



VMware vSphere 7.0 U1 Tagging Best Practices

Performance Study - January 7, 2021



VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com
Copyright © 2020 VMware, Inc. All rights reserved. This product is protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at <http://www.vmware.com/go/patents>. VMware is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

Table of Contents

1	Executive Summary.....	3
2	Introduction.....	3
3	Terminology and Scope of Paper	4
4	Tagging APIs: Java.....	5
5	Tagging APIs: PowerShell.....	23
6	Linked Mode.....	24
7	Tags vs. Custom Attributes	25
8	Scale Numbers for Good Performance.....	26
9	Concluding Remarks	27
10	References	29
11	Appendix.....	32

1 Executive Summary

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

This paper is an updated version of the tagging whitepaper written for the vSphere 6.7 release. For 7.0 U1, we have seen performance improvements in various APIs vs. 6.7. We will highlight these changes throughout the paper, but briefly:

- Performance for tag and category creation is improved. (Section 4.1, [Create tag](#))
- Performance for list APIs is improved. (Section 4.3.1, [Query VMs associated with tags](#))
- Additional list API options are available to simplify retrieving associations. (Section 4.3.1, [Query VMs associated with tags](#))
- To accommodate the performance improvements in the list APIs, some of the `attach()` APIs perform some caching. This cache leads to a small degradation in performance at `attach()` time, but improves performance for creating and listing tags, as indicated above. (Section 4.2.3, [Multiple tags to a single VM](#))

2 Introduction

vSphere 5.1 introduced an inventory tagging feature that has been available in all later versions of vSphere, including vSphere 7.0 U1. Tags let datacenter administrators organize different vSphere objects like datastores, virtual machines, hosts, and so on. This makes it easier to sort and search for objects that share a tag, among other things.

In vSphere, tags are organized into categories. A category is typically used 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 2016.” If the category is “application name,” the tags might be “DB,” “Middleware,” or “vCenter Server.” If the category is “CPU type,” the tag values might be “AMD,” “Intel Broadwell,” or “Intel Cascade Lake.”

Writing code to use tags can be challenging in large-scale environments. In this whitepaper, we discuss the scalability limits of tags, and we give some tips and tricks for writing performant tagging code in either Java or PowerShell. While the performance results in this paper are specific to the hardware configuration and vCenter Server Appliance (VCSA) configuration in our lab, the basic trends will be the same as in customer environments.

We first discuss the scalability limits for tags in vSphere 7.0 U1. We then provide best practices for writing code for tagging.

3 Terminology and Scope of Paper

Before describing the scale limits for tagging, we first define some terms:

- “Category definitions” refers to the number of tag categories, for example, “OS type” or “application name.”
- “Tag definitions” refers to the number of tag definitions in the system, for example, “Windows” from category “OS type” or “SQL Server” under category “application name.”
- “Tag associations” refers to the mapping between a tag and an object (like a virtual machine or datastore). For example, if tag “Windows” is applied to VM “vm-111,” then that is one association, and if tag “Windows” and tag “SQL Server” are both applied to “vm-111,” then that is two associations. If a user associated 25 tags (“Windows,” “SQL Server,” “Pasadena Datacenter”, etc.) with 1,000 VMs, that would be 25,000 tag associations.

There are no *hard* limits to the number of tags, categories, and tag associations. However, increasing numbers of tags, categories, 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 will focus on Java- and PowerCLI-based programs, although the same principles apply to other languages like Python. Moreover, we will focus primarily on three operations:

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

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

4 Tagging APIs: Java

We will first discuss tag creation. In vSphere, a tag can only be created as part of an existing category. Thus, a category must be created before a tag can be added 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 a simple example of creating a category using Java, please see “[Example J2: Create Category](#)” in the [appendix](#).

The Java API for vSphere 7.0 U1 is documented [here](#) [5]. The full documentation set is [here](#) [6].

4.1 Create tag (assuming category is already created)

Sample code for creating a category and creating a tag within that category is given in the [appendix](#) (please see “[Example J2: Create Category](#)” and “[Example J3: Create Tag](#)”).

Tag creation requires a name and a description. The list of associable types (that is, what types of objects, like VMs or hosts, can be associated with this tag is optional): if it is not supplied, then a tag can be associated with any object. The category for a tag must specify the cardinality of the category. A description of cardinality is given in the performance team [tagging blog](#) [1], but briefly, *multiple cardinality* means that multiple tags (say, “Alice” and “Bob”) from a given category (say, “Owners”) can be assigned to a given object (like a VM). If the cardinality is one, then only one tag within a category may be applied to a given entity. For example, if the category is OS, then only one tag from that category (say, Ubuntu Linux or Windows Server 2016) may be applied to the VM.

The tag creation API allows a user to specify a single tag to be created—in this API, only one tag can be created at a time. The time to create a tag varies with the number of tags per category. For example, creating many tags under a single category is slower than creating a smaller number of tags in multiple categories.

Figure 1 shows the 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,” and so on. Creating 8,000 tags under 1 category increases slightly from the first tag to the last tag and takes on average about 9ms. This is a performance improvement over 6.7, which showed a much more dramatic linear increase in latency as the number of tags was increased. This improvement is due on part to caching of tag information and database query optimizations.

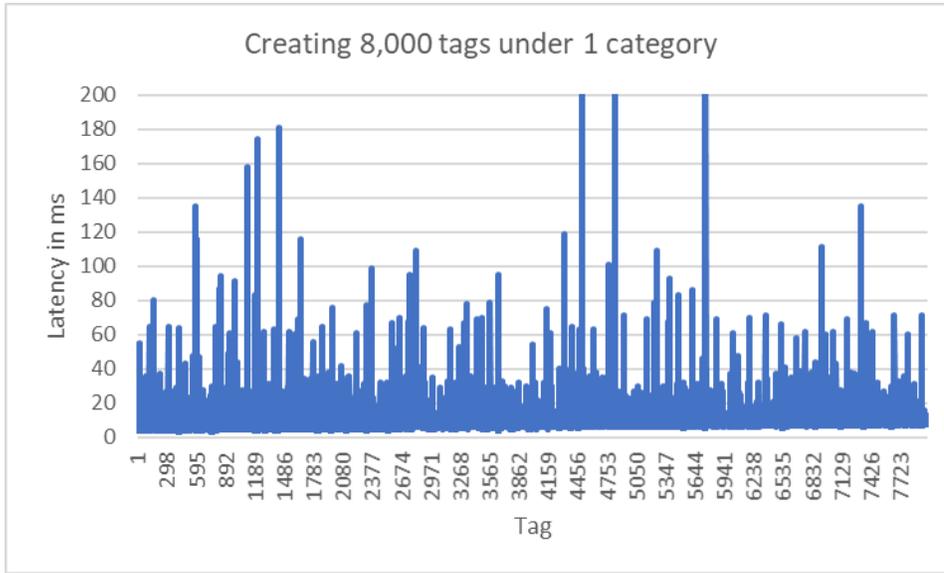


Figure 1: Creating 8,000 tags under 1 category. In 7.0 U1, the time to create a tag stays roughly constant with the number of tags.

