Secure GitHub workflows with self-hosted runners backed by Azure Container Instances

Problem statement

GitHub provides a great platform for implementing CI/CD for applications, infrastructure (IaC), etc. The platform provides runners for executing CI/CD workflows hosted by GitHub. Though there are benefits of using GitHub hosted runners as it is a managed service; there are a few benefits to self-host these runners:

- For compliance heavy organizations dealing with sensitive data, or mission critical applications, it becomes important to have more control of hardware, OS, and software tools than what GitHub-hosted runners provide.
- Configuring Auto scaling based on business needs would help optimize resource usage.

Self-hosting comes with its own cons though, the biggest one being Operational overhead and managing costs.
If Virtual Machines are used for this purpose, they come with operational overhead and slow boot time. Could probably be an overkill if the workflow jobs are not very compute intensive.

Solution

So, we need a lightweight compute with less operational overhead which can be scaled easily with control over OS and software tools! Containers to the rescue! Azure Container Instances offer the fastest and simplest way to run a container in Azure.

Implementation

Let's start with the startup script that registers the container instance as a GitHub runner.

GitHub Authentication

To register the runner, it needs to authenticate with GitHub. Usually, when a VM is registered as a runner, 2 types of tokens are used alternatively:

1. Personal Access Token (PAT): Token associated with a user account, explained here.
2. GITHUB_TOKEN: At the start of each workflow job, GitHub automatically creates a unique GITHUB_TOKEN secret to use in your workflow. You can use the GITHUB_TOKEN to authenticate in the workflow job.
   When runner is being used for an organizational repository, it becomes tricky to use PAT. The user may leave the organization or the project. So, ideally it is good to avoid this dependency.
   As far as the GITHUB_TOKEN is concerned, though it is the right way to do this, the token expires when a job finishes or after a maximum of 24 hours. This causes a problem. If there is ever a need to reboot the containers, to install updates or upgrade runner module, you must inject the token again from GitHub to the container. Basically, re-create the container. Not very ideal, is it?
   The solution is to generate the token within the container in the startup script. Let's see how we can achieve that:

   1. Register a GitHub app at the organization level (needs GitHub administrator privileges)
   2. Create a script that gets the JWT token to be used:
      a. Generate GitHub App Token. This gives you an APP Id and a private key (pem). Prefer private key over client secret as it is more secure.

b. Use the bearer token to get the access token for the installation of the GitHub App. The Installation Id can be obtained by calling a GitHub REST API or just viewing the installed app in the repository, and copying the ID in the URL.
   https://github.com/<org_name>/<repo_name>/ settings/installations

Refer to the script here:
https://github.com/shwetayadkikar/github-self-hosted-runner-aci/blob/main/GitHubRunner/generateJWT.py

Runner Registration

After successfully getting the JWT token, container must register itself as a runner with GitHub using a REST API. The script below caters to registration of any type of a runner, (repo, org, or enterprise) you need to pass the right scope.

Start.sh

NOTE: A runnergroup parameter has been sent to the config script. This allows a group of repositories to share the runner. So, even if you create an org level runner and you wish to allow only project related repositories to use it, you can restrict the runner to a set of repositories using a runner group.

Docker Image

Docker Image depends on the kind of workload you wish to run on the runner.

For example, if you want to run PowerShell to interact with Azure, then install Azure PowerShell modules. The important thing is to ensure we set the entry point to the start.sh script. Here I have created a Linux based container, you could create the same code on a Windows-based container and startup script can be written in PowerShell. Here is the reference to a docker image I used: https://github.com/shwetayadkikar/github-self-hosted-runner-aci/tree/main/GitHubRunner/Dockerfile

