DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. Reference herein to any social initiative (including but not limited to Diversity, Equity, and Inclusion (DEI); Community Benefits Plans (CBP); Justice 40; etc.) is made by the Author independent of any current requirement by the United States Government and does not constitute or imply endorsement, recommendation, or support by the United States Government or any agency thereof.

ORNL/TM-2024/3502

Oak Ridge National Laboratory Scaling a Podman Container Factory in the Cloud

David Heise[1]
Steven Amren[2]
Drew Morehead[2]
Brittany Flores[2]
Curtis Taylor[1]

[1]Oak Ridge National Laboratory
[2]The University of Texas San Antonio

February 21, 2025

OAK RIDGE National Laboratory

ORNL IS MANAGED BY UT-BATTELLE LLC FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2024/3502

National Security Sciences Directorate
Cyber Resilience and Intelligence Division

OAK RIDGE NATIONAL LABORATORY
SCALING A PODMAN CONTAINER FACTORY IN THE CLOUD

David Heise[1]
Steven Amren[2]
Drew Morehead[2]
Brittany Flores[2]
Curtis Taylor[1]

[1]Oak Ridge National Laboratory
[2]The University of Texas San Antonio

February 21, 2025

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831
managed by
UT-BATTELLE LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

LISTINGS

LIST OF ABBREVIATIONS

AD	Active Directory
CI	continuous integration
CI/CD	continuous integration/continuous deployment
CI/CDe	continuous integration/continuous delivery
CyManII	Cybersecurity Manufacturing Innovation Institute
GLRM	GitLab Runner Manager
IFT	Integrated Foundational Task
NIC	network interface card
ORNL	Oak Ridge National Laboratory
OS	operating system
PinP	Podman-in-Podman
PTC	pull through cache
RG	Resource Group
RHEL	Red Hat® Enterprise Linux
SAMI	System Assigned Managed Identity
SRDI	Secure Reseach & Development Infrastructure
SSD	solid state drive
SSH	Secure Shell
UID	user ID
vCPU	virtual CPU
VM	virtual machine
VMSS	Virtual Machine Scale Set

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the Advanced Materials & Manufacturing Technologies Office (AMMTO) Award Number DE-EE0009046. The views expressed herein do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

ABSTRACT

This technical report describes an implementation of a scalable continuous integration/continuous deployment infrastructure in the cloud using Microsoft Azure™ and GitLab™ resources. We utilize GitLab continuous integration podman executors to provide rootless container operations in both privileged and unprivileged modes of operation. Privileged operation mode permits container-in-container capability, while unprivileged mode can handle general job execution. We utilize the GitLab Fleeting plugin for Azure to manage the scaling of continuous integration execution resources. This creates a scalable, rootless, and isolated continuous integration/continuous deployment job execution infrastructure.

1. INTRODUCTION

The Secure Reseach & Development Infrastructure (SRDI) Integrated Foundational Task (IFT) provides a shared continuous integration/continuous delivery infrastructure to Cybersecurity Manufacturing Innovation Institute (CyManII) researchers. There are three key challenges that arise in providing such a shared infrastructure: user permission management, user and job isolation, and a need to scale resources based on utilization. We utilize three technologies to solve these problems: rootless podman, GitLab Fleeting, and Azure Virtual Machine Scale Sets (VMSSs).

One of our important use cases is container-in-container operations while maintaining user isolation and limited user privileges. Container-in-container operations are needed in order to build, scan, and deliver containers within our continuous integration/continuous deployment (CI/CD) pipelines. Utilizing a traditional a rootful container runtime requires giving users *privileged mode* access to a daemon running as a root user on the host. This access level permits the user of the CI/CD system to have root access to the host. Effectively, any user with this level of access has root level arbitrary code execution. Rather than relying on policy to control access to the CI/CD executor, we wish to utilize technical means to prevent elevated access to the system. We therefore utilize a rootless podman runtime, which executes in the context of a user on the container host. In order to support container-in-container operations, rootless podman still requires a privileged mode; however, the level of access to the host is restricted to that of a limited, non-root, user.

By utilizing GitLab Fleeting and Azure Virtual Machine Scale Sets we are able to arbitrarily scale both in and out depending on the current CI/CD utilization. As a configuration option, we can also allow only a single continuous integration (CI) job to execute per virtual machine, which provides strong isolation between users in the case of privileged mode executions. For protection against data persistence, virtual machines can also be deleted after a single job execution.

1.1 MOTIVATION

We have built out a container factory template following the DevSecOps principles described in [2]. However, as usage of this pattern in our CI/CD infrastructure has increased, users have been experiencing performance bottlenecks. This affected container-in-container operations most acutely because only a single job at a time is permitted to run on a privileged executor, even when functioning in rootless mode. Therefore, a solution that provided greater performance was desired. By utilizing GitLab CI/CD autoscaling we can run an arbitrary number of CI jobs in parallel utilizing executors created on demand. In addition to providing an increase in job throughput, this results in cost savings because Azure compute resources are only running when needed rather than waiting idle for work to do. The 24/7 compute resource is reduced to a small manager VM with a low cost virtual hardware allocation. If load increases beyond the current architecture limitations, it is simple to expand the available number of executor VMs, or scale up the size of the manager with minimal or no software reconfiguration.

1.2 BACKGROUND ASSUMPTIONS

For purposes of this document, it is assumed the reader has knowledge of how to install podman 4.0 or greater[1], install GitLab Runner[2], register a GitLab Runner[3], and the basics of working with Azure[4] (e.g., the Azure Portal, Azure Wizards, setting up networking, setting up resource groups). The specifics of how to configure these tools will be discussed. The steps described here exist as a snapshot at a point in time of the tool user interfaces. Cloud offerings and GitLab frequently modify their user interfaces, and the reader may need to adapt the instructions to new workflows.

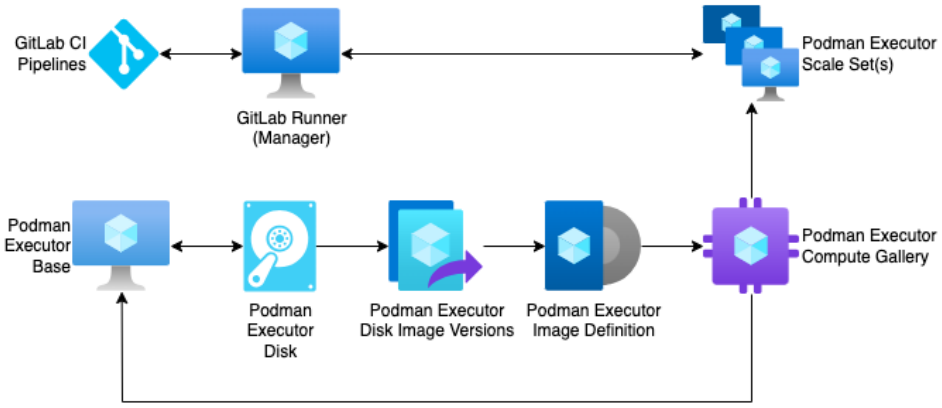
For setting up a GitLab runner with rootless podman without virtual hardware instance scaling, refer to [1].

[1] https://podman.io/docs/installation
[2] https://docs.gitlab.com/runner/install/
[3] https://docs.gitlab.com/runner/register/
[4] https://azure.microsoft.com

2. ARCHITECTURE DISCUSSION

The autoscaling GitLab Runner infrastructure utilizes GitLab and Azure scaling features. On the Azure side, we utilize VMSS to create and destroy virtual machines on demand. The virtual machines are booted from an image configured to act as a GitLab Runner Docker Executor with the podman container runtime. We will refer to this as a Podman Executor, although the GitLab documentation refers to this as a 'Docker' executor due to communications using the Docker communication protocol. For GitLab, we utilize the [5]GitLab Runner Autoscaler, which in turn utilizes the [6]Azure Fleeting Plugin. The GitLab Runner registered in the GitLab repository manager now acts as a manager for the VMSS to boot executor VM on demand, assign jobs for execution, and then delete the virtual machine once it fulfills its configuration limits.