Figure 2 shows the latency to create 8,000 tag definitions under 200 categories. In each case, there are 40 tag definitions per category. As the graph indicates, the latency to create tags is roughly constant: the time to create the 40 tags per category is around 8 milliseconds per tag. This number is on par with creating all tags within a single category. As we had described in our [6.7 whitepaper](#) [7], one option to speed up tag creation is to use multiple clients to create tags concurrently. In our labs, we have tested up to 8 clients with promising speedups: up to 7x for 8 clients. [5] Although performance in 7.0 U1 is similar with one category vs. many categories, multiple categories allow for better parallelism when using multiple clients. Thus, in 7.0 U1, as with 6.7, if practical, we recommend multiple categories with fewer tags per category vs. a small number of categories with many tags in them.

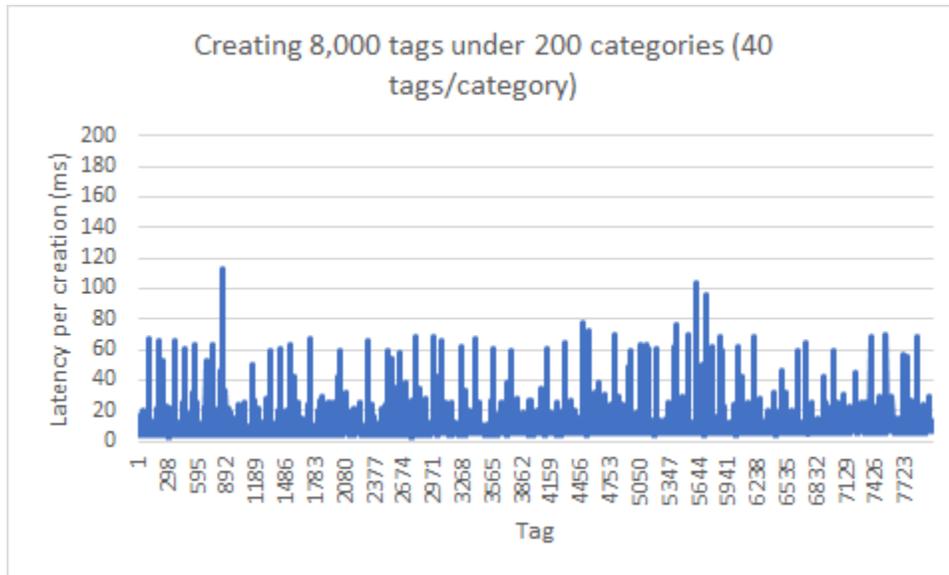


Figure 2: Creating 8,000 tags under 200 categories. Spreading tags among categories speeds up tag creation. This makes it roughly constant for small numbers of tags per category.

4.2 Associate tag(s) with VM(s)

There are 3 ways to attach a tag to a VM:

1. `attach(tagID, VM_ID)`
2. `attachTagToMultipleObjects(tagID, list of VM_IDs)`
3. `attachMultipleTagsToObject(list of tagIDs, VMID)`

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). Please see [Example J4: Associating a tag with a VM \(attach\(\)\)](#) in the [appendix](#), where we assign a tag “Windows 10” from category “GuestOS” to a VM. We recommend keeping a map of tag ID to tag name and VM ID to VM name, if possible, in order to avoid having to call vCenter or the tagging service each time such a mapping is needed. Pseudocode for using `attach()` is given here:

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

Attaching a single tag to a single VM typically takes around 25 milliseconds. Unlike in 6.7, in which the time varies as the number of tag associations increases, with 7.0 U1, the time is roughly constant. Figure 3 shows the latency to attach a tag to a VM as we increase the number of VMs from 1 to 10,000. The latency varies, but most of the `attach()` calls take around 25 milliseconds. The total end-to-end time is 4.2 minutes.

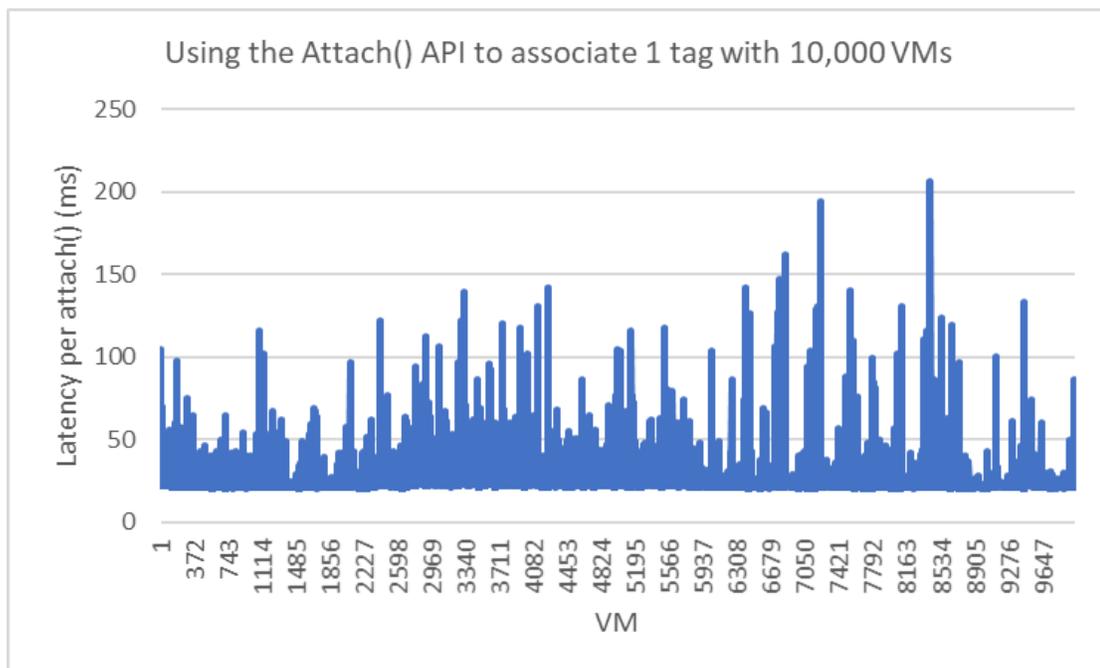


Figure 3: Using `attach()` to associate 1 tag with each of 10,000 VMs. The time for an individual `attach()` is roughly constant even as we increase the number of VMs.

