LA-UR-23-23879

Approved for public release; distribution is unlimited.

Title:	HPC Cluster CI/CD Image Build Pipelining
Author(s):	Cotton, Travis Bradley
Intended for:	Cray User Group, 2023-05-07/2023-05-11 (Helsinki, Finland)
Issued:	2023-04-17 (Rev.1) (Draft)

HPC Cluster CI/CD Image Build Pipelining

1st Cotton
HPC at LANL
Los Alamos National Laboratory
Los Alamos, USA
trcotton@lanl.gov

***Abstract*—Building images for HPC clusters tends to be a monolithic process, requiring complete rebuilds when new packages or configurations are added to them or updating existing images. It is also generally a manual process, heavily involving a system administrator and system-specific custom tooling. Rebuilding images from scratch can be time consuming and updating existing images can introduce unwanted/unexpected changes to production systems. These problems can be mitigated by using existing container models, creating and layering images to create the final "production"-ready results. This allows for rapid turnaround and a guarantee that existing layers remain unchanged while safely updating others. We can then leverage this layer-based image building to allow for a more automated process using Continuous Integration/Continuous Delivery (CI/CD) pipelines. Leveraging standard tools for configuration management and version control combined with the OCI standard of layer-based image building and CI/CD pipelines, we can create an automated and even distributed image building workflow while still being customizable for specific sites and systems.**

***Index Terms*—CI/CD, pipeline, image building**

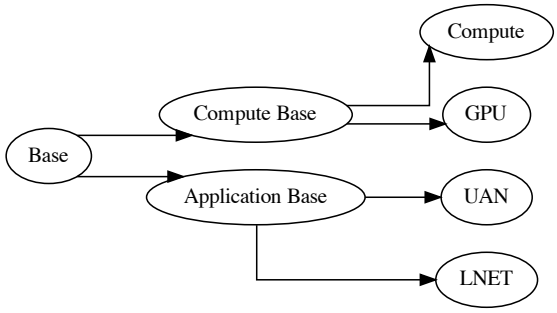
I. Introduction

Los Alamos National Laboratory (LANL) has significant HPC resources including generic clusters and HPE Cray EX systems utilizing the CSM system management environment. The image building technology described below is being developed to deal with the increasingly complex environments and are intended for deployment in CSM and generic cluster environments. In this paper, we will discuss the two major pieces of CI/CD image building; the tools required for building images in layers, and a pipeline configuration to automate these builds. Linking these two pieces together can create a flexible and portable cluster image building process that can have a myriad of positive effects when managing a complicated set of clusters.

II. Layered Image Building

The core piece of this workflow is the ability to build cluster images in pieces, or layers. A layer can be defined as a distinct part of an image that can be created from a combination of a parent layer and the configurations that define this layer. The parent layer in this context can either be a hard dependency, a way to manage distinctness between layers, or an ease-of-use practice for a site specific reason. An example of a parent-child layered configuration can be seen in Fig. 1. The configuration step just provides the content for this layer, which can be as simple as a list of packages to install or a more complicated set of operations like configuring files and templating.

Fig. 1. =
Layer Dependency Example.



A. Tools in Use

This kind of workflow naturally lends itself to leverage a container-based build. Being able to import a previous layer as the parent, and then modify the current layer are all things container technologies can already do. There are a number of implementations like Docker [1] or Podman [2] that can import from a perviously built container image, but in this workflow we will be using a dockerfile-less approach, so our tool of choice is Buildah [3]. Buildah can easily spin up a container on the fly, using an existing image (i.e. the parent) or a scratch container. We can then leverage this container as a starting point when configuring the target layer. Once configuration is complete, the target can be exported into a number of formats.

Configuring a layer can be a generic operation, ranging from installing base packages to setting configuration files, which can be specific to the kind of layer the target is. If the idea is to build a "base" layer, one which all future layers import from, then a simple package install into a scratch container can accomplish this. An example using the DNF package manager and the Buildah command line tools can be seen in Listing 1. If something more complicated is needed, then a configuration management tool can also be used.