Upload docker Image to Azure container registry

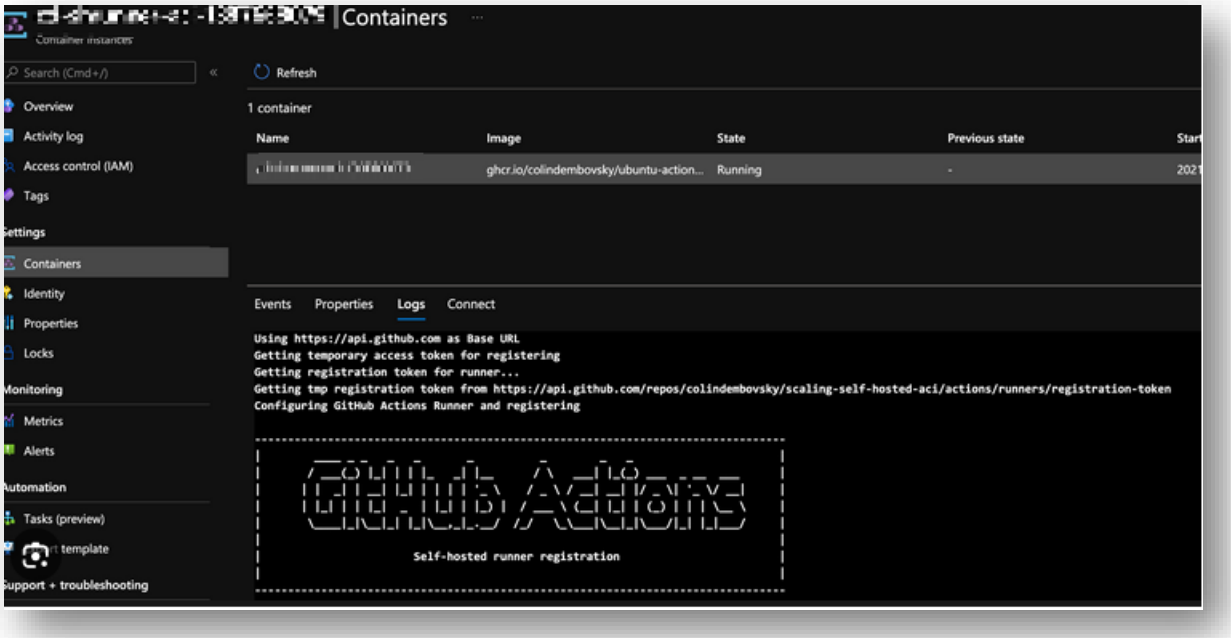
We create Azure Container Registry resource and build and upload the above docker image. Refer to this GitHub workflow for the code. The GitHub workflow should use a managed Identity or Service principal to authenticate to Azure with OIDC using federated credentials.

Create Azure Container Instance

Run following command to create azure container instance with the image published on the ACR. Create multiple instances to increase the availability of the runners, refer this.