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 “[Example J5: Associating a tag with multiple VMs \(attachTagToMultipleObjects\(\)\)](#)” in the [appendix](#) for an example of attaching a single tag to multiple VMs using `attachTagToMultipleObjects()`.

Pseudocode is given here:

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

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()`. As noted in the caveat above, in the case of `attachTagToMultipleObjects()`, we attach the tag to 2,000 VMs at a time. In the case of `attach()`, we sequentially attach the tag to each VM. The numbers are slightly worse than in 6.7 U1 due to caching introduced after an association is created, but the creation and list APIs are faster overall, as we indicate in other sections of this paper. In fact, in 6.7, a 5ms sleep is required between iterations in order to avoid hitting a throttle within the vapi-endpoint process. In 7.0 U1, no such sleep is required. If we include the 5ms sleep in the end-to-end time for 6.7, then its time is the same as 7.0 U1. So even though the individual attach time in 7.0 U1 is slower than in 6.7, due to improvements in other parts of the system, we can match the throughput of 6.7.

Operation	Number of iterations	Latency for 1 tag, 20,000 VMs
<code>attach()</code>	20,000 (1 per VM)	482 seconds
<code>attachTagToMultipleObjects()</code>	10 (2,000 VMs at a time)	265 seconds

Table 1: `attach()` vs. `attachTagToMultipleObjects()`

4.2.2.1 *Chunking VMs in calls to `attachTagToMultipleObjects()`*

We suggest putting no more than 2,000 VMs per `attachTagToMultipleObjects()`. Exceeding this number may cause errors, since the tagging service has some limits regarding the number of items in a request. To see if this issue is occurring in your system, look for this sort of error in the log file for the vapi-endpoint service (that is, `/var/log/vmware/vapi/endpoint/endpoint.log`):

```
Received request (910,429 bytes) is bigger than the allowed request size (204,800 bytes) - request blocked!
```

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. We give an example of attaching multiple tags to VMs in the the [appendix](#) in “[Example J6: Associating multiple tags with multiple VMs \(attachMultipleTagsToObject\(\)\)](#)”. Here is 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)
```

For best performance, we recommend using no more than 2,000 tags in a single call.

Figure 4 shows the latency of attaching 1 through 200 tags to a single VM using `attachMultipleTagsToObject()`. The latency is nearly constant until around 50 tags, and then it starts to grow linearly. There is some run-to-run variability, which is why the result with 45 tags is slightly worse than with 50 tags. After 150 tags, the overhead stays fairly constant until 200 tags: caching keeps overheads low, so the incremental overhead of attaching more tags is small.

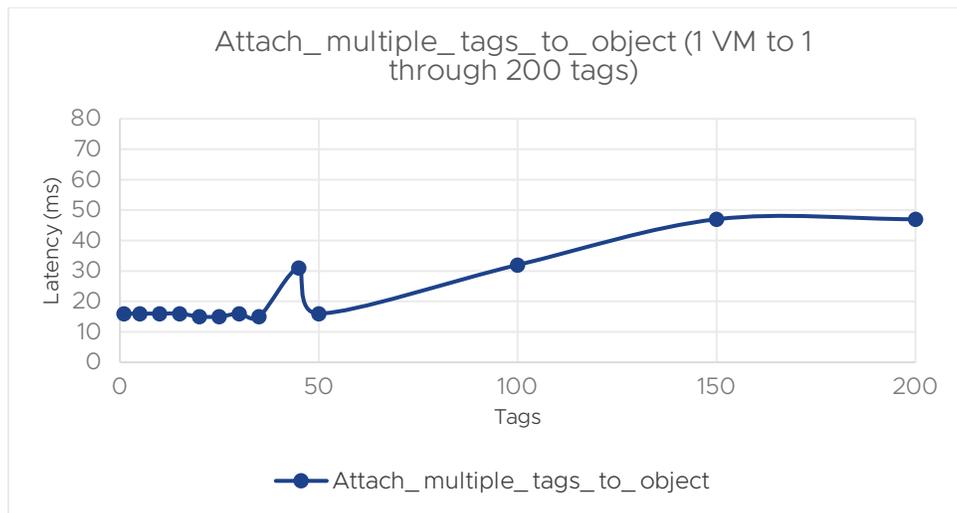


Figure 4: `attachMultipleTagsToObject()` for 1-200 tags and 1 VM. The latency is fairly steady until around 50 tags, when it starts to grow linearly.

In figure 5, we compare the latency of attaching multiple tags (1 up to 2,000) to a single VM using `attachMultipleTagsToObject()` vs. using serial `attach()` calls vs. `attachTagToMultipleObjects()`. As the figure shows, each method works well with a small number of tags, but `attachMultipleTagsToObject()` works best as the number of tags grows. For a small number of tags (< 50), the latency of `attachMultipleTagsToObject()` is around 200 ms. For 2,000 tags, the latency is around 500 ms. The `attach()` and `attachTagToMultipleObjects()` calls are both slightly slower per-iteration in 7.0 U1 vs. 6.7, as described earlier. When attaching 2,000 tags to 1 VM, both `attach()` and `attachTagToMultipleObjects()` must be called 2,000 times, paying this extra cost on each iteration. In contrast, `attachMultipleTagsToObject()` is significantly faster in 7.0 U1 vs. 6.7, primarily due to caching tag information in vCenter, and it is the preferred API for this use case.

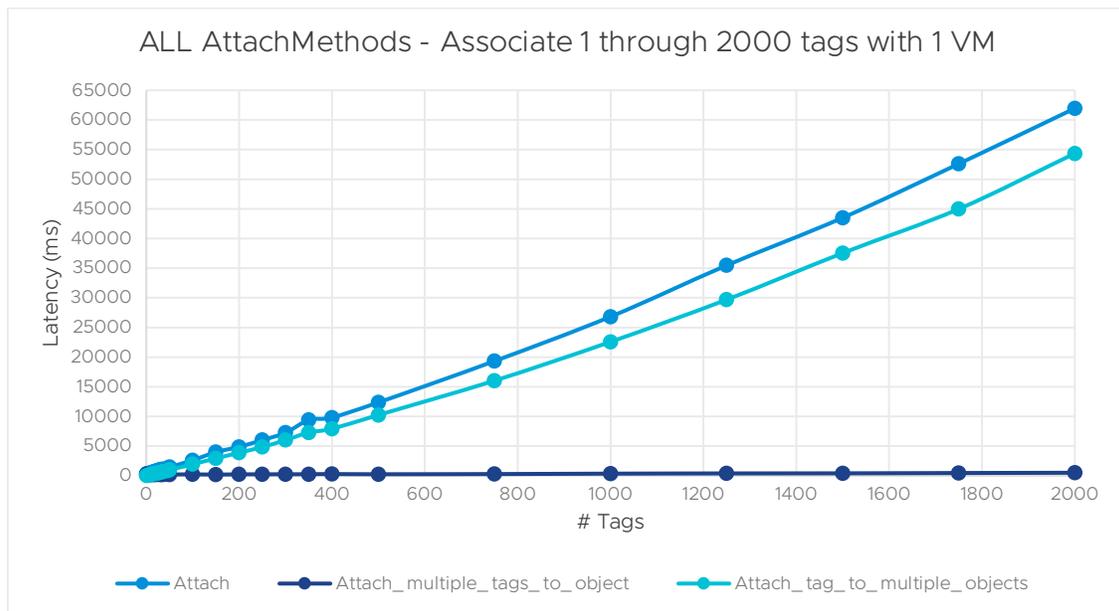


Figure 5: Attaching 1 through 2,000 tags to 1 VM. `attachMultipleTagsToObject()` is well-suited for this use case.

4.2.4 Multiple tags to multiple VMs

The previous cases considered assigning one tag to multiple VMs or multiple tags to one VM. In this section, we discuss assigning multiple tags to multiple VMs.

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

Suppose we had a group of 10 tags (for example, “Windows 2019 Server,” “DB app,” etc.), and we wanted 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 the [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.

API Call	Number of iterations	End-to-end latency	Time per iteration
attachTagToMultipleObjects()	10	136 seconds	14 seconds
attachMultipleTagsToObject()	1,000	19 seconds	19 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()` executes 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.

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. To conform to the limit of 2,000 objects per call, for this example, we divided the 5,000 VMs into 2 batches of 2,000 VMs and 1 batch of 1,000 VMs. We then looped through the tags and attached the tag to each batch. Here is some pseudocode to illustrate this:

```
// (not shown): Get tag IDs into List<String> tagIDs
// (not shown): Get VM IDs into List<DynamicId> VM_IDS
int chunkSize = 2000
int numChunks = ((VM_IDS.size()/2000) + 1)
for (String tagID : tagIDs) {
    for (int j = 0; j < numChunks; j++) {
        int index = j * chunkSize
        // Make sure we don't exceed the bounds of the VM ID list
        end = max(VM_IDS.size(), index+chunkSize);
        attachTagToMultipleObjects(tag, VM_IDS.sublist(index, end));
    }
}
```

In the graph below, the time taken to attach the first tag to all VMs is around 68 seconds, while the time to attach the 15th tag to all VMs is around 60 seconds. In contrast, in 6.7, the time to attach the 15th tag is around 200 seconds, nearly 3x the time taken by 7.0 U1. The end-to-end time to attach all 15 tags is the sum of the time for each iteration, around 1,023 seconds for 7.0 U1. In 6.7, the end-to-end time was around 3,000 seconds. This experiment creates 75,000 associations. It is important to note that this is above the out-of-the-box suggestion of 25,000 associations (see "[Scale Numbers for Good Performance](#)" in section 8), but we can exceed the out-of-the-box limits of the API by using batches of VMs. It is important to note that while the API works properly, the UI may be slow in this case.

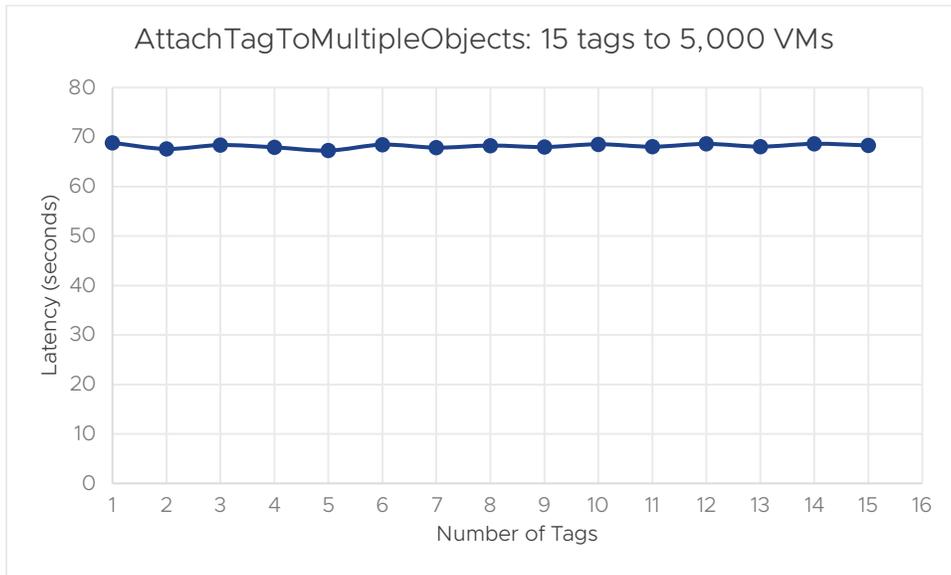


Figure 6: attachTagToMultipleObjects(): 15 tags to 5,000 VMs.

4.2.4.2.2 attachMultipleTagsToObject()

We can attach 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:

```
// (not shown) Get tagIDs into List<String> tagIDs
// (not shown) Get VM IDs into 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 the first few VMs is between 16 and 40 milliseconds, and the time to attach 15 tags to the five thousandth VM is also between 16 and 40 milliseconds. The total time to attach 15 tags to 5,000 VMs is the sum of the latency of each iteration, around 103 seconds.