Once this new container is up and running, Ansible or a similar configuration management application can be run against it. We will be using Ansible in our case and specifi-

```
#!/bin/bash
CNAME=$(buildah from scratch)
MDIR=$(buildah mount $CNAME)
dnf --installroot=$MDIR groupinstall base
buildah umount $MDIR
buildah commit --rm $CNAME my_base_image
```

Listing 1. Buildah cmdline example.

cally, we will be using the python libraries Ansible provides. This will allow us to programmatically run Ansible against our containers. We can dynamically add the layer container to our inventory, with any Ansible grouping we wish to employ, along with the Buildah connection plugin [4]. The Buildah connection plugin will replace the default SSH behavior Ansible uses so that we can treat the layer containers as Ansible hosts. Any playbooks configured with the layer container as a host, or any groups the layer container has been added to, will now configure the layer container.

You can continue to build more layers from previous layers until you have a final image. There are many possibilities once you have a final container image, one of which is to mount this container and leverage Squashfs to package up the image. Squashfs images are very versatile and can be used in many booting schemes. Since these are container images, it will also be possible to push to an external registry, making built cluster images available for future use.

An import piece of the containerized image build is the ability to build layers as a non-root user. This can be accomplished in a single podman instance or a kubernetes deployment. The non-root user is built into the builder container image and some capabilities are needed, namely the cap_setuid=ep and cap_setgid=ep for the newuidmap and newgidmap commands respectively. These capabilities are set inside the builder container image and not the host machine. If you are going to use a local storage then this will need to be writable by the non-root user and if you push to a registry the non-root user will need credentials to do so. Some buildah commands require the mount command, and this is disabled with AppArmor by default, so AppArmor will need to be set as unconfined or a new profile will need to be created. We did not try this with selinux, but a similar workaround may be needed. A final necessity is using the buildah unshare command for all container operations. The functionality of buildah unshare is to run commands under a new namespace allowing mounts as a non-root user inside the builder container. This is the current setup for building cluster image layers as a non-root user, but there may be other options for minimizing capabilities and restricting UID and GID mappings that we have not yet explored. Many of these capabilities and exceptions are workarounds for installing packages and other traditional privileged operations, but further configurations can be explored and an excellent starting point can be seen in [8].

III. CI Pipeline

The pipeline aspect of this project is the most versatile piece of CI/CD image building. There are many, many CI runner options for git alone, disregarding other version control systems. We will focus on using gitlab-runners [5], and hopefully many, if not all, of the pieces here are translatable to other software stacks.

```
stages:
  - full
  - layer1
  - layer2

inventory-job:
  stage: full
  script:
    - image-build base
    - image-build layer1
    - image-build layer2
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - inventory/**

layer1-job:
  stage: layer1
  script:
    - image-build layer1
    - image-build layer2
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - < list of layer 1 roles >

layer2-job:
  stage: layer2
  script:
    - image-build layer2
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - < list of layer 2 roles >
```

Listing 2. Single Repository Pipeline Config. A very simple example that will start an image build pipeline based on where changes occur in the repo. An inventory change will build the base, layer1, and then layer2, while changes to layer1 files will only build layer1 then layer2. Changes to layer2 files will only build layer2 and import a previously built layer1. The rules here will only run the pipeline if this is a merge request.

A. Pipeline Examples

The most simple example is to have a single repo that contains all the configurations needed to build cluster images. This repository will house the Ansible inventory, roles, and playbooks alongside any configurations needed to build a base image. In this example a single file with a list of packages and package repositories (yum, zypper, etc) will define our base layer. Our pipeline configuration will be a single .gitlab-ci.yml file inside our image repository as seen in Listing 2. In this example we have a simple three layer setup: base, layer1, and layer2. This configuration will key off changes that occur in specific places, defined in the 'changes' stanza. The gitlab CI pipeline will start the pipeline at the first detected change. We have to carefully decide here what changes belong in each layer stanza. If each layer is defined by a playbook, and each playbook is a list of roles, then we can get our list of changes directly from the playbooks, but this is not strictly necessary. Changes to inventory files is more complicated, and in this example inventory changes will kick off a build starting from the base layer.

