PrestoChango Project

**\*Precursor\***

First, I want to say I made a conscious decision to attempt to complete the project with technologies I virtually have no work experience with. I wanted to challenge myself to see how quickly I could learn both Docker and AWS platforms along with also trying to automate against them. Sorry to say I was not able to put together a fully working build and deployment “pipeline” in the time that is recommended for this exercise. With that said, I was able to create some infrastructure, and learn the commands in order to enable me to automate them in a process (more on that in my “What if Jim had more time” section) and was able to hit my presto container via public IP. What I created was an EC2 instance instead of an ECS Container instance to complete the project. I started with an ECS instance and had a container running but struggled to automate that process. I needed more time to learn how to automate against the ECS Container instance to better create the pipeline I had in mind.

With the EC2 instance created, I was able to install Docker and login to me ECR repo which contains the Presto image I was able to build and push up to AWS. Needless to say, I did go past the 3-5 hour time box and roughly spent 12-13hours. The majority of that was learning how to create the instances in AWS as well as setup the security group, create my access keys, and connect to a remote linux instance using powershell.

**\*Initial Task - Pipeline Creation\***

As stated early, I created an EC2 instance following this guideline to create a linux host and install docker: (Not ideal, but accomplishes my “MVP” pipeline)

<https://www.youtube.com/watch?v=IDUyROEXnNA>

I am looking at this strictly as a development pipeline and not one intended to be used in a production environment. Objective is to increase the feedback loop for the developer to aide in giving them more agility and increase velocity. (“Extend Further” will cover taking the pipeline further up the development and deployment chain)

I wanted to take a different approach from selecting a particular branch to initiate a build, and instead just allow the developer to perform a commit which would trigger “build” steps to create the docker image. The docker file itself contains the version of the image that would be ultimately built. The build process would inspect the docker file for the version number and execute a command to build the image and tag it with either the version number itself or the version number plus branch name (ex. – tag=312-e.7-mybranch). Once built, the image is then pushed to the ECR repo, where then any image could be deployed by specifying the “version” tag.

I didn’t want to do a lot of manipulation at the code repo level and rather use webhooks or triggers to initiate the docker image build, push to repo, then deploy based on image tag, which could be an input variable value. This would constitute an “MVP” (minimal viable product) for the developers to leverage by just having to push a commit, and then inputting what version image to deploy. The deployment and build would be handled by an orchestrator where variables can be stored and substituted in the scripts to select the “version” along with type environment specific values(preprod and prod).

My “MVP” state is a manual code pull from the repo to a directory from which I then execute the tasks to build, push, and then deploy the Presto Container to AWS ECR and EC2. I didn’t have enough time to complete the automate Octopus tasks, but I have all code that the tasks would run uploaded in my repo for review and listed here in this document.

PreReqs:

* Docker client installed on local workstation
* PowerShell Module “Posh-SSH” installed for connection to EC2 instance
* AWSPowershell module installed and credential profile created

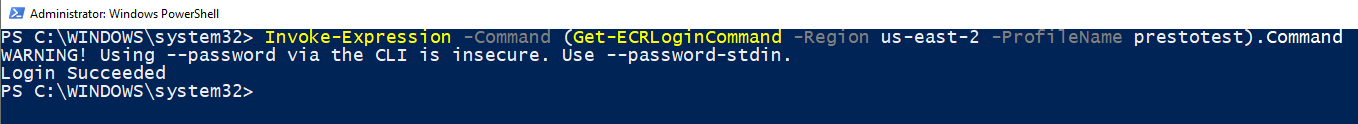
## Quick Overview of Steps:

1. Clone Presto repo to local workstation
2. Trigger Octopus Deploy to build and deploy the Presto Container.

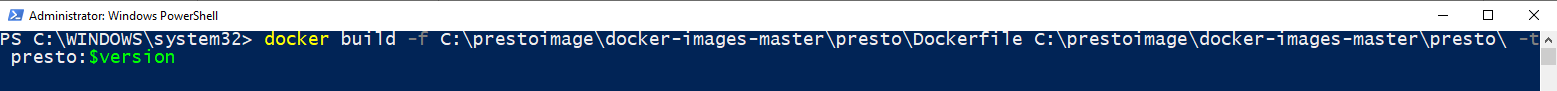
* Octopus runs command to log into AWS ECR registry – Task 1
* Task runs to build docker image from local repo path – Task 2
* Task runs to tag the image with variable substitute – Task 3
* Task runs to push to AWS ECR repo with variable tag – Task 4
* Task runs to log into AWS ECR repo from EC2 instance – Task 5
* Task runs to pull presto image with variable version tag – Task 6
* Task runs to run the docker container specifying ports and variable version – Task 7
* Task runs “Invoke-WebRequest” against EC2 Container instance expecting status code “200” for successful deployment. If not “200” then deployment fails. (Health-check test) – Task 8