Figure 1. Architecture Overview



The diagram in Figure 1 is an overview of the Azure component architecture. The GitLab Runner Manager (GLRM) watches for pipelines to execute from the GitLab repository manager. When it detects a job to execute, it will communicate with a Podman Executor Scale Set to create a GitLab Runner Executor virtual machine if needed. A single GLRM can have multiple logical runners which communicate with separate scale sets. Once a GitLab Runner Executor VM has booted, the GLRM communicates with the Executor VM to execute the pipeline tasks and return the results to GitLab. The Podman Executor Scale Set is configured to boot off of a Podman Executor Disk Image Version.

The Podman Executor VMSS disk image versions are taken from the Podman Executor Base virtual machine. This virtual machine is configured with the podman container runtime and a podman socket user service so it can function as a podman executor for GitLab. An image is taken of the Podman Executor Disk (that is, the operating system (OS) disk attached to the Podman Executor Base virtual machine) and stored as a Disk Image Version in a Podman Executor Compute Gallery. Once the virtual machine image is taken, the Podman Executor Base virtual machine and its disk can be deleted until there is a need to update the image as part of a maintenance plan.

In order to perform routine maintenance on a Podman Executor Disk Image stored in the Podman Executor Compute Gallery (e.g., OS patching), a new Podman Executor Base VM is created from a Podman Executor Disk Image Version in the Podman Executor Compute Gallery, and the process above for making system modifications, taking an image, and configuring the Podman Executor VMSS (if it is not configured to use the latest image) is repeated.

[5] https://docs.gitlab.com/runner/runner_autoscale/
[6] https://gitlab.com/gitlab-org/fleeting/plugins/azure

3. IMPLEMENTATION

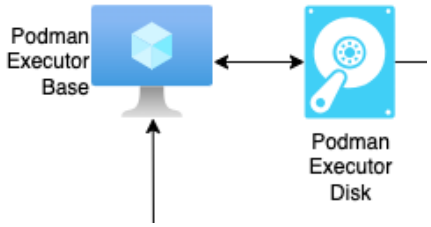
Throughout this section, we will trace the implementation of the architecture described above. Each subsection will focus on a specific area of interest until we have described the full system.

3.1 CONFIGURE THE RUNNER EXECUTOR VIRTUAL MACHINES

This section covers the initial bootstrap process for the runner executor disk, see Figure 2. Updates from an existing disk image version are covered under Section 4.

Instructions will be given for the creation of the resource in Azure, then the system setup.

Figure 2. Architecture Focus: Executor Virtual Machine



3.1.1 Create Azure Virtual Machine

Only the directly pertinent subset of Azure settings are given. Settings not enumerated here are environment specific (e.g., resource group, subnet, and most network security settings).

Basic system requirements are enumerated in Listing 1. We require a Red Hat® Enterprise Linux (RHEL) based image to obtain a recent enough release of podman, although any distribution flavor with podman version greater than or equal to 4.0 is expected to work.

Listing 1. Create a virtual machine

```
Image: Red Hat Enterprise Linux 9.3 (LVM) - Gen2
VM architecture: x64
Size: Standard D2s v3 (2 vCPUs, 8 GiB memory)
OS disk size: Image default (64GB)
OS disk type: Premium SSD LRS
Use managed disks: Yes
Delete OS disk with VM: Enabled
Ephemeral OS disk: No
Public IP: None
Delete NIC when VM is deleted: Enabled
```

The virtual CPU (vCPU) and memory configuration at this phase are arbitrary; at runtime their configuration will be managed by the scale set. Just enough resources are needed to configure the system.

The disk configured here persists into the scale set in both its size and performance tier. The performance tier is the primary consideration; this virtual machine should not require significant storage capacity. 64GB is the lowest available performance tier at this time. A higher performance tier may be desirable, but may require

unnecessary storage capacity as well. It may be possible to override during VMSS deployment, but the disk will not automatically expand to use additional space.

It is possible to utilize Azure Active Directory (AD) based Secure Shell (SSH) Login, so that and any other needed extensions may be installed on the executor virtual machines. However, this is not required to follow this manual successfully and this architecture has been deployed without it as well.

The authentication mechanism chosen at this stage will be overridden at a later stage; use whatever is typical in the environment with respect to gaining root level access to install and configure the system.

For the operating system install we will resize the home area filesystem, install podman, enable users to run systemd services, and add a limited user to run a podman socket service. The CI/CD jobs will run in the context of this rootless user. The commands to accomplish this are given in Listing 2

The default RHEL 9 VM in Azure has much of the disk unallocated. By default, podman stores container images in the executing user's home area. Container images can be large, therefore, we expand the home partition significantly to allow for the execution of substantially sized images and any data the user may insert into the container instance during job execution. One could expand to a larger size than specified here, or use a larger disk as well.

Listing 2. Podman Executor VM Operating System Configuration

```
[root]# lvresize /dev/rootvg/homelv --resizefs --size 32G
[root]# dnf install podman podman-docker
[root]# adduser gitlab-runner-executor
# Allow a normal user to run a systemd service
[root]# loginctl enable-linger gitlab-runner-executor
# Create a strong password, and note it for later.
[root]# passwd gitlab-runner-executor
```

The default open file limit on many Linux® systems is 1024; we will increase that substantially to prevent resource exhaustion during CI/CD pipeline execution. The configuration in Listing 3 allows setting the open file limit higher.

Listing 3. /etc/security/limits.d/90-nofile.conf

```
soft nofile 65536
hard nofile 65536
```

Add the lines in Listing 4 to `/home/gitlab-runner-executor/.bashrc` so the socket can start correctly and the GLRM can locate the correct socket to connect to.

Listing 4. /home/gitlab-runner-executor/gitlab-runner/.bashrc

```
export XDG_RUNTIME_DIR=/run/user/$UID
export XDG_DATA_HOME=$HOME/.local/share
export XDG_CONFIG_HOME=$HOME/.config
export DOCKER_HOST=unix://$XDG_RUNTIME_DIR/podman/podman.sock
ulimit -n 65536
```

Switch to the limited user just created and enable and start the podman socket service as shown in Listing 5.

Listing 5. Enable Podman User Service

```
[root]# sudo -iu gitlab-runner-executor
[gitlab-runner-executor]$ systemctl --user enable podman.socket
[gitlab-runner-executor]$ systemctl --user start podman.socket
```

Reboot the system. Ensure the podman socket service is working correctly by checking the service status as in Listing 6. The user ID (UID) may differ.

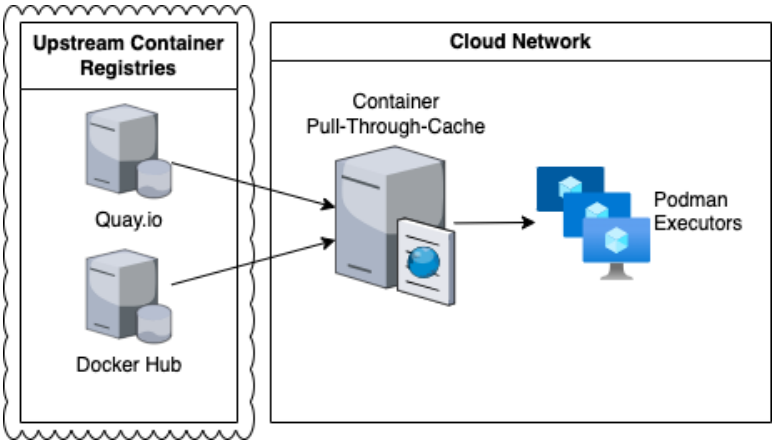
Listing 6. GitLab Runner Service Verification

```
[gitlab-runner-executor]$ systemctl --user status podman.socket
 podman.socket - Podman API Socket
    Loaded: loaded (/usr/lib/systemd/user/podman.socket; enabled; preset: disabled)
    Active: active (listening) since Mon 2024-03-25 19:15:12 UTC; 14min ago
     Until: Mon 2024-03-25 19:15:12 UTC; 14min ago
  Triggers: podman.service
      Docs: man:podman-system-service(1)
    Listen: /run/user/1001/podman/podman.sock (Stream)
    CGroup: /user.slice/user-1001.slice/user@1001.service/app.slice/podman.socket
```

Shut down the system.