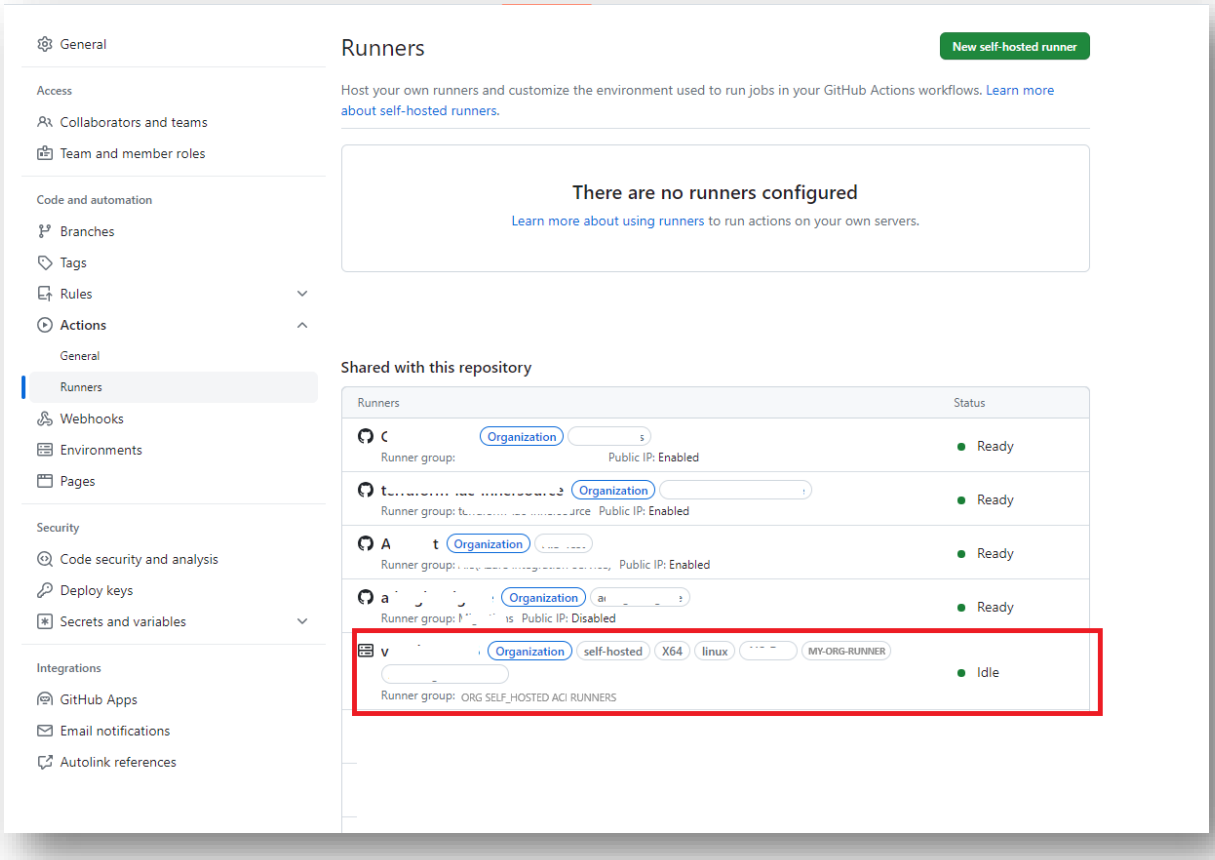
```
az container create \
--name github-runner-demo \
--resource-group acirunnerdemorg \
--image acirunnerdemoreg/githubrunner:${{ github.run_id }} \
--registry-login-server acirunnerdemoreg \
--registry-username ${{ secrets.SPN_APP_ID }} \
--registry-password ${{ secrets.SPN_SECRET }} \
--dns-name-label github-runner-demo-$RANDOM \
--query ipAddress.fqdn \
--subnet aci-subnet-1 \
--subnet-address-prefix 10.0.0.0/24 \
--vnet aci-vnet-1 \
--vnet-address-prefix 10.0.0.0/16 \
--environment-variables 'GH_APP_ID'='${{ secrets.GH_APP_ID }}'
'GH_PRIVATE_KEY'='${{ secrets.GH_APP_ID }}'
'GH_APP_INSTALLATION_ID'='
 'RUNNER_ORGANIZATION_URL'='https://github.com/shwetayadkikar
 'RUNNER_NAME'='github-runner-demo-$RANDOM'
```

After booting up, container should execute the start up script and register itself as a GitHub runner.



You should be able to see the runner in the runners' section in your repository:

Secure GitHub workflows with self-hosted runners backed by Azure Container Instances