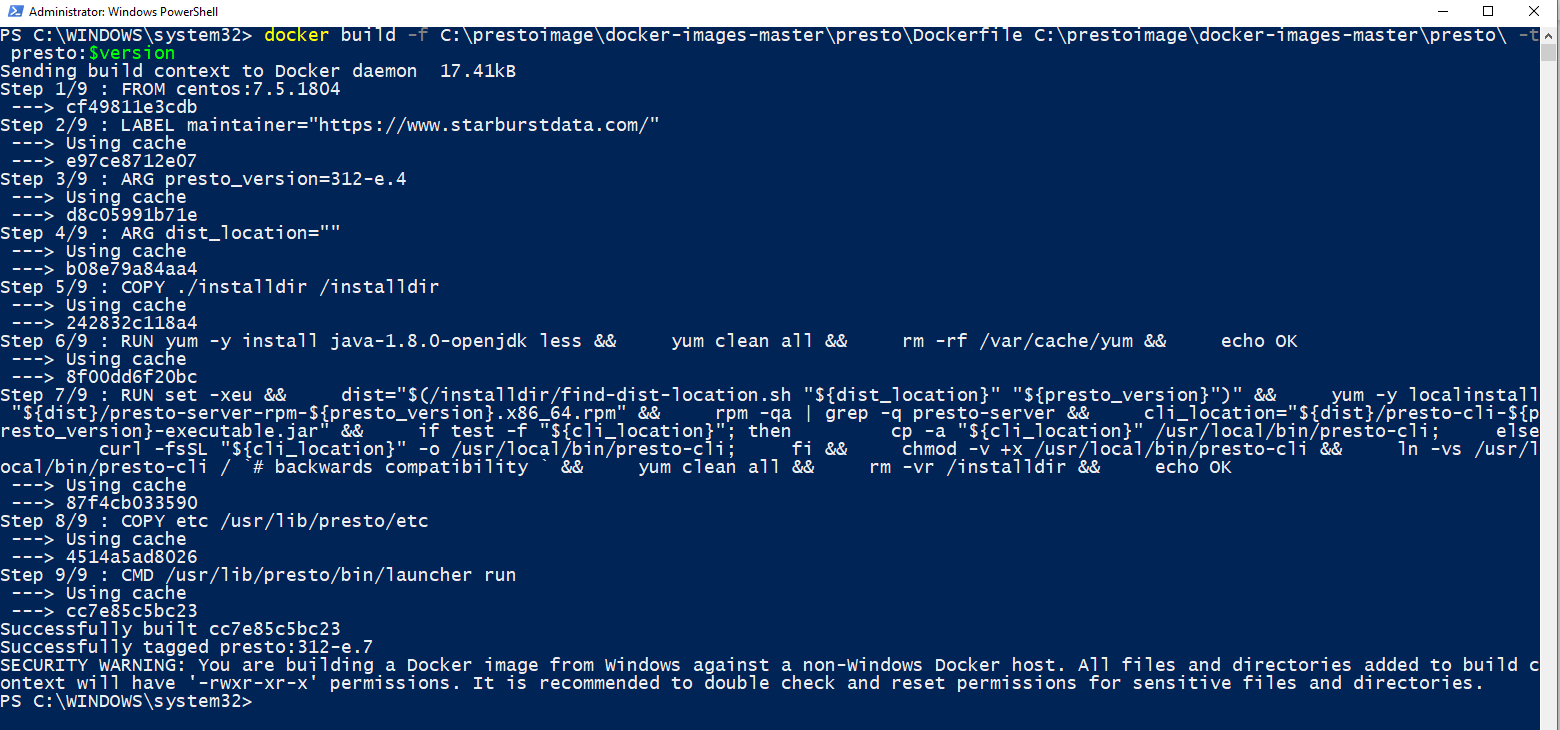
Screenshots of each task run manually:

Task 1: (Aws Credential was built previously to this step)

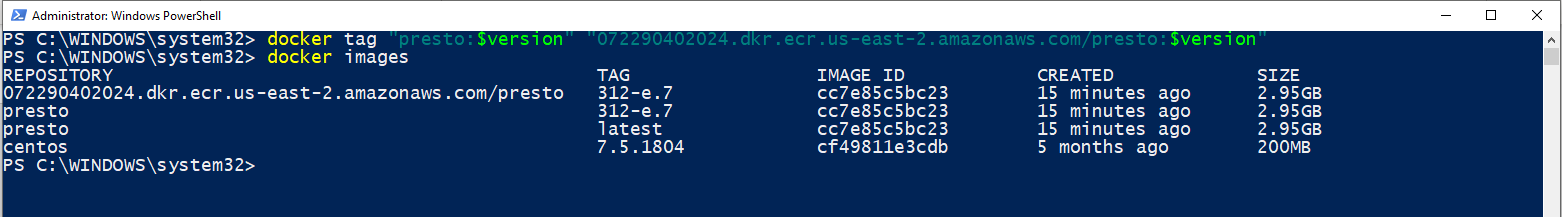


Task 2:

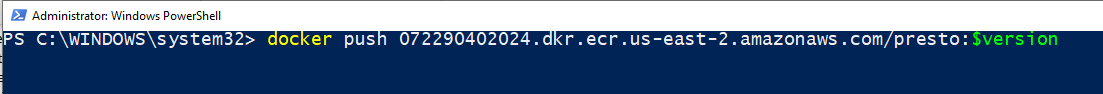




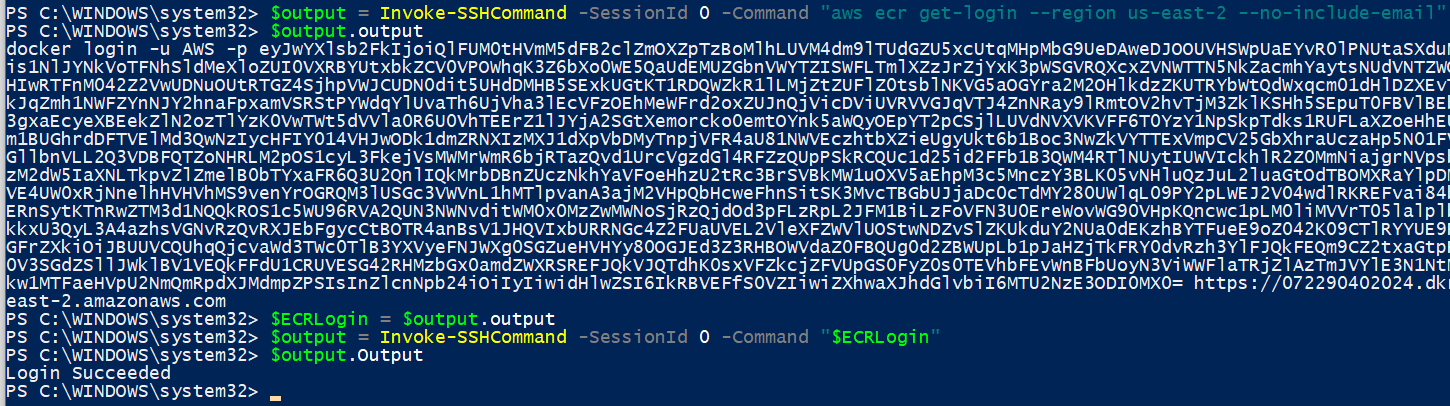
Task 3:



Task 4:



Task 5: This uses the Posh-SSH module and a session is established before running this task. Will elaborate more on this further in “extend further” section:



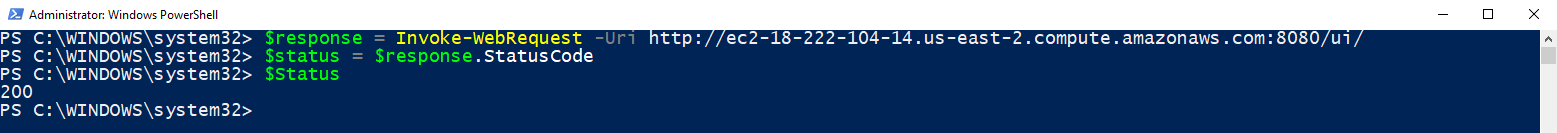
Task 6:



Task 7:



Task 8:



**\*Extending Further\* (If Jim had more time)**

Building on what has been created so far, my ideal pipeline would work something like this:

Before Merge to “master”

Build

1. Developer initiates a PR and triggers build process
2. Build performs static code analysis and publishes results and must pass quality threshold
3. Build executes “Unit Tests” and publishes results and must pass quality threshold
4. Build Process builds the docker container with version/branch tag
5. Build pushes the image to “Test” repo
6. Build triggers “deploy” orchestrator for next steps
7. Build packages infrastructure manifest and pushes to orchestrator

Deploy

1. Deploy orchestrator executes infrastructure provision step which builds out AWS infrasture needed for container deploy (load balancing(if needed environment driven), ECS Instance, Security groups, etc)
2. Deploy orchestrator then runs container
3. Deploy orchestrator then performs health-checks to confirm container is operational
4. Deploy orchestrator then triggers test runner

Test Runner

1. Test runner executes Integration tests and or regression testing (if needed, In my practices it has been customary to perform this against mock data, services, or whatever the integration points are)
2. Test runner then publishes test results and either passes or fails the pipeline based on results.

Assuming all the above tasks succeed then the pipeline will then merge code in to the “master” branch of the code repository then execute the above tasks again, but now against the master branch. Once complete it then instead puts the container image in a “preprod” repo. Where further testing may be needed before deciding to promote the container version to the “production” repo. This may raise concern for resources of storing the almost 3GB container in AWS, so I would ensure retention policies would be put in place limited how many versions are kept in the “test” and “preprod” repo based on deployment cadence and or business need. The above tasks could be orchestrated by one or possibly two tools.

Most of the “CI/CD” tools all accomplish the same concepts as above, the difference is how they would execute those tasks.

Another note – any secrets, api keys, passwords, or other sensitive data would be stored in a key vault solution that automation above would leverage and call into to retrieve any said sensitive data points.

I will finish with this, it was unbelievable to me how vast the AWS library of services is and the possibilities that platform can offer. This was my first attempt at using this, and at first it was very overwhelming at first, but I was able to manage to achieve a little bit of my objective which did feel rewarding. I was able to script against two platforms I have had virtually zero experience with, and my mind was racing with all the things I could possibly achieve doing this project. Which did start to lead me down some rabbit holes, but I was able to stay focused and fall back on some of my Agile training of trying to deliver an MVP instead of the ENTIRE solution. Which is almost unattainable as requirements and technology changes, so delivering in small increments is the approach I use to attack problems or taking on large projects.

**\*Time Spent\***

As stated earlier I spent roughly a total of 12-13hours on this project, not including this documentation.