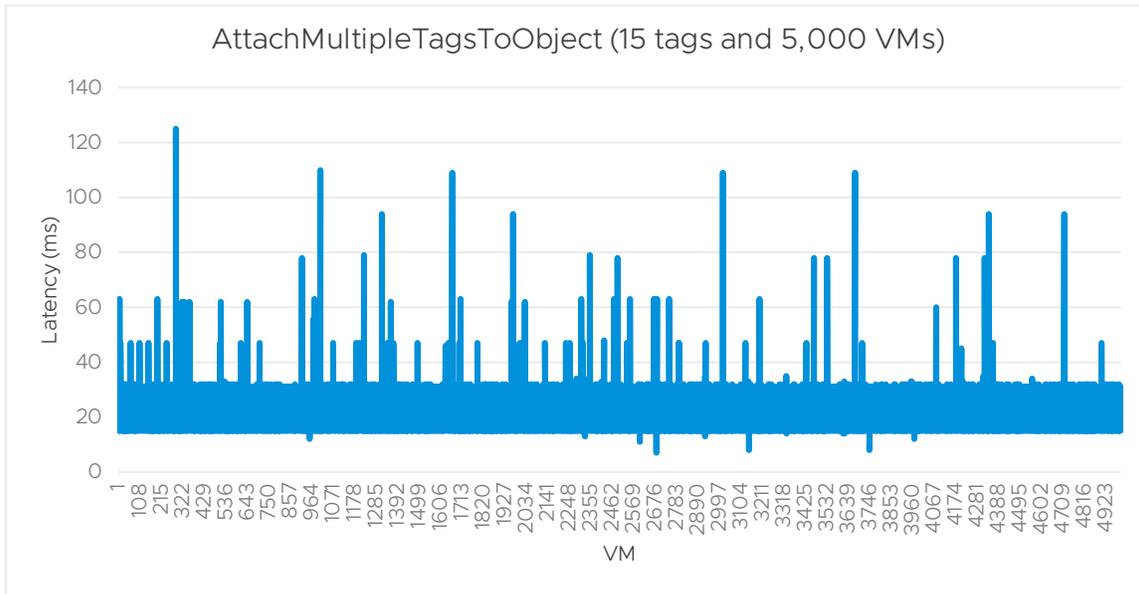


Figure 7: `attachMultipleTagsToObject()` . This call is roughly constant with the number of objects.

Caveat about number of objects or tags in a single call. As mentioned earlier, while it is faster to use these batched `attachTagToMultipleObjects()` and `attachMultipleTagsToObject()` calls, there is a limit to the size of a request that can be processed by the tagging service. When this limit is exceeded, the request cannot be processed. For `attachTagToMultipleObjects()`, to avoid hitting this limit, we recommend using no more than 2,000 VMs within a single call. For `attachMultipleTagsToObject()`, we recommend no more than 20 tags per call. If you want to associate a single tag with 20,000 VMs, you should do it using 10 iterations with 2,000 VMs per iteration.

In table 3, we summarize the performance of these API calls when attaching 15 tags to 5,000 VMs (75,000 tag associations):

API Call	Number of Iterations	End-to-End Latency
<code>attachTagToMultipleObjects()</code>	45 (15 tags * 3 batches)	17 minutes
<code>attachMultipleTagsToObject()</code>	5,000 (one per VM)	1.7 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 `attachMultipleTagsToObject()` . In addition, we observed that 7.0 U1 outperforms 6.7 (that is, the end-to-end latencies in 7.0 U1 are lower than in 6.7). As the table indicates,

`attachMultipleTagsToObject()` is faster with a large number of VMs, and overall, the APIs are nearly twice as fast as 6.7. Table 4 shows a few more data points: 10 tags associated with 1,000 VMs, and 1, 10, or 15 (hereafter referred to as {1,10,15}) tags associated with 5,000 VMs. For large numbers of VMs and more than one tag, `attachMultipleTagsToObject()` scales the best. With one tag and many VMs, `attachTagToMultipleObjects()` scales the best.

Operation	10 tags to 1000 VMs (10k associations)	1 tag to 5,000 VMs (5K associations)	10 tags to 5,000 VMs (50K associations)	15 tags to 5,000 VMs (75K associations)
<code>attach()</code>	3.5 minutes	1.8 minutes	18.1 minutes	27.1 minutes
<code>attachTagToMultipleObjects()</code>	2.2 minutes	1.1 minutes	11.3 minutes	17 minutes
<code>attachMultipleTagsToObject()</code>	0.31 minutes	1.5 minutes	1.6 minutes	1.6 minutes

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

The *takeaway* points from the table above are as follows:

1. The time to perform a single association increases as more tag associations are added.
2. Batching APIs (`attachTagToMultipleObjects()` Or `attachMultipleTagsToObject()`) are preferable to looping over every tag and every VM and using an `attach()` on each tag/VM pair.
3. When performing an operation on more than 2,000 VMs or 2,000 tags, break down tags or VMs into batches of less than 2,000. If you don't, you'll see an error such as this: `Hitting vapi endpoint limit during AttachTagToMultipleObjects()`.
4. As an additional consideration, to avoid denial-of-service attacks to the tagging service, there is a limit of 1500 tagging-related operations per second, increased from 360 in vSphere 6.7 U3. To detect if you are hitting this limit, look for messages like "Request rejected due to high request rate. Try again later." in the vapi-endpoint log file (`/var/log/vmware/vapi/endpoint/endpoint.log`). If your creation code is exceeding this limit, we recommend inserting a pause of 2ms or more between tag-creation calls.

4.3 Associate tag(s) with other objects

Associating tags with other objects (for example, hosts or 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 4 cases to consider:

1. To associate 1 tag with 1 object (for example, 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.

4.3.1 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 object id (as a DynamicID) and tag id. A DynamicID is an object that contains an identifier and a type. We give an example in the [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<TagAssociation.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, new in 7.0)** This call will list all the tag associations that are present in vCenter, provided a page at a time, with a maximum of “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 new pagination-based API is accessed via `https://{api-host}/api/vcenter/tagging/associations`, while the other APIs are accessed via `https://{api-host}/rest/com/vmware/cis/tagging`.

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 list it separately because its performance is much worse than the other approaches.