3.1.2 Optional Component: Pull Through Cache

Figure 3. Pull Through Cache to Executors



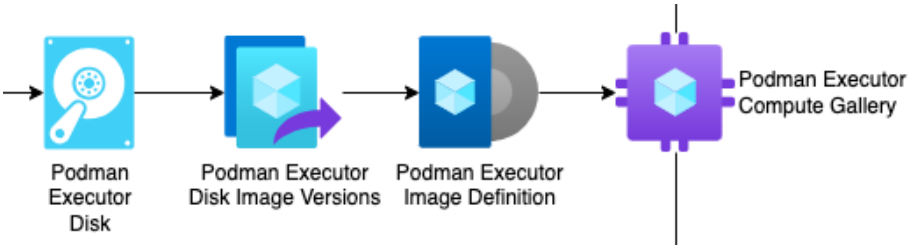
A component that can be included in the container factory architecture is a container image pull through cache (PTC). As shown in Figure 3, the PTC will stand between the executors and the upstream container registries. While a stand-alone executor running in an always-on VM will maintain its own local cache of images, the ephemeral executor VMs utilized here will not have an effective local cache. Therefore, setting up a PTC is highly recommended both for performance reasons and because jobs will fail if the upstream registry pull limits are reached.

Setting up a PTC and configuring it to be used by the podman runtime is beyond the scope of this document, but you can refer to [1] for details on how to build the PTC and configure podman on the executor image to utilize it.

3.2 CREATE AZURE COMPUTE GALLERY

This section will management of the disk images from the executors, as shown in Figure 4.

Figure 4. Compute Gallery Architecture



The Azure Compute Gallery stores image definitions for later use in a VMSS. We now create a compute gallery that will hold our Podman Executor Base image definition.

Navigate to the Azure compute galleries dashboard and click `"Create"` as shown in Figure 5.

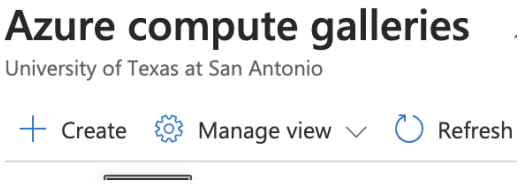


Figure 5. Create Compute Gallery

Enter the basic information, e.g., Name and Resource Group.

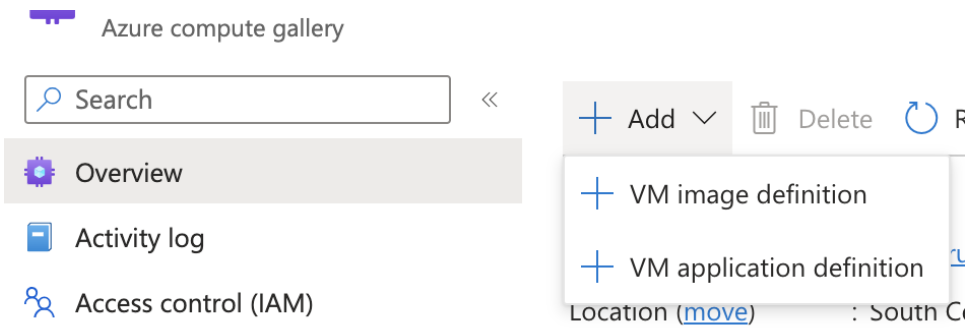
For sharing method select `"Role based access control (RBAC)."`

3.3 CREATE A VM IMAGE DEFINITION

Next we will take an image of the disk of the temporary Podman Executor base VM and store it in the compute gallery that was just created. The image definition provides some parameters for how an image is intended to be used; for example, setting what VM generation can boot from the image. Recommended settings can be found in Listing 7.

Navigate to the compute gallery you just created and select `"Add → VM image definition"` shown in Figure 6.

Figure 6. Create VM Image Definition



Listing 7. Image Definition

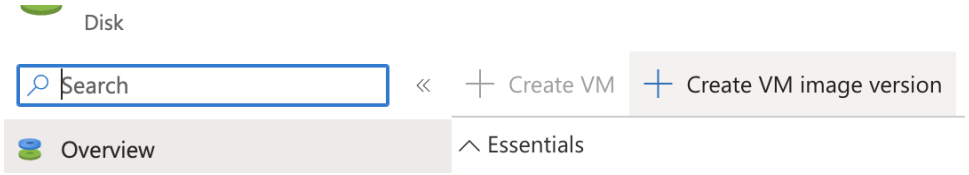
```
OS type: Linux
Security type: Trusted launch supported or Standard
VM generation: V2
Accelerated networking: Enabled
VM architecture: x64
OS state: Generalized
Publisher: <Your Organization>
Offer: <Arbitrary>
SKU:<Arbitrary>

Recommended VM VCPUs: 1-16
Recommended VM memory: 1-32 GB
Excluded disk types: None
VM image definition end of life date: None
```

3.4 TAKE THE INITIAL VM IMAGE

Now we have a Compute Gallery and an image definition and can take the actual virtual machine image and store it in the image definition.

Navigate to the virtual machine disk dashboard (`Virtual Machine → Settings → Disks → Click the OS Disk`) and click `Create VM Image Version`, see Figure 7. Reccomended settings are given in Listings 8 and 9.

Figure 7. Create VM Image Version



Listing 8. Basics

```
Subscription: <your subscription>
Resource group: <your resource group>
Region: <your region>
Target Azure compute gallery: <the compute gallery created above>
Target VM image definition: <The compute image definition created above>
Version number: <arbitrary version number>
Source: Disks and/or snapshots
OS disk: <name of the OS disk selected above>
Exclude from latest: No
End of life date: <arbitrary date in the future>
Lock deleting Replicated Locations: Yes
Shallow replication: No
Default replica count: 1
```

Listing 9. Replication

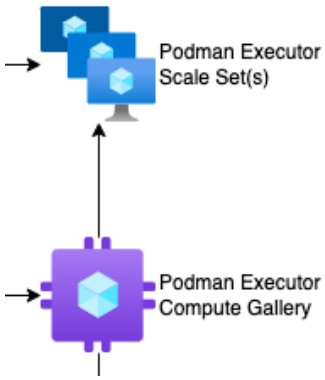
```
Default storage sku: Premium SSD LRS
Default replica count: 1
Target Regions: <your choice>
Replica count: <1, or your choice>
Stoage SKU: Premium SSD LRS for first entry, arbitrary for the rest
```

We use Premium SSD LRS as our primary storage because disk performance is important for system boot times and container operations.

3.5 CONFIGURE THE SCALE SETS

This section will cover configuration of the scale sets. Scale sets are used to create an arbitrary number of virtual machines booted from image definitions contained in an Azure Compute Gallery, as shown in Figure 8.

Figure 8. Scale Set Image Source



We will configure two identical scale sets. One will be used for unprivileged Podman executors, and the other set will be used for privileged Podman-in-Podman (PinP) executors. We use two scale sets because the GLRM sets (un)privileged mode and the creation/deletion of VMSS assets at the level of the logical runner configuration. The password configuration is important here. Due to the mechanism of Azure virtual machine access with Fleeting, authentication must be set to "Password" so that the GLRM can log in to the limited user created previously with SSH Password authentication. The admin user user name should be set to something other than the default, and the password should be strong. However, neither of these data need to be retained; there should be no reason to log into the ephemeral executor VMs with an admin account.

Navigate to Virtual Machine Scale Sets and click "Create" as shown in Figure 9.

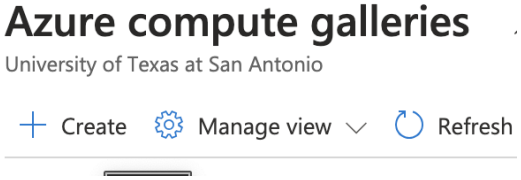


Figure 9. Create Virtual Machine Scale Set

The Virtual Machine Scale Set is more complex than the other workflows and as of this writing also requires a workaround for an Azure user interface issues, so we will step through it tab by tab.