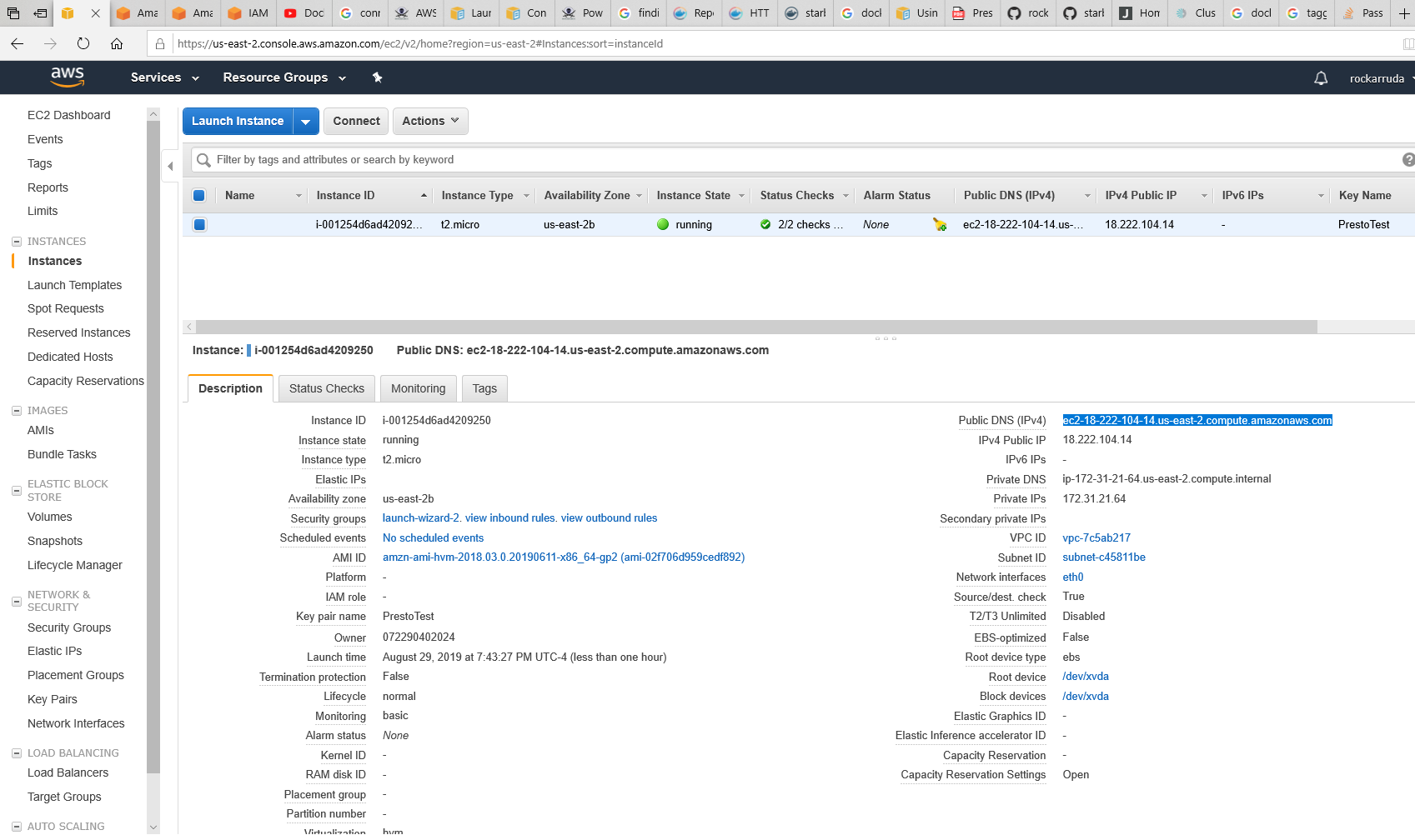
Breakdown:

AWS discovery and learning with build infrastructure – roughly 10hours

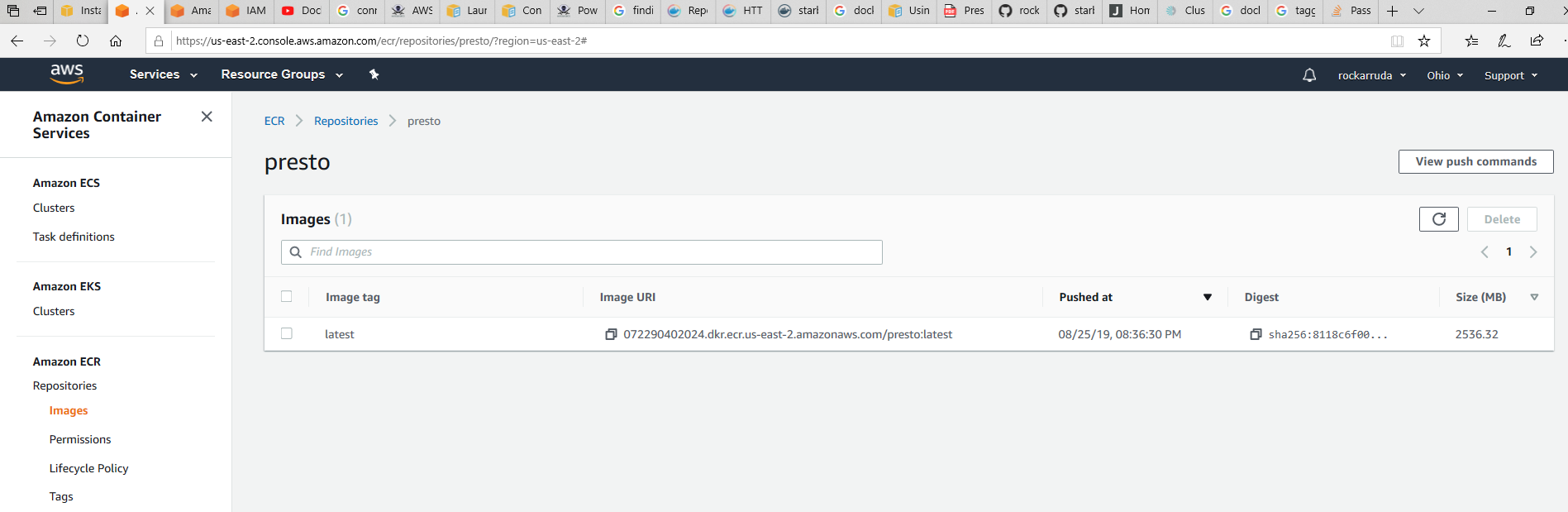
Scripting against linux and setting up windows workstation to manage linux endpoint – 2 hours