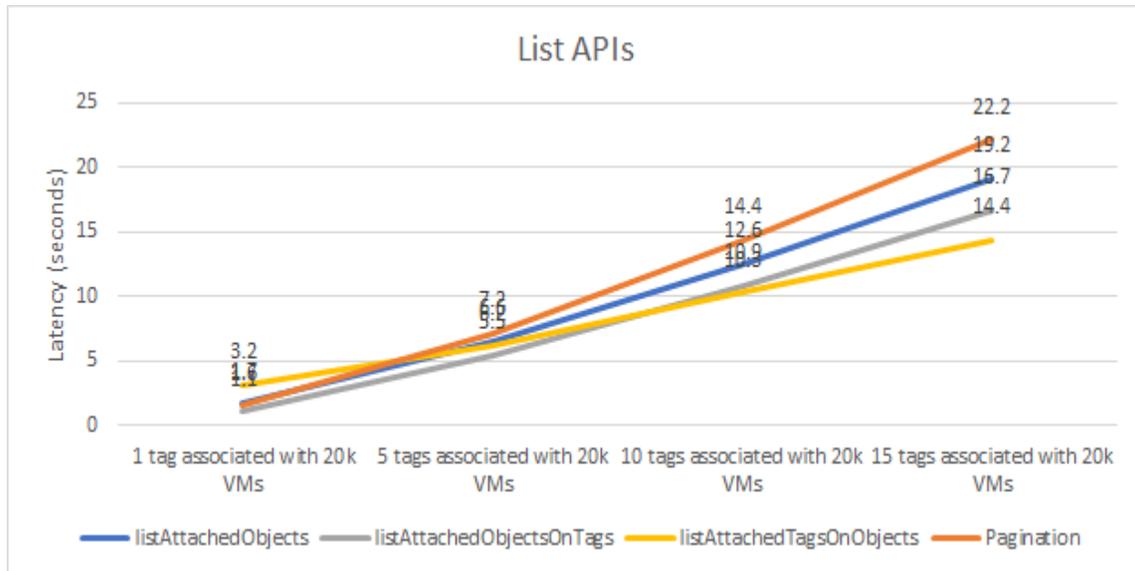


Figure 8: End-to-end time to retrieve associations for {1,5,10,15} tags attached to 20,000 VMs using various List APIs.

From figure 8 above, 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 1.7 seconds to 45 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 3 or more tags, the call failed due to internal buffer sizes. 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. Because of this, we do not have to retrieve the tags for all 20,000 VMs at once. Instead, we break down the list of 20,000 VMs into batches of 2,000 VMs. For 1 tag, we provide 1 tag and iterate over the VMs in batches 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, again for a total of 10 iterations to get associations for 20,000 VMs. This method scales the best, finishing more than 2x faster than the other approaches. The results are also faster than in 6.7.
- 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 batches of 5,000 per iteration). In this case there will be 10 iterations to get all 50,000 tag associations. Using this API, a client does not have to manually break down the objects into chunks.

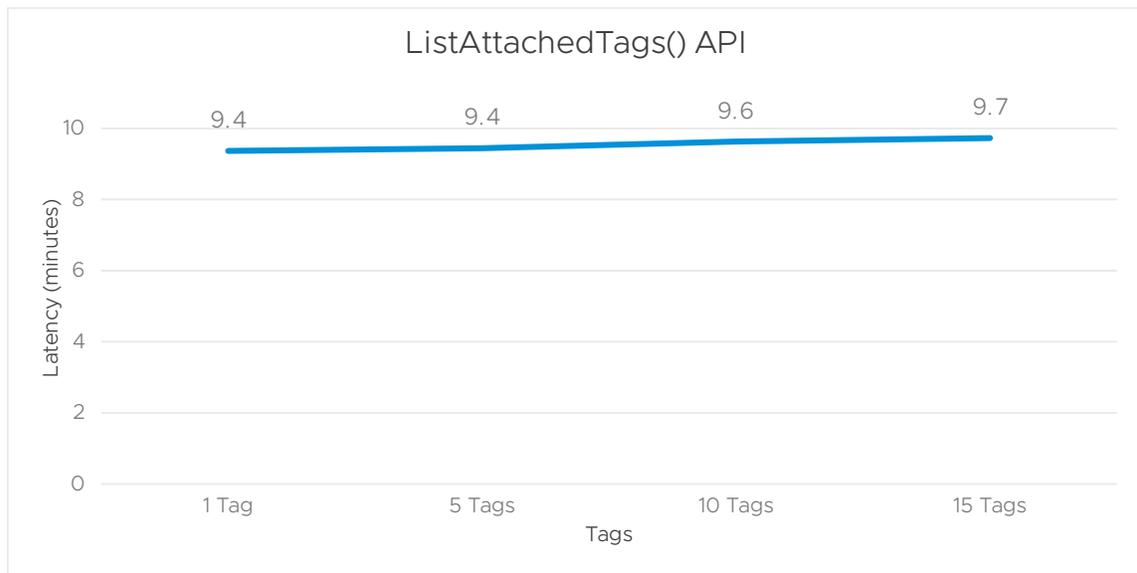


Figure 9: End-to-end time to retrieve associations for {1,5,10,15} tags attached to 20,000 VMs using `listAttachedTags()` API. `listAttachedTags()` is much slower than the other list calls, and we do not recommend using it in environments with more than 2,000 VMs.

From figure 9 above, we see that `listAttachedTags()` is much slower than the other approaches. This call only takes 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.

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
<code>listAttachedObjects</code>	1.7 seconds	6.6 seconds	12.6 seconds	19.2 seconds
<code>listAttachedObjectsOnTags</code>	1.1 seconds	5.5 seconds	10.9 seconds	16.7 seconds
<code>listAttachedTagsOnObjects</code>	3.2 seconds	6.2 seconds	10.3 seconds	14.4 seconds
<code>listAttachedTags</code>	9.4 minutes	9.4 minutes	9.6 minutes	9.7 minutes
<code>List (Pagination)</code>	1.6 seconds	7.2 seconds	14.4 seconds	22.2 seconds

Table 5: Latency for various list APIs.

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. One caveat is that when using `listAttachedTagsOnObjects()`, if any of the input objects has been deleted, the call will return errors for that object, so a client must be prepared to handle those errors. Alternatively, you can use pagination-based `list()` API, which also gives predictable performance, so that you do not have to manually break the object list into chunks of 2,000 entities. In addition, the pagination API will only return valid associations, rather than returning an error if an object is no longer present in the inventory.

It is important to point out that the list APIs above return IDs. Most client applications require names. To retrieve tag names from these IDs, you must use `get ()` calls. For example, `Tag.get ()` returns a `TagModel` object that has `name`, `id`, `category_id`, `description`, and `used_by` fields. For performance reasons, if it is practical, we recommend keeping a map of IDs to names on the client-side. Tag names are scoped to categories, so when storing a tag name, be sure to store the category name as well.

As a final example of batching, consider getting the mapping of all VMs associated with each tag. One way to do this is using `listAttachedObjects ()`:

```
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, we recommend using `listAttachedTagsOnObjects ()` instead. Because we do not know a priori 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:

```
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.