Listing 10. Basics

```
Virtual machine scale set name: <Arbitrary>
Orchestration mode: Uniform
Security type: <Filler based on image selected>
Image: Select the image created above (See all images -> Shared images -> Select the
    image)
Run with Azure Spot discount: Unchecked
Size: <Arbitrary>
Authentication type: Password
Username: <Arbitrary>
Password: <Arbitary>
```

Listing 10 shows the basic settings. We will be creating two scale sets, so give names that distinguish between the set intended for unprivileged podman and privileged podman execution(e.g., `CincExecutor` and `ContainerExecutor`). It is critical that the VMSS be placed in its own Resource Group (RG). This will be used to manage Azure permissions when setting up the GLRM later.

Skip the Spot configuration section.

The selection of OS disk size as shown in Listing 11 is primarily for the associated performance tier. We have not had any issues with the 64GB/P6 tier, but this can be selected as needed. The size selected should be at least as large as the size of the disk image in the Image Definition. We choose premium solid state drive (SSD) for performance reasons, but as these VMs are ephemeral there is no need for redundancy.

Listing 11. Disks

```
OS Disk Size: 64GB (P6)
OS Disk Type: Premium SSD (locally-redundant storage)
Key Management: Platform managed key
```

As of this writing, the Networking section of the workflow has an issue with creating the network interface card (NIC) for the scale set. The NIC needs to be created after first viewing the "Review + Create" screen; any network configuration prior to viewing that screen will be lost. Nevertheless, we will describe how to create and configure the NIC here.

The virtual network the VMSS is placed in needs to meet several criteria. It needs (1) a route to the GLRM, (2) a route to the GitLab server, (3) a subnet with enough free space to house the maximum intended number of runner executors, (4) a route to any intended external resource (e.g., the Internet to pull in packages), and (5) a route to reach the container image PTC, if there is one.

Click the "Edit" pencil next to the NIC and enter the setting from Listing 12.

Listing 12. Network Interface

```
Name: <Arbitrary>
Subnet: Your subnet created for the VMSS
NIC network security group: Follow your network policy.
Public inbound ports: None
Public IP address: Disabled
Accelerated networking: Enabled
```

Skip the Load Balancing Options section.

Listing 13. Scaling

```
Default Instance Count: 0
Scaling Policy: Manual
Scale-in Policy: Any
Apply force delete to scale-in operations: Checked
```

The scaling will be controlled by the GLRM so we do not need any Azure management of the VMSS scale, as shown in Listing 13.

No required settings in Management, Health, and Advanced.

As noted above regarding the Azure UI issue, on the "Review and Create" tab, review all settings carefully, especially the networking configuration. Go back to the Networking tab and apply the changes a second time if needed, and they should remain applied as configured the second time through. Check the settings again. You are now ready to create the VMSS. Once the first scale set is created, go through the process a second time for the other executor type.

3.6 GITLAB RUNNER MANAGER CONFIGURATION

The GLRM shown in Figure 10 controls job execution caused by activity on repositories in GitLab and controls scale set scaling to handle the jobs based on current load and capacity. The GLRM will be a persistent virtual machine that runs 24/7. However, the hardware resources needed are light. We have successfully run the infrastructure with 18 executors across two scale sets on a 2-core burstable vCPU (`Standard_B2ls_v2`) VM.

Figure 10. GitLab Runner Manager

3.6.1 Create The Virtual Machine

For virtual machine creation, only a subset of Azure settings are given explicitly. Settings not enumerated here are environment specific (e.g., resource group, subnet, most network security settings).

Listing 14. System Requirement

```
Image: Red Hat Enterprise Linux 9.3 (LVM) - Gen2
VM architecture: x64
Size: B2ls_v2 (2 burstable vCPUs, 4 GiB memory)
OS disk size: Image default (64GB)
OS disk type: Premium SSD LRS
Use managed disks: Yes
Delete OS disk with VM: Enabled
Ephemeral OS disk: No
Public IP: None
Delete NIC when VM is deleted: Enabled
```

The primary settings in Listing 14 that may be tweaked if needed for performance are the VM size and the OS disk performance tier. However, both our experience and the GitLab documentation indicate that GLRM tasks are not generally resource intensive.

3.6.2 Configure System Assigned Managed Identity

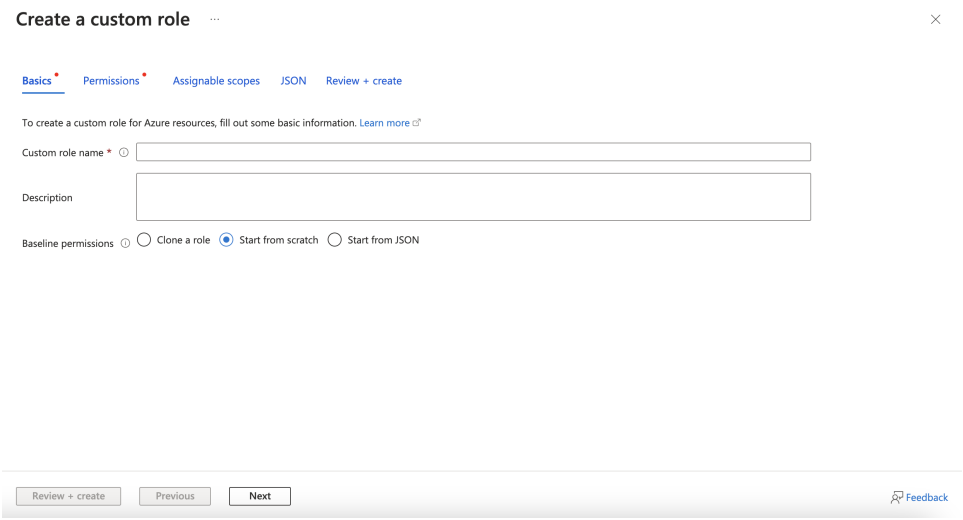
The GLRM will need permission within the Azure cloud infrastructure to manage a VMSS. This is accomplished by assigning the GLRM a System Assigned Managed Identity (SAMI) role. A system managed identity allows the system it is installed on to authenticate against Azure and execute Azure actions using the Azure CLI or API. This is a system level authorization and not tied to a particular user on the system in question. We utilize this method because this effectively a single-user system.

It is important to know that the GLRM is not involved in executing CI/CD scripts. It only manages the start up and shut down of executors in the VMSS. Therefore, users of the CI/CD system do not have access to its Azure authorizations at any time.

3.6.3 Create the Azure Role

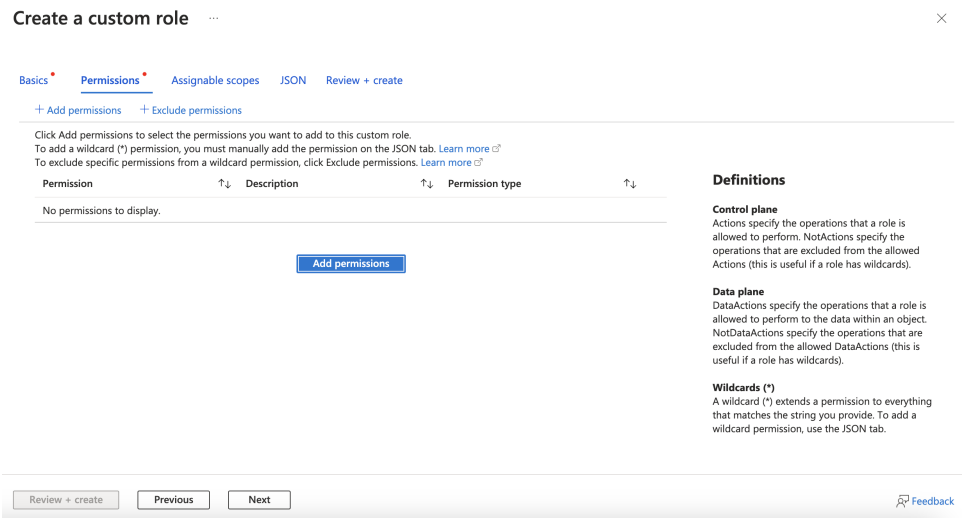
We must first create the Azure Role with the correct permissions to manage a VMSS. The Azure wizard steps are shown in Figures 11 through 15.