The next example is more complicated but still relatively easy. In this case the base, the inventory, and each layer are housed in a separate repository. The idea is similar, but now we need to tie the layers together in the desired order. This is

```
stages:
  - build

base-build:
  stage: build
  script:
    - build-image base
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - inventory/**

trigger-next-layer:
  stage: build
  rules:
    - if: $CI_MERGE_REQUEST_ID
  trigger:
    project: '/imgbuild-project/layer1'
  needs:
    - job: base-build
```

Listing 3. Inventory Repo Change. A change to our inventory will kick off a full rebuild. In this case the inventory is a separate repository, but can still trigger downstream layers with the 'trigger' keyword. If no changes to the inventory directory, or any subdirectories or files occur, then the pipeline will not start an image build.

done with the 'trigger' keyword, which will start the next layer build if changes are detected in the current layer. See Listing 3 as an example for an inventory change and Listing 4 for an example of a change in a layer. In the inventory example, we again make the assumption that a change in the inventory implies a complete pipeline rebuild. In the layer example, the build stanza will only run if the specified changes occur, but now will trigger the next layer if any changes are detected. If no changes are made then nothing will be triggered. The current layer will also need to be able to run if it is triggered by a parent pipeline, which is accomplished in the 'when' keyword. There are several pre-filled gitlab variables available for use [6].

B. Automated Pipeline Generation

As can be seen by our so-called simple examples, setting up a CI pipeline can be a complicated endeavor and requires careful thought. It is possible to manually create pipelines for each cluster image type but the implementation of them will depend on the structure of the git repositories where image layers are built and configured from. There are many other even more complicated ways to organize cluster image configurations and all these would require a separate pipeline schema, which is the kind of rigidity we would like to avoid. Our solution to this is to create an automated workflow that can handle many types of repository setups while still maintaining the layered image build approach.

The first step is to auto-generate the pipeline. Instead of writing each step of the pipeline manually we generate it based on a configuration file that defines how a layer inherits from a parent and where it gets its configuration data from. We can think of our cluster images as being defined by a directed acyclic graph (DAG), where the nodes are our defined layers and edges define the parent-child relationship. For the implementation of our DAG we use the python library networkx [7]. This allows us to "attach" information to each node, like repository data, image types, which playbook to use for this layer, etc. These DAGs are defined in a simple YAML

```
stages:
  - build

build-merge-request:
  stage: build
  script:
    - build-image layer1
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - < list of layer1 roles >

build-pipeline-trigger:
  stage: build
  script:
    - build-image layer1
  rules:
    - if: '$CI_PIPELINE_SOURCE == "pipeline"

trigger-job:
  stage: build
  rules:
    - if: $CI_MERGE_REQUEST_ID
    - if: '$CI_PIPELINE_SOURCE == "pipeline"
  trigger:
    project: '/imgbuild-project/layer2'
  needs:
    - job: build-merge-request
      optional: true
    - job: build-pipeline-trigger
      optional: true
```

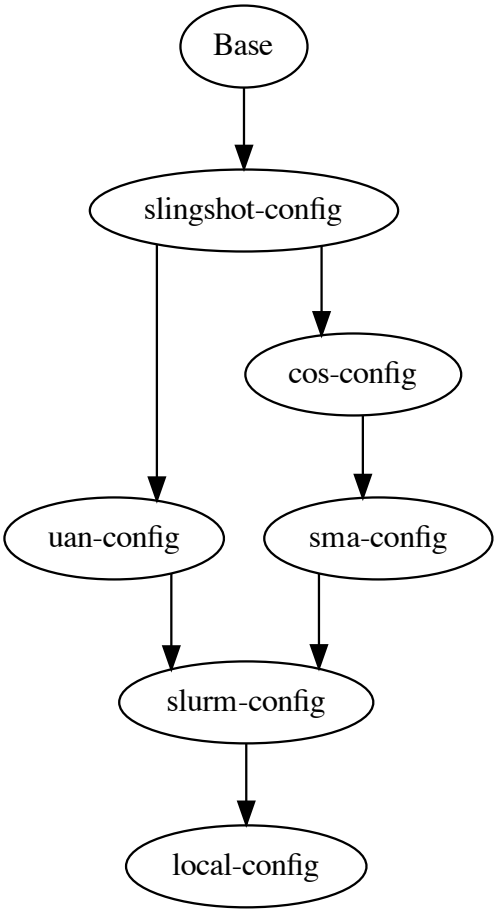
Listing 4. Layer Repo Change. Similar to the Inventory repository, our Layer1 repository will start on a merge request but additionally it will also start if triggered from another pipeline. If triggered from another pipeline we no longer care whether files have changed or not since the parent layer is assumed to have been updated. Once, and only if, the Layer1 build succeeds will we trigger Layer2.

```
inventory:
   path: 'cluster-inventory'
   cluster_name: 'clusterA'
   cluster_name_short: 'ca'

my-fav-os:
  image_types:
    - {type: 'base', firstlayer: True, children: ['hsn']}

hsn:
  repo: 'hsn-config'
  image_types:
    - { type: 'base', groups: ['HSN'], pb: 'hsn.yml',
        ↪ children: ['compute','uan']}

compute:
  repo: 'compute-config'
  image_types:
    - { type: 'compute', groups: ['Compute'], pb: 'comp.yml
        ↪ ', children: ['slurm']}

uan:
  repo: 'uan-config'
  image_types:
    - { type: 'uan', groups: ['UAN'], pb: 'uan.yml',
        ↪ children: ['slurm']}

slurm:
  repo: 'slurm-config'
  image_types:
    - { type: 'compute', groups: ['Compute'], pb: 'slurm.
        ↪ yml'}
    - { type: 'uan', groups: ['UAN'], pb: 'slurm.yml'}
```