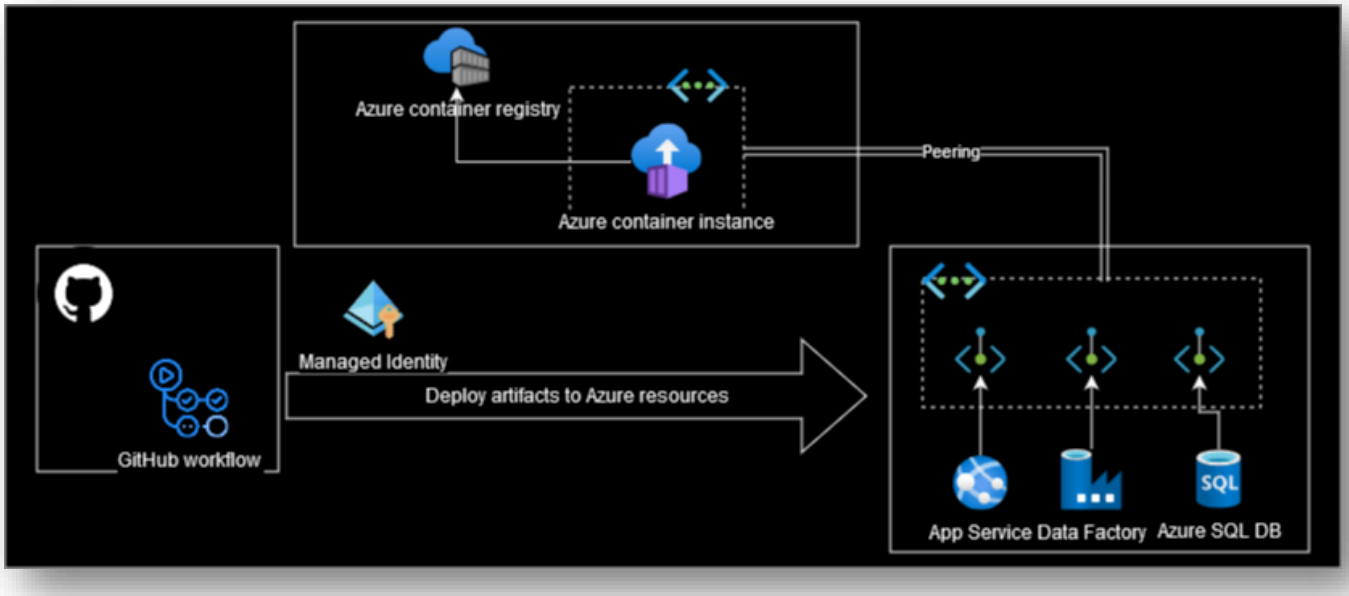


Reference the runner in you GitHub workflow by addressing it with the label:

```
runs-on: [MY-ORG-RUNNER]
```

Note that we have given reference to the virtual network this container group will reside in. This takes us to the next segment of this story: private connectivity.

Connecting to Azure resources from GitHub over a private connection



Now that we have deployed the container in a virtual network, it can be used to deploy artifacts to the Azure resources **that have public access disabled.**

- You can deploy the ACI in the same network where the private endpoints for the Azure resources have been created OR
- You can have the ACIs in a separate network and peer it to your azure resources private network.

Ensure to further harden the network security by using Network security groups and/or Azure Firewall to regulate the ingress and egress traffic. You can also secure your Container registry with restricting the network access and following the security baseline.

Challenges during implementation

1. Reboot Issues
    a. Initially, I was using GITHUB_TOKEN, which would get expired after 24 hours. If container rebooted after that, it could not register with GitHub giving unauthorized error. The solution to it as mentioned is to generate the token during startup.
    b. Another challenge was about the name of the container. Every time the container boots up, it registers itself with a name. If the container has already been registered with that name and has not been cleaned up, then the registration API returns an error saying that 'Runner with {name} name already exists.'
    Solution here was to register the runner with a random name. Hence, in the startup script I have added a randomly generated string as a suffix to the runner's name, ensuring it registers with a new name on reboot.
2. Offline runners' cleanup
    The offline containers had to be cleaned up. I created a GitHub action that would use GitHub CLI to de-register the offline runners. Ref: Workflow.

Summary

- We can leverage Azure container Instances to deploy artifacts to Azure PaaS resources, which deal with mission critical applications or sensitive data and connect to these resources over a private link from GitHub.
- By ensuring usage of GitHub App Tokens, OIDC for authentication with Azure we further enhance the security of this implementation.
- Using Azure Firewall and NSGs we can control the inbound-outbound network traffic and harden the security of the infrastructure.