Figure 11. Create Role



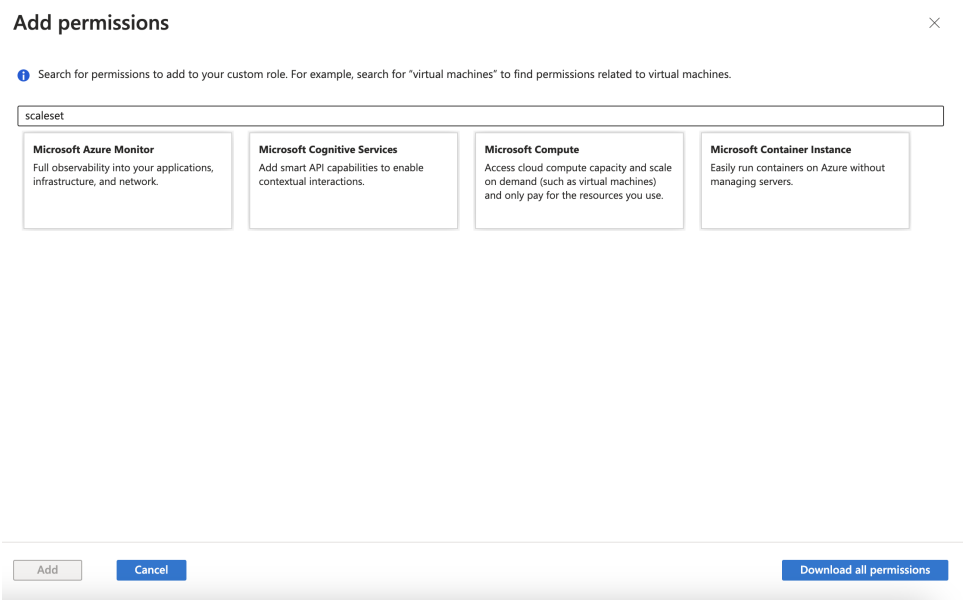
The first step of role creation is to give a name and description. Here we use the name `VmssManager`.

Figure 12. Begin Adding Permissions



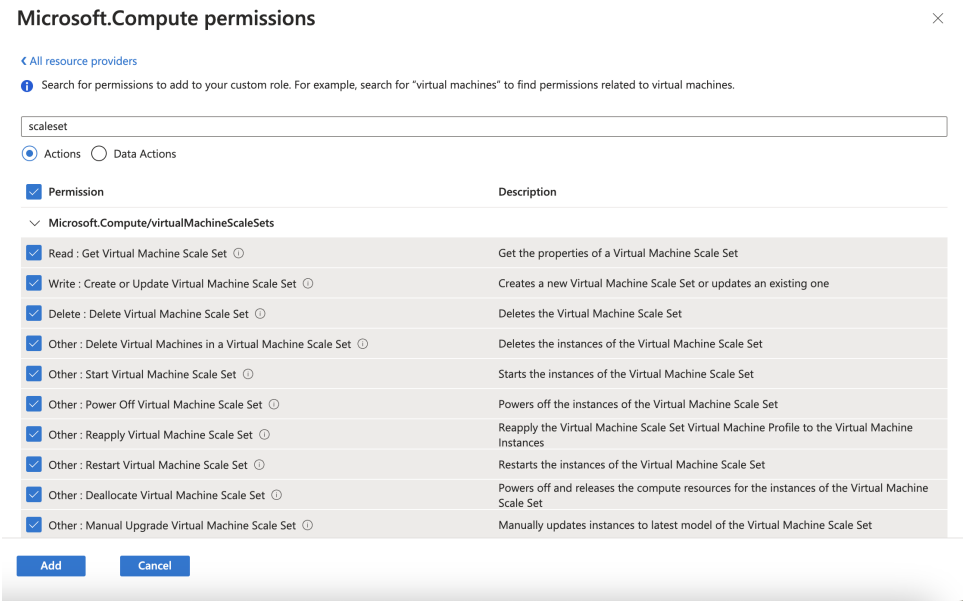
We can now begin adding specific permissions. We first click "Add Permissions."

Figure 13. Permission Category Selection



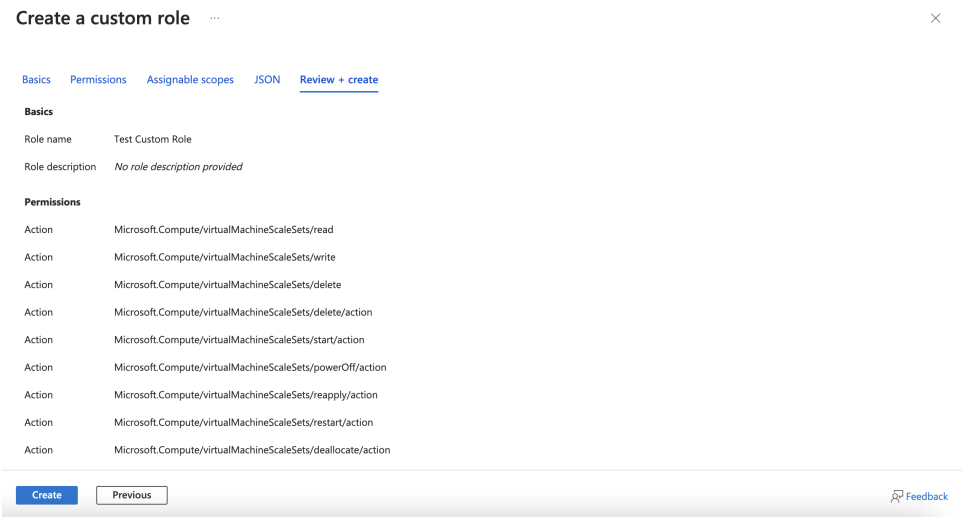
Azure has hundreds of available permissions. The first step is to locate the category that contains permissions relevant to managing VMSS. The VMSS permissions are inside the `Microsoft.Compute`.

Figure 14. Specific Permission Selection



For the purposes of this document, we give the role all permissions related to managing VMSS by searching for `scaleset` and choosing the checkbox next to `Permission` to select all. While it is likely a more restricted set of permissions could be utilized, that has not been explored for this report.

Figure 15. Confirm and Create



We will call our SAMI role `VmssManager`.

3.6.4 Assign the System Assigned Managed Identity

Once we have created the Azure role we can turn on SAMI for our GLRM. We navigate to the GLRM virtual machine and select `Security → Identity`.

Figure 16. Turn on System Assigned Managed Identity

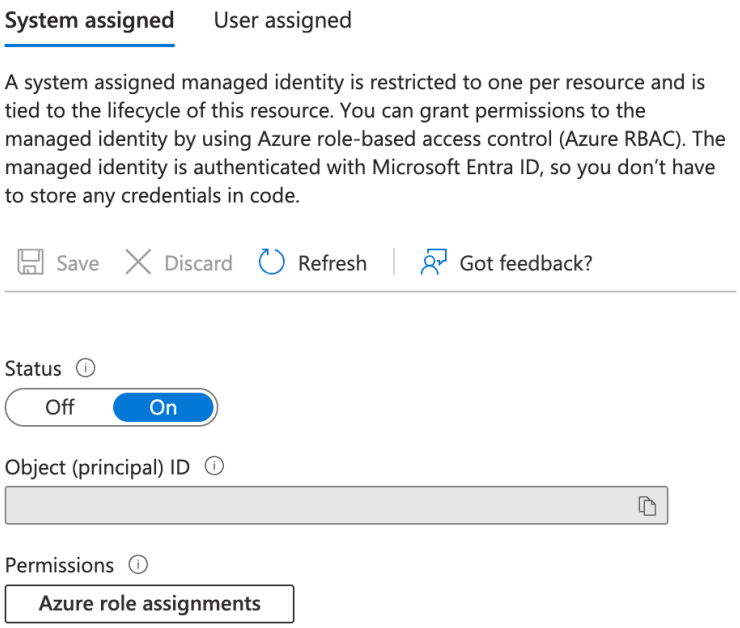


Figure 16 shows the `Identity` screen for the virtual machine. We change `Status` to on, click `Save`, and then click `Azure role assignments`.