Listing 5. Image Layer DAG definition. A layer can have multiple children and multiple image types.

file, an example can be seen in Listing 5. This example is a very simple layer dependency DAG, but a graph shown in Fig. 2 maps out what a CFS-like DAG might look like. Each layer has a repository defined but there is no restriction on layers having the same repository source. On one extreme end you can have a single repository for every layer, and on the opposite extreme have a separate repository for layer.

Fig. 2. =
CFS-like DAG. This is the logic the automated pipeline generator will use when building the pipeline out. The flow of the actual pipeline will depend on the changes attempted and the child-parent dependency illustrated here



The next step is to write the pipeline based on the changes made to a layer's configuration repository. This is harder than it may seem at first. A layer is defined by a playbook, roles, and the inventory associated with the layer. The inventory data can come from a combined source (for all systems) or a separate repository for each system at a site. When a change occurs in a role, we can find where to start the pipeline by searching through each image type's playbook(s), looking for matches between changes made and the roles/tasks defined in the playbook(s). If a match is found, then we can start the build chain at the target layer. This is relatively easy especially when compared to changes in inventory data. Inventory changes are much harder to map to not only a specific layer, but a layer for a specific image type. To accomplish this we need a way

```
# t = target layer, G = graph, c
InventoryCheck(target, G, changed_files, layer_list)
    for cf in changed_files
        variables = getVariablesInFile(cf)
        roles = getRoles(t.playbooks)
        for v in variables
            found = False
            for r in roles
                found = lookupVar(v, r)
                if found == True
                    layer_list.append(target)

    children = getChildren(target)
    if len(children) > 0 and not found
        for c in children
            inventoryCheck(c, G, changed_files)


# read in layer DAG
G = digraph(my_layer_config.yaml)
# start from root of DAG
target = "root"
# get a list of changed files
changed_files = read("changes.git")
# empty list to hold layere matches
layer_list=[]

InventoryCheck(target, G, changed_files, layer_list)
```

Listing 6. Pseudo code for finding variables in roles listed in a playbook.While not a full example, this should illustrate the idea of mapping inventory changes to a layer by moving through the dependency DAG and searching playbooks associated with each layer.

to map inventory variables to the roles defined in a layer's playbook(s). See Listing 6 for the pseudo-code for finding a variable in the roles listed in a playbook. Inventory changes are mapped to roles by reading in the changed files, and for each variable contained in changed files, searching each role defined in a layer playbook. If a match is found, then all downstream layers can imply the need to be rebuilt and the build can start with the current layer. If no matches are found in the current layer, then we search the next layer in the DAG. Since a layer can contain configurations for multiple image types it can be possible for a build chain to get created only for a specific image type, while other types do not get rebuilt. In its current form, inventory changes may result in unneeded image builds as we key off file changes, and any variables in a changed file may induce an image build. We currently do not key off explicit variable changes in a file.

