodSvc = Get-CisService com.vmware.cis.tagging.tag

# Use method above to get tag ID from tag name
$tagId = Get-TagIdFromName “myTag”
$vmList = $allTagAssociationMethodSVC.list_attached_objects($tagId)
```

11.3.4 Example P3: List tags associated with a group of VMs

```
$assocSVC = Get-CisService com.vmware.cis.tagging.tag_association

# Pass in VMs in groups of 2,000
$sampleSize = 2000

# Assume a list of VMs: $useTheseVMIDs
$index = 0
$tagList = @()
for ($i = 0; $i -le ($useTheseVMIDs.Count/$sampleSize); $i++) {
   $tagList +=
     $assocSVC.list_attached_tags_on_objects($useTheseVMIDS[$index..($index +
     $sampleSize)])
}
```

11.3.5 Example P4: List VMs associated with a group of tags

```
$assocSVC = Get-CisService com.vmware.cis.tagging.tag_associations


# Loop over 15 tags and get the result.
# If there are too many VMs and you pass in all 15 tags in one go, you may get a RESPONSE error
$tagIdList = $tagList[0 .. 15]
$VMAssocList = @()
foreach ($tag in $tagIdList){
   # Must use a list because list_attached_objects_on_tags assumes a list of
   # tags as an input
   # We are doing this one at a time, but you can use multiple at a time as
   # long
   # as you do not get a RESPONSE error
   $useTheseTags = @()
   $useTheseTags += $tag
   $VMAssocList += $assocSVC.list_attached_objects_on_tags($useTheseTags)
}
```

11.3.6 Example P5: Retrieve associations using the pagination API

```
# Get a connection to the new tagging API (not: com.vmware.vcenter, not
# com.vmware.cis)
$assocSVC = Get-CisService com.vmware.vcenter.tagging.associations

# Create an object to help us iterate through the list of associations
$iterator = $assocSVC.Help.list.iterate.Create()

# Loop over pages until we have received all associations
# When the result status is END_OF_DATA, we have received all pages
do {
   $result = $assocSVC.list($iterator)

   # display associations
   $result.associations

   # set the iteration marker to the next page
   $iterator.marker = $result.marker
} while ($result.status -ne “END_OF_DATA”)
```

About the authors

Ian Ku is a software engineer in Broadcom's VMware Cloud Foundation (VCF) Performance team. His main areas of focus are vSphere scalability and performance. Ian received his BS and MS degrees from National Tsing Hua University in Taiwan and his PhD degree from UCLA, all in Computer Science.

Alper Mizrak is a software engineer in Broadcom's VMware Cloud Foundation (VCF) Performance team. His focus is on the performance and scalability of VMware products. Alper earned his PhD in computer science from UC San Diego in 2007, specializing in network security and distributed systems. His graduate research earned the prestigious William C. Carter Award and resulted in a book, *Secure Networking: Detecting Malicious Routers* (2008), along with several academic publications and technical reports.

Ravi Soundararajan is a distinguished engineer in the VMware Cloud Foundation (VCF) Performance Engineering group at Broadcom. His focus is vCenter performance and scalability—from the UI, to the server, to the database, to the hypervisor management agents. He has been at VMware/Broadcom since 2003, and he has presented on the topic of vCenter Performance at VMworld from 2013-2018. Ravi received his SB from MIT and his MS and PhD degrees from Stanford, all in Electrical Engineering. His X handle is @vCenterPerfGuy.

Acknowledgments

We gratefully acknowledge Raju Angani and Maarten Wiggers, who authored the previous versions of this paper. We also thank Dinesh Suresh, Xuwen Yu, Chung-Yen Chang, Joseph Zuk, Emil John, Mahesh Unnikrishnan, and Hristo Maradzhiev for their assistance with the content of this paper.

vmware by Broadcom

Copyright © 2025 Broadcom. All rights reserved.
The term "Broadcom" refers to Broadcom Inc. and/or its subsidiaries. For more information, go to www.broadcom.com. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies. Broadcom reserves the right to make changes without further notice to any products or data herein to improve reliability, function, or design. Information furnished by Broadcom is believed to be accurate and reliable. However, Broadcom does not assume any liability arising out of the application or use of this information, nor the application or use of any product or circuit described herein, neither does it convey any license under its patent rights nor the rights of others.