Figure 17. Assign Azure Role to System

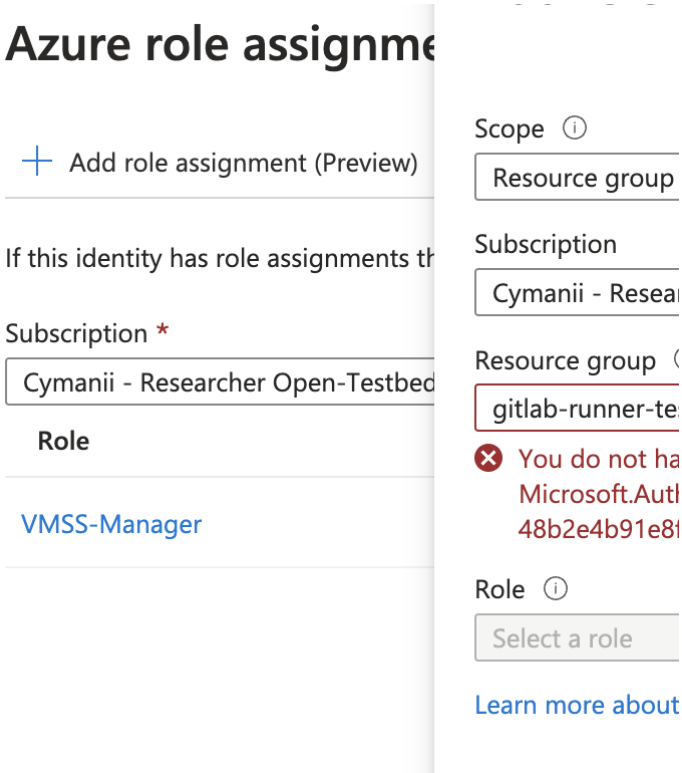


Figure 17 shows the role assignment interface. We select `Resource Group` as the scope for the role, and select the subscriptions and resource group that contains the CI/CD executor VMSS. This restricts the GLRM to the VMSS within the specified subscription and resource group.

3.6.5 Configure the Virtual Machine Operating System

As shown in Listing 15 we expand the root volume and file system, follow the instructions for installing a GitLab Runner, and install the Azure Fleeting Plugin. This virtual machine will not need significant user storage.

Listing 15. Configuring the VM Operating System

```
[root]# lvresize /dev/rootvg/rootlv --resizefs --size +5G
[root]# curl -L \
"https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.rpm.sh" | \
sudo bash
[root]# sudo yum install gitlab-runner
[root]# sudo yum install azure-cli
[root]# wget \
https://gitlab.com/gitlab-org/fleeting/plugins/azure/-\
/releases/v0.2.0/downloads/fleeting-plugin-azure-linux-amd64
[root]# mv fleeting-plugin-azure-linux-amd64 /usr/local/bin/
[root]# chmod +x /usr/local/bin/fleeting-plugin-azure-linux-amd64
[root]# ln -s /usr/local/bin/fleeting-plugin-azure-linux-amd64 \
       /usr/bin/fleeting-plugin-azure
[root]# fleeting-plugin-azure --version
Name:       fleeting-plugin-azure
Version:    v0.2.0
Git revision: f8ddf20a
Git ref:    refs/pipelines/1150104799
GO version: go1.19.6
Built:      2024-01-24T17:16:42+0000
OS/Arch:    linux/amd64
```

To ensure the GLRM always has the needed authorizations, we configure a crontab entry to periodically refresh the Azure login as shown in Listing 16.

Listing 16. /etc/crontab

```
@reboot az login --identity
0 */8 * * * az login --identity
```

The GitLab Runner systemd service will function out of the box; however, we have found that some overrides can provide increased service reliability.

Listing 17. systemd Unit Overrides: gitlab-runner

```
[Service]
KillSignal=SIGQUIT
ExecStartPre=-az vmss scale --resource-group '<Unprivleged VMSS Resource Group>' \
  --name '<Unprivleged VMSS Name>' --new-capacity 0
ExecStartPre=-az vmss scale --resource-group '<Privleged VMSS Resource Group>' \
  --name '<Privleged VMSS Name>' --new-capacity 0
```

Listing 17 gives some recommended gitlab-runner systemd unit file overrides. We open the systemctl override editor with the command `[root]# systemctl edit gitlab-runner`.

Setting the kill signal to `SIGQUIT` will tell the runner to cease accepting new jobs and to delay exit until active jobs are finished. The default service kill timer is 60 seconds, so you may wish to combine this with setting `TimeoutSec` to the same length as the CI/CD timeout extend the timeout period long enough to prevent job failure on service restart; however, long service stop timeouts may not be desirable.

The two `ExecStartPre` directives destroy all VMs in the privileged and unprivileged executor VMSSs. Occasionally, the GLRM may not be able to recover the VMSS into a usable state on start up, so this creates a clean slate for the Fleeting plugin to start in. Restarting the service on some schedule may also enhance overall availability.

We assume the reader has GitLab Repository Manager application administrator privileges to set up a shared runner for all GitLab users. Similar steps can be followed to create a group or project runner, since tokens can be created for those access levels as well. The token is obtained through a different GitLab workflow for each strategy. Refer to the GitLab documentation.

3.6.6 Register Runners in GitLab

We create two runners in GitLab, one for unprivileged jobs and one over privileged jobs that have PinP capbaility.

Navigate to the New Instance Runner screen in GitLab (Admin Area → CI/CD → Runners → New Instance Runner) to register the unprivileged podman runner as shown in Listing 18. We allow this runner to execute untagged jobs, but that is not required.

Listing 18. Platform

```
Operating systems: Linux
Tags: podman, x86
Run untagged jobs: checked
Runner Description: GitLab Podman Autoscaled Manager
```

A token will now be displayed: "`The runner authentication token <token>.`" Note it for use in the configuration file.

Navigate to the New Instance Runner screen in GitLab (Admin Area → CI/CD → Runners → New Instance Runner) to register the privilged podman runner with the settings given in Listing 19. Because this is a privileged runner, we require users to opt into using it with job tags.

Listing 19. Platform

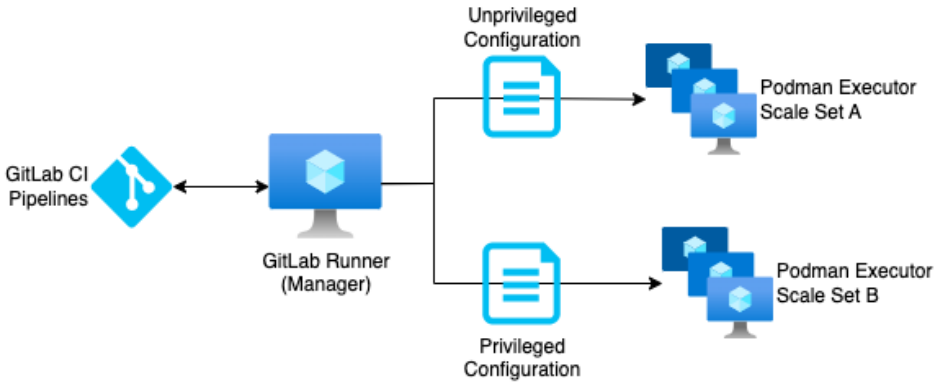
```
Operating systems: Linux
Tags: pinp, x86
Run untagged jobs: unchecked
Runner Description: GitLab Podman Autoscaled Manager
```

A token will now be displayed: "`The runner authentication token <token>.`" Note it for use in the configuration file.