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

This example also appears in the [appendix: “Example J12: Retrieving 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());

```

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. `listAttachedTagsOnObjects()`
2. `Pagination list()`

The pagination `list()` API 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 `list()` API 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()` also offers predictable performance, but requires the user to properly chunk the list of objects into groups of less than 2,000 objects. 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](#) [1], there are typically two ways to write PowerShell scripts for vSphere 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 above.

The direct tagging service calls in PowerShell use the same vCenter server and tagging service resources as Java. Thus, the performance results are the same as the Java results above. In this section, we just focus on the difference in the API calls themselves. The API calls 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	<code>Attach()</code>	<code>Attach()</code>
Attach one tag to multiple objects	<code>attachTagToMultipleObjects()</code>	<code>Attach_tag_to_multiple_objects()</code>
Attach multiple tags to one object	<code>attachMultipleTagsToObject()</code>	<code>Attach_multiple_tags_to_object()</code>
List objects attached to a tag	<code>listAttachedObjectsOnTags()</code>	<code>List_attached_objects_on_tags()</code>
List associations (pagination)	<code>List()</code>	<code>List()</code> (instead of <code>com.vmware.cis</code> , please use <code>com.vmware.vcenter</code>)

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 the [appendix](#) for details.

PowerCLI version 11.5.0 is compatible with vCenter 6.0 through 7.0 U1. The documentation for the various versions of VMware PowerCLI is [here](#) [6].

6 Linked Mode

In [Enhanced Linked Mode \(ELM\)](#) [7], vCenter servers replicate tag and category information between peer nodes periodically. Each node 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, consider figure 10, which shows a [simple ring topology](#) [9] for vCenter servers.

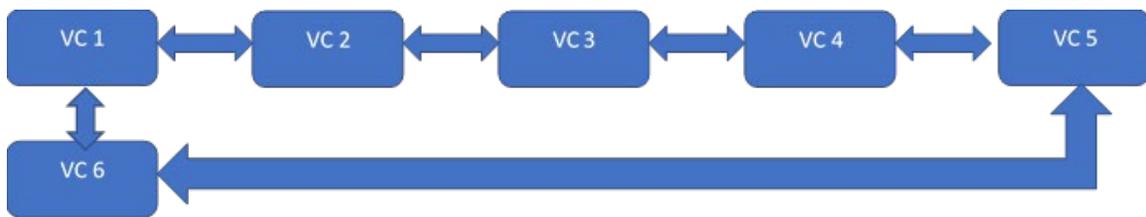


Figure 10: Ring topology for linked mode

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

- VC 1: VC 2 and VC 6
- VC 2: VC 1 and VC 3
- VC 3: VC 2 and VC 4
- VC 4: VC 3 and VC 5
- VC 5: VC 4 and VC 6
- VC 6: VC 1 and VC 5

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

1. Time 0: tag added to VC 1
2. Up to 30 seconds later, VC 1 replicates to VC 2 and VC 6
3. Up to 30 seconds later, VC 2 replicates to VC 3; VC 6 replicates to VC 5
4. Up to 30 seconds later, VC 3 replicates to VC 4; VC 5 also replicates to VC 4, but ELM performs conflict resolution to ensure only one new tag definition is created on VC 4.

As the timeline above suggests, in this 6-node ring topology, a tag may take up to 90 seconds to replicate throughout the system. Because each node pushes changes at 30-second intervals, this is the worst-case latency. The nodes do not push changes at the

same time, so in the best case, a tag change may occur right before that node has been scheduled to push changes. In this case, the changes would be reflected more quickly.

Different topologies will have different delays. For example, suppose VC 1 were NOT connected to VC 6, as shown below in figure 11.

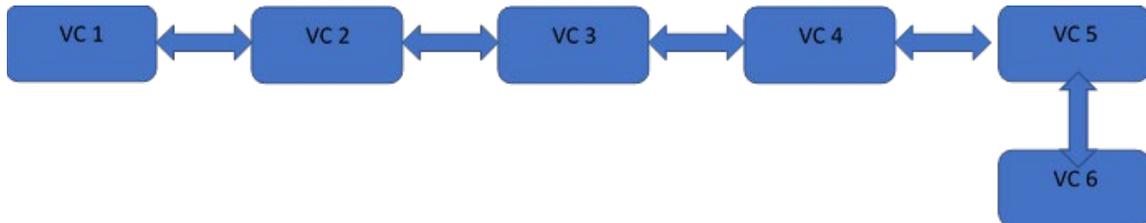


Figure 11: Linear topology for linked mode. We do not recommend this approach due to latency of tag/category replication.

1. Time 0: tag added to VC 1
2. Up to 30 seconds later: VC 1 replicates to VC 2
3. Up to 30 seconds later: VC 2 replicates to VC 3
4. Up to 30 seconds later: VC 3 replicates to VC 4
5. Up to 30 seconds later: VC 4 replicates to VC 5
6. Up to 30 seconds later: VC 5 replicates to VC 6.

As the timeline suggests, the worst-case latency for the last node to receive a tag update is $30 \text{ seconds} * 5 = 150 \text{ seconds}$ after the tag has been added. The best case is nearly instantaneous, if the changes occur right before a node is scheduled to push tag updates.

7 Tags vs. Custom Attributes

There are some situations in which tags might not be the right fit. For example, suppose a user has 4,000 VMs in their vSphere environment, and they wish 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 may cause performance issues. Another example might be assigning a “last power-off time” to each VM. One 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 might be better to use vSphere custom fields and custom attributes. A discussion of custom fields and custom attributes is beyond the scope of this paper: please refer to official [VMware documentation](#) [9] or [William Lam’s blog on this topic](#) [11].

8 Scale Numbers for Good Performance

As mentioned in the “[Introduction](#),” there are no hardcoded limits the number of categories, tags, or tag associations that the tagging service can support. However, as we have shown in this paper, 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 vSphere 7.0 U1. For these numbers of tags, categories, and associations, we expect acceptable out-of-the-box performance for the UI, second- and third-party applications, and custom scripts written by customers, as long as the scripts adhere to our best practices. As mentioned earlier, tag definitions and category definitions are shared among all nodes in a Linked-mode domain, while associations are scoped to a single vCenter. Thus, the associations below are for a single vCenter Server in a Linked-mode domain, and the number 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 250,000 total associations across those nodes, with no more than 25,000 per vCenter.

Target of Limit	Limit
Category definitions	6,000
Tag definitions	8,000
Tag Associations	25,000

Table 7: vSphere 7.0 U1 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. While the numbers above are the same as for 6.7, performance is vastly improved in 7.0 U1, and we have shown in this paper how careful coding can allow a user to exceed these limits.

9 Concluding Remarks

In this paper, we have provided information about the proper use of tagging APIs for performance. We have discussed the following considerations:

Coding tips:

- For faster tag or category creation time, consider using multiple threads.
- It is faster to create tags spread across many categories, rather than putting all tags 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 pagination `list()` API, rather than `listAttachedObjects()` Or `listAttachedTags()`. The first two calls allow the user to specify a concrete list of tags or objects, and they allow chunking the requests to avoid overwhelming the tagging service. The `list()` API 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 earlier in the text.