Using this information the staging repo will build out the pipeline automatically starting from the first layer affected by the changes made in the originating repository. This is done by traversing the DAG and searching for matches between changed files and files that define a layer. In the roles case, each node in the DAG is a layer and has an attached list of playbooks, so we search the roles/tasks defined in the playbooks for changes. For an inventory change we must map the variables in the changed files to roles in the playbooks, and if a match is found we start with the earliest layer in the DAG. If a change occurs that only affects a single image type, Compute for example, then our pipeline will only build a Compute type image starting from the earliest affected layer,

Fig. 3. =
Single image type pipeline visualized. The change occurred in the COS layer, affects the Compute type image, and so the pipeline start in this layer for this image type. No other image types were affected and so are not rebuilt. This is visualized oval-dotted layers not being built, while the boxed layers are the target rebuilds.

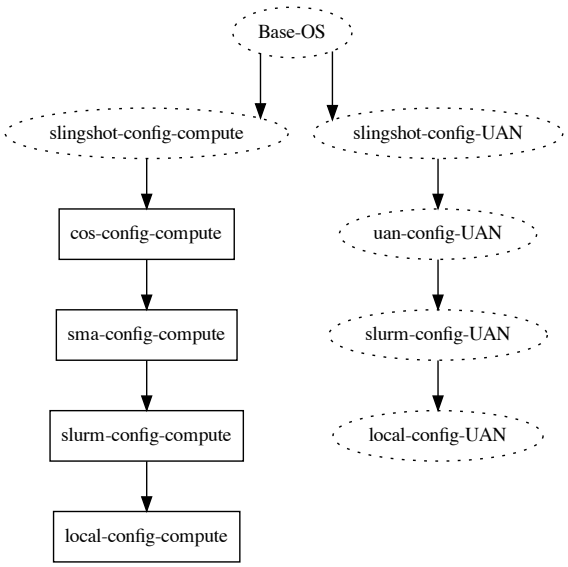
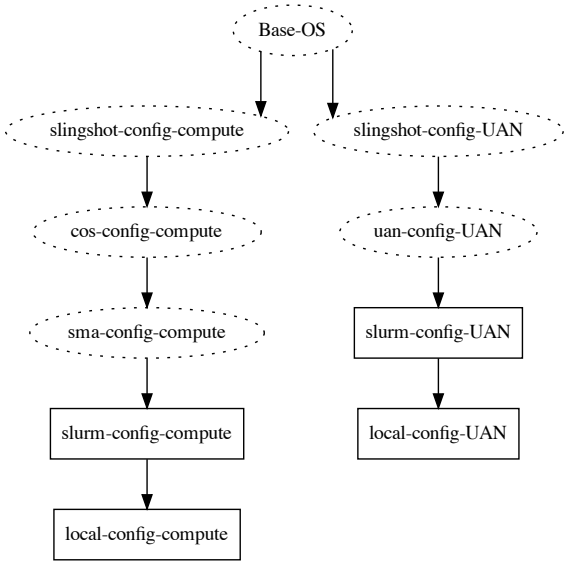


Fig. 4. =
Multiple image type pipeline visualized. The change occurred in the Slurm layer and affects multiple image types. In this example the two affected image types were compute and UAN type images. The pipeline will not rebuild unaffected layers, represented by the dotted-oval shapes, while the Slurm layer for each image type will be rebuilt and all downstream layers.



COS in this case Fig. 3. If a change occurs that affects both Compute and Application_UAN, say a Slurm package update, then the pipeline will build both image types as seen in Fig. 4. Figures Fig. 3 and Fig. 4 visualize what the pipeline will look like after processing the changes made and how they affect our dependency DAG, so it is possible to have a single node graph if the changes only affect the very last layer of an image.

IV. Observations and Benefits

Building images in layers, and a clever schema for organizing layers, can have a number of useful benefits. It is not uncommon for a site to have multiple systems which share many of the same features including the OS release, software stack(s), libraries, etc. A layered build approach allows us the ability to share layers across multiple systems which can reduce the time it takes to push out changes to production systems and forces consistency across systems.