3.6.7 Configuring the GitLab Runner Manager Service

When configuring the manager, it is important to know that the distinction between unprivileged and privileged mode is made a podman run-time option set by the GLRM. That is, the distinction is made in the manager configuration and is not related to the executor virtual machine images or the VMSS settings. There are two separate scale sets so that the manager can have two separate runner configurations each corresponding to one run-time privilege level, and manage them independently and in isolation as depicted in Figure 18. This allows CI/CD users to use the appropriate runner set for the job they need to execute simply via CI/CD job tags.

Figure 18. Two Runner Configurations



We will step through each major section of the GLRM configuration below. You can find a full sample configuration without breaks with both a privileged and unprivileged runner section in Appendix A.

Although we utilize podman as our container runtime, the GitLab Runner executor is "docker-autoscaler" because the docker protocol is used for communication. Since Podman supports the docker protocol, it can be used as a drop in replacement run-time instead of Docker.

Listing 20. GitLab Runner Configuration: Preamble

```
concurrent = 18 # Maximum number of possible parallel executors
check_interval = 0
shutdown_timeout = 0

[session_server]
  session_timeout = 1800
```

The first lines of the GLRM configuration are given in Listing 20. The primary setting for our purposes is `concurrent`. This should be set to the maximum possible number of executors across all of the logical runners controlled by this manager. In other words, the sum of the size of the privileged and unprivileged runner subnets, minus address space reservations by Azure.

Listing 21. GitLab Runner Configuration: GitLab Connection & Environment Variables

```
[[runners]]
  name = "Podman Runner (Managed)"
  limit = 9 # The combined limit of both runners must not exceed the "concurrent"
            # setting above
```

```
url = "<GitLab URL>"
token = "<The Podman Runner Token>"
shell = "sh"

environment = [
  "FF_NETWORK_PER_BUILD=1", # Isolated networking for each job.
  "FF_USE_DOCKER_AUTOSCALER_DIAL_STDIO=true", # Use the Docker stdio protocol.
  "FF_ENABLE_JOB_CLEANUP=true", # Clean working directory at end of job. Optional.
  "FF_SCRIPT_SECTIONS=true", # Provides job log UI enhancements. Optional.
  "FF_USE_IMPROVED_URL_MASKING=true" # Better masking of sensitive URLS. Optional.
]
```

Listing 21 is the start of the configuration section for a logical runner. In this example, we give a name to indicate this is an unprivileged runner with GLRM managed scaling. The `limit` parameter sets the maximum number of jobs to be handled by this runner. Because we create a virtual machine for every job, this is equal to the size of the VMSS subnet minus Azure address reservations. The two required environment variables are `FF_NETWORK_PER_BUILD` and `FF_USE_DOCKER_AUTOSCALER_DIAL_STDIO`. The option `FF_NETWORK_PER_BUILD` is required for podman executors, and `FF_USE_DOCKER_AUTOSCALER_DIAL_STDIO` ensures the correct communication protocol is utilized between the manager and executor.

Listing 22. GitLab Runner Configuration: Executor

```
executor = "docker-autoscaler"

# Docker Executor config
[runners.docker]
  # Check the UID for the socket path!
  host = "unix:///run/user/1001/podman/podman.sock"
  tls_verify = false
  image = "quay.io/containers/podman:latest"
  privileged = false # Set according to need
  disable_entrypoint_overwrite = false
  oom_kill_disable = false
  # We pass in the host container registry configuration as a read-only bind mount so
  # that the images hit the container pull through cache
  volumes = ["/certs/client",
    "/etc/containers/registries.conf:/etc/containers/registries.conf:ro"
  ]
  shm_size = 0
```

The section in Listing 22 configures the options for the podman runtime on the executor. The `host` parameter specifies the user service socket to communicate on. Refer to taken from Listing 6. The `image` directive sets the default image. We use the podman image since container operations are our default focus. The `privileged` flag is set to false, since this is our unprivileged generic runner. For the privileged runner, we simply change this flag to `true`. In the `volumes` directive we pass in the `registries.conf` file from the host. This causes the podman-in-podman runtime in the executor to use the same registry configuration as this host. In our configuration we use a PTC and this is the configuration line that allows podman-in-podman operations to make use of the PTC.

Listing 23. GitLab Runner Configuration: Executor: Autoscaling

```
# Autoscaler config
```

```
[runners.autoscaler]
  plugin = "fleeting-plugin-azure"

  capacity_per_instance = 1 # Each job gets its own VM
  max_use_count = 2 # Each VM is used for this many jobs
  max_instances = 9 # Should equal the limit setting above
  delete_instances_on_shutdown = true

[runners.autoscaler.plugin_config]
  name = "<VMSS Name in Azure>" # VMSS Name
  subscription_id = "<The Azure subscription ID of the VMSS>"
  resource_group_name = "<The Azure resource group name of the VMSS>"

[runners.autoscaler.connector_config]
  username = "gitlab-runner-executor"
  password = "<the password for the gitlab-runner-executor user>"
  use_static_credentials = true
  timeout = "10m"
  keepalive = "60s"
  use_external_addr = false

[[runners.autoscaler.policy]]
  idle_count = 0 # How many idle instances are required
  idle_time = "30m0s" # How long instances exceeding idle_count will remain
```

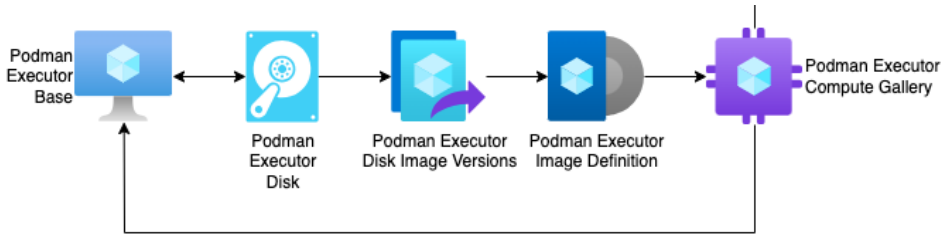
The configuration section in Listing 23 specifies the parameters that control VMSS scaling. The `plugin` parameter specifies which Fleeting plugin to use. In our case this is Azure, but there are plugins for other cloud providers as well; the required parameters for other provider will differ from those given here. The `capacity_per_instance` directive is how many jobs per virtual machine executor. Setting this to 1 means every job gets its own VM. The parameter `max_use_count` is the maximum number of jobs an executor will run before retiring an executor VM. The choice for this parameter depends on how isolated jobs need to be from one another. We generally choose one for privileged jobs and a number greater than one for generic jobs. The parameter `delete_instances_on_shutdown` configures whether to delete the VMSS machine when no longer in use. The information for `runners.autoscaler.plugin_config` is taken from the Azure portal in your environment. The password in `runners.autoscaler.connector_config` is the one set in Section 3.1.1 Listing 2.

The `runners.autoscaler.policy` controls the idle executor capacity awaiting work. The `idle_count` parameter controls the minimum number of idle executors that are always available. The `idle_time` parameter controls how long executors in excess of `idle_count` will remain remain idle before shutting down. For example, if an executor has `max_use_count = 2` then it will remain active for `idle_time` before being removed if it does not receive a second job.

4. MAINTENANCE

Patching the executor images needs to be part of regular system maintenance. We accomplish this by booting a virtual machine off of the same compute gallery image as the executor VMSS, applying patches and/or configuration changes, and then creating a new VMSS image. The VMSS can be configured to use the latest image in a compute gallery, or a specific image version. This update lifecycle is shown in Figure 19.

Figure 19. Maintenance Workflow



Navigate to the Podman Executor Compute Gallery Azure dashboard, select the latest image version, and select "Create VM" to create a new virtual machine based off the current Podman Executor image as shown in Figures 20 and 21.