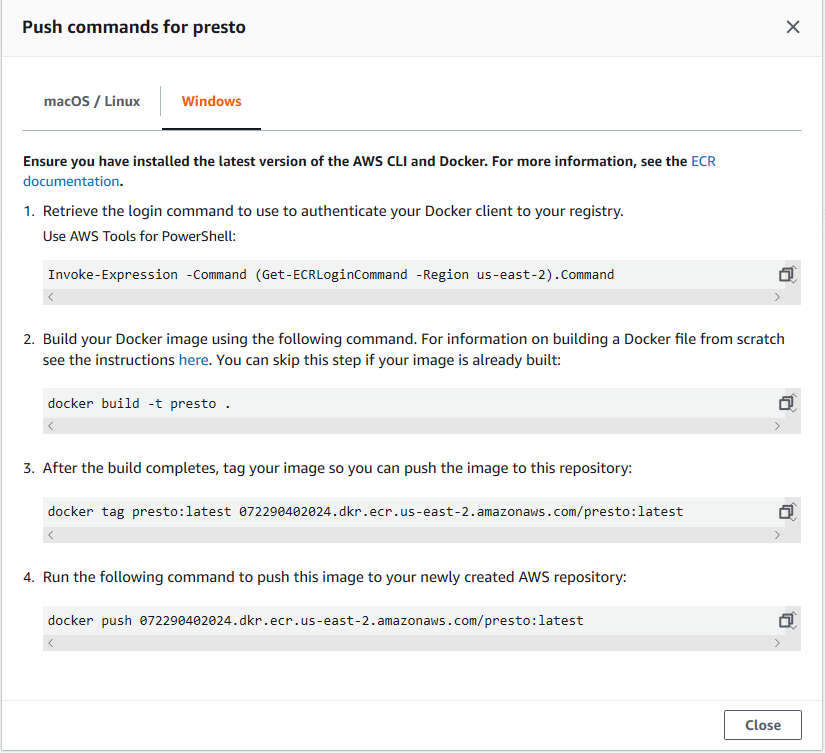
**\*Evidence\***

**AWS EC2 Instance:**

****

**ECR Repo:**

****

****

**Amazon Resources: (Basically A LOT of Amazon docs) Some are below**

[**https://docs.aws.amazon.com/AmazonECR/latest/userguide/ECR\_AWSCLI.html**](https://docs.aws.amazon.com/AmazonECR/latest/userguide/ECR_AWSCLI.html)

[**https://docs.aws.amazon.com/powershell/latest/userguide/pstools-appendix-sign-up.html**](https://docs.aws.amazon.com/powershell/latest/userguide/pstools-appendix-sign-up.html)

[**https://docs.aws.amazon.com/AmazonECR/latest/userguide/docker-basics.html**](https://docs.aws.amazon.com/AmazonECR/latest/userguide/docker-basics.html)

[**https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html?icmpid=docs\_ec2\_console**](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html?icmpid=docs_ec2_console)