- Linked-mode topology can impact the latency to replicate tags. We recommend a ring topology. For more details please see [this blog](#) [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 attach calls, the maximum request and response size we recommend is 2,000 VMs or 2,000 tags. If possible, consider these limits when choosing which APIs calls to use.
- For listing associations, if using `listAttachedTagsOnObjects()`, make sure to break down the list of objects into groups of 2,000.
- Consider using shorter tag names if you need to exceed 2,000 tags per attach call. The limit of 2,000 is based on the number of bytes in the response, not the sheer number of tags, so smaller tag names may allow you to retrieve more items in a single call.

- Creating and deleting tags creates tombstones in the tagging service database. These tombstones are needed for proper replication in linked mode, but excessive numbers of tombstones can impact performance. Please see [KB 52387](#) [5] for more information.
- 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, content library). If you are exceeding 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`.
- 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 to perform 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. The tag association table is a table within the main vCenter database. To perform such cleanup, please 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] VMware. (2020) VMware vSphere Automation SDKs Programming Guide.
<https://vdc-download.vmware.com/vmwb-repository/dcr-public/2fa085a7-3ff5-477e-9cf9-5a03eff7549a/68a01bae-01c6-4d48-84b5-e7678d1921dc/vsphere-automation-sdk-701-programming-guide.pdf>

- [2] VMware. (2020) vSphere Automation SDK for Java.
<https://techpartnerhub.vmware.com/sdks/vsphere-automation-java/7.0>

- [3] Ravi Soundararajen and Joseph Zuk. (2019, June) Writing Performant Tagging Code: Tips and Tricks for PowerCLI.
<https://blogs.vmware.com/performance/2019/06/writing-performant-tagging-code-tips-and-tricks-for-powercli.html>

- [4] Raju Angani, Ravi Soundararajan, and Maarten Wiggers. (2020, August) VMware vSphere 6.7 Tagging Best Practices.
<https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/performance/tagging-vsphere67-perf.pdf>

- [5] VMware. (2018, November) Troubleshooting and addressing accumulation of tombstones in a Platform Services Controller (52387).
<https://kb.vmware.com/s/article/52387>

- [6] VMware. PowerCLI SDK. <https://code.vmware.com/tool/vmware-powercli>

- [7] VMware. (2019, May) vCenter Enhanced Linked Mode for a vCenter Server Appliance with Embedded Platform Services Controller.
<https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vcenter.install.doc/GUID-7A90361C-E95B-4C53-A328-3DB9EE20536D.html>

- [8] Emad Younis. (2018, May) vCenter Server Architecture Part 1 – The Basics. [Online].
<https://emadyounis.com/vcenter-server-architecture-part-1-the-basics/>

- [9] VMware. (2019, May) Custom Attributes in the vSphere Web Client. <https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.vcenterhost.doc/GUID-73606C4C-763C-4E27-A1DA-032E4C46219D.html>
- [10] William Lam. (2015, January) Custom Attributes != vSphere Tags. <https://www.virtuallyghetto.com/2015/01/custom-attributes-vsphere-tags.html>
- [11] VMware. VMware vSphere Automation SDK for Java. <https://github.com/vmware/vsphere-automation-sdk-java>
- [12] VMware. Package com.vmware.cis.tagging. <https://vmware.github.io/vsphere-automation-sdk-java/vsphere/6.7.1/vsphereautomation-client-sdk/com/vmware/cis/tagging/package-summary.html>
- [13] VMware. Tagging Service SDK (PowerCLI). <https://code.vmware.com/docs/7335/powercli-11-0-0-user-s-guide/doc/GUID-F0088D49-CCFC-4F26-9C8C-845DE5AA951A.html?h=tags>
- [14] VMware. vSphere Java APIs. <https://github.com/vmware/vsphere-automation-sdk-java>
- [15] VMware. Java API for vSphere 7.0. <https://code.vmware.com/web/sdk/7.0/vsphere-automation-java>

About the Authors

Raju Angani is a staff performance engineer in the Performance team at VMware. His main areas of focus are vSphere tagging performance and vCenter database performance.

Ravi Soundararajan is a principal engineer in the Performance team at VMware, specializing in vCenter performance and scalability. He has presented numerous times about vCenter at VMworld. His twitter handle is @vCenterPerfGuy. Ravi received his SB from MIT and his MS and PhD degrees from Stanford, all in Electrical Engineering.

Maarten Wiggers is a staff II engineer and works on vCenter with a focus on tagging, policies, and resource management. He has been with VMware for 6 years and has a PhD from the University of Twente, The Netherlands.

Acknowledgements

We gratefully acknowledge Joseph Zuk, Oleg Zaydman, Sangeeta Jogeklar, Abhijith Marulaiah Prabhudev, Kyle Ruddy, Jake Robinson, Chung-Yen Chang, Yong Li, Udaya Ganiga, and Vishwa Srikanth for their assistance with the content of this paper.

11 Appendix

11.1 Experimental setup and software versions

We used VMware PowerCLI 10.1.0 build 8346946. Note that newer versions may demonstrate better performance.

We used Java 1.8 (“1.8.0_231”).

Our vCenter was vCenter 7.0 U1 running on ESXi 6.5 U2. Our inventory was composed of 2,000 hosts and 25,000 VMs (simulated using a proprietary VMware-internal tool). Because of the large number of VMs, we used the “Extra Large” size for the VCSA.

11.2 Java code examples

Some of these examples come from the following Github repository:

<https://github.com/vmware/vsphere-automation-sdk-java> [12].

For each of the Java examples below, we need 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 [10]), 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 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 Tag

In this example, we use the categoryId from “[Example J2: Create Category](#)” above.

```
/**
 * 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 10" in categoryId from above
String tagID = createTag("Windows 10", "DB Operating System", categoryId);
```

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

In this example, we use the VM DynamicID (vmDynamicId) from “[Example J0: VM DynamicID and Tag TagID](#)” above and the tagID from “[Example J3: Create Tag](#)” above.

```
/**
 * 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())

Attach 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())

Attach 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 (e.g., name, 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 code examples

In this section, we give select examples of PowerShell. 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 \[1\]](#) and are therefore not repeated here.

11.3.1 Example PO: Creating VMID Objects and retrieving Tag IDs

```
# 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 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")
```