Figure 20. Create a VM from a VM Image Version

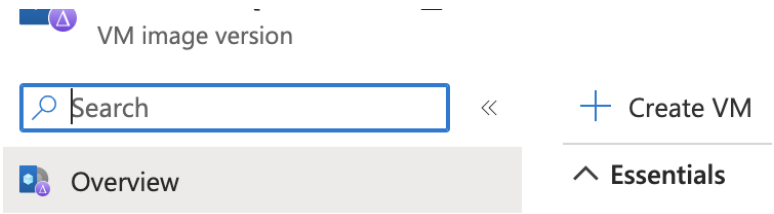
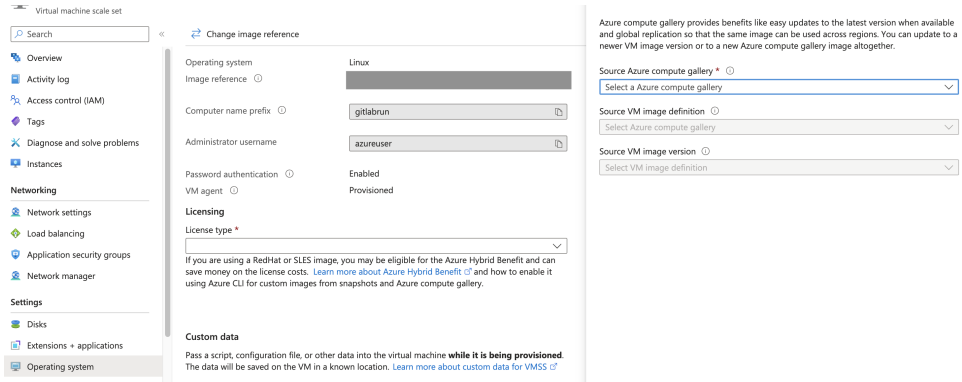


Figure 21. Create a VM from a VM Image Version



Set up the Virtual Machine according to the directions in Section 3.1.1.

Log in to the virtual machine machine and perform any needed system maintenance (e.g., operating system software updates, configuration modifications). After performing system maintenance, shut down the machine, and then you can perform the tasks described in Section 3.3.

Take particular care that if you rotate the limited user password during maintenance that you will need to change the password in the GitLab Runner Manager configuration and restart the manager `gitlab-runner` service.

If the VMSS is configured to use the latest image, then it will seamlessly begin booting new instances with the new image. If you have not configured it to use the latest compute gallery image, then perform the following steps. These configuration steps are given in Listing 24.

Listing 24. Update VMSS Image

```
Navigate to the VMSS dashboard in Azure
Click "Operating Systems"
Click "Change image reference"
Select the Compute gallery, image definition, and image version to use
(or configure to always use the latest version)
```

5. CONCLUSION

By utilizing the limited privileged mode of rootless podman and ephemeral virtual machine scaling, we are able to implement container-in-container operations with strong user isolation at scale. The trade offs between performance and isolation can be controlled at configuration time of the GitLab Runner. This architecture has enabled the implementation of a container factory at scale in multiple multi-user and multi-project environments. This has increased development velocity and shortened research product delivery timelines. This technical report describes an implementation utilizing a particular cloud provider but the principles can be applied to other cloud offerings.

5.1 FUTURE WORK

Oak Ridge National Laboratory (ORNL) has a local OpenStack environment. It may be desirable from a security and/or policy perspective to have scalable runners hosted on-site. While there is no GitLab supported OpenStack Fleeting plugin at this time, Sardina Systems® has published an open source OpenShift Fleeting plugin[7]. We would like to explore the use of this plugin.

[7] https://github.com/sardinasystems/fleeting-plugin-openstack

6. REFERENCES

[1] David Heise. Container factories in the oak ridge research cloud. Technical report, Oak Ridge National Laboratory (ORNL), Oak Ridge, TN (United States), 03 2024. https://www.osti.gov/biblio/2341396.

[2] Chief Information Officer. Dod enterprise devsecops reference design. Technical report, Department of Defense, August 2019. https://dodcio.defense.gov/Portals/0/Documents/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20v1.0_Public%20Release.pdf?ver=2019-09-26-115824-583.

APPENDIX A. GITLAB RUNNER CONFIGURATION FILE

APPENDIX A. GITLAB RUNNER CONFIGURATION FILE

/etc/gitlab-runner/config.toml

```
concurrent = 18
check_interval = 0
shutdown_timeout = 0

[session_server]
session_timeout = 1800

[[runners]]
name = "Podman Runner (Managed)"
limit = 9
url = "<GitLab URL>"
token = "<The Podman Runner Token>"
shell = "sh"
environment = [
"FF_NETWORK_PER_BUILD=1",
"FF_ENABLE_JOB_CLEANUP=true",
"FF_SCRIPT_SECTIONS=true",
"FF_USE_IMPROVED_URL_MASKING=true",
"FF_USE_DOCKER_AUTOSCALER_DIAL_STDIO=true"
]

# Docker Executor config
executor = "docker-autoscaler"

[runners.docker]
# Check the UID for the socket path!
host = "unix:///run/user/1001/podman/podman.sock"
tls_verify = false
image = "quay.io/containers/podman:latest"
privileged = false # Explicitly disable the privileged flag
disable_entrypoint_overwrite = false
oom_kill_disable = false
# We pass in the host container registry configuration as a read-only bind mount so
# that the images hit the container pull through cache
volumes = [
  "/certs/client",
  "/etc/containers/registries.conf:/etc/containers/registries.conf:ro"
]
shm_size = 0
# Autoscaler config

[runners.autoscaler]
plugin = "fleeting-plugin-azure"
capacity_per_instance = 1 # Each job gets its own VM
max_use_count = 2 # Each VM is used for this many jobs
max_instances = 9 # Should equal the limit setting above
delete_instances_on_shutdown = true

[runners.autoscaler.plugin_config]
name = "<Podman VMSS Name in Azure>" # \ac{vmss} Name
subscription_id = "<The Azure subscription ID of the VMSS>"
```

```
resource_group_name = "<The Azure resource group name of the \ac{vmss}>"

[runners.autoscaler.connector_config]
username = "gitlab-runner-executor"
password = "<the local password for gitlab-runner-executor>"
use_static_credentials = true
timeout = "10m"
keepalive = "60s"
use_external_addr = false

[[runners.autoscaler.policy]]
idle_count = 0 # How many idle instances are permitted
idle_time = "30m0s" # How long they are allowed to idle

[[runners]]
name = "Podman-in-Podman Runner (Managed)"
limit = 9
url = "<GitLab URL>"
token = "<The Podman Runner Token>"
shell = "sh"
environment = [
"FF_NETWORK_PER_BUILD=1",
"FF_ENABLE_JOB_CLEANUP=true",
"FF_SCRIPT_SECTIONS=true",
"FF_USE_IMPROVED_URL_MASKING=true",
"FF_USE_DOCKER_AUTOSCALER_DIAL_STDIO=true"
]

# Docker Executor config
executor = "docker-autoscaler"

[runners.docker]
# Check the UID for the socket path!
host = "unix:///run/user/1001/podman/podman.sock"
tls_verify = false
image = "quay.io/containers/podman:latest"
privileged = true
disable_entrypoint_overwrite = false
oom_kill_disable = false
# We pass in the host container registry configuration as a read-only bind mount so
# that the images hit the container pull through cache
volumes = [
  "/certs/client",
  "/etc/containers/registries.conf:/etc/containers/registries.conf:ro"
]
shm_size = 0

# Autoscaler config
[runners.autoscaler]
plugin = "fleeting-plugin-azure"
capacity_per_instance = 1 # Each job gets its own VM
max_use_count = 1 # Each VM is used for this many jobs
max_instances = 9 # Should equal the limit setting above
delete_instances_on_shutdown = true
```

```
[runners.autoscaler.plugin_config]
name = "<Podman \ac{vmss} Name in Azure>" # VMSS Name
subscription_id = "<The Azure subscription ID of the VMSS>"
resource_group_name = "<The Azure resource group name of the VMSS>"

[runners.autoscaler.connector_config]
username = "gitlab-runner-executor"
password = "<the local password for gitlab-runner-executor>"
use_static_credentials = true
timeout = "10m"
keepalive = "60s"
use_external_addr = false

[[runners.autoscaler.policy]]
idle_count = 0 # How many idle instances are permitted
idle_time = "30m0s" # How long they are allowed to idle
```