A. Turnaround Time

Changes made to image configurations no longer require a full rebuild, depending on what kind of change is made. A change to the last layer of a specific image will not require rebuilds of parent layers and will not require rebuilds of different image types if those changes do not have an effect on those layers. So it is possible to make a change that updates a compute type image, rebuild the image starting from only the changed layer, and leave all the parent layers unchanged. This can speed up the time it takes to push out changes.

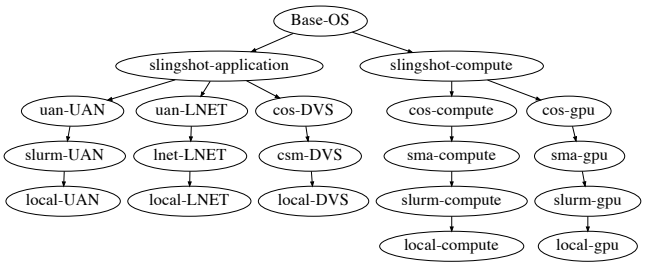
Another type of change is a product change. For example a Slingshot update will have an effect on downstream layers, but may not require a full image rebuild. The CI pipeline will detect the change and only rebuild the Slingshot layer and all downstream layers, leaving previous layers unchanged, but it will rebuild all image types that use slingshot without administrator intervention.

Some changes will require full rebuilds. Kernel updates and security fixes are generally done in the base OS layer, but because of the CI pipeline this can be done automatically and for all systems simultaneously.

B. Portability and Scalability

It has been the practice, at least at LANL, to build images for each cluster. While it is true that different systems have different configurations, at some level these systems share many of the same properties. Because the CI image build pipeline is containerized, it should be possible to build layers outside a cluster's management plane and share these layers between systems. Another use case is to build production images on a testbed, validate changes, and then push images

Fig. 5. =
CFS full layer dependency. A stripped down example of the layer dependencies of one of our local test systems. The node names are mapped as ⟨config repo⟩-⟨image type⟩.



to a production system, without having to rebuild images for production systems.

The CI image build pipeline is also scalable. The only limitation is on how many gitlab-runners are available for configuring layers, and because the image layer build is containerized, this can be done in a variety of ways. A single Podman container can only build a single layer at a time, but a Kubernetes deployment of many instances of the image builder can build as many layers simultaneously as there are image builder pods.

C. Consistency, Validation, and Security

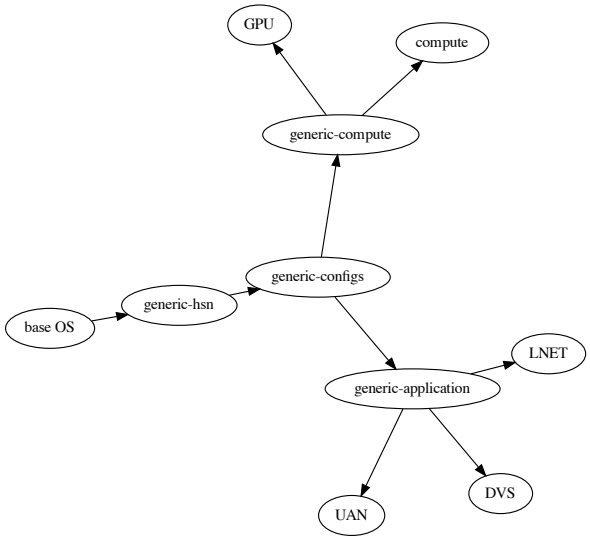
Because layers are not rebuilt unless changes are made or a parent is updated, there is a forced consistency to the CI image build pipeline. Layers known to be in a good state will not change or get updated erroneously and any attempted changes will always be validated because the pipeline will catch these changes. Validating changes through the pipeline will prevent user error and prevent untested changes, from the image build perspective, from reaching production systems.

This of course requires some administration on the gitlab servers to prevent push to production branches and gitlab CI configs to trigger the pipeline on the desired action (i.e. merge requests). But if configured correctly no unanticipated changes should go unvalidated or skip the image build pipeline.

D. Ansible Configuration Structure

The automated pipeline generation should be able to handle a number of Ansible configuration structures. For example, in a CFS like structure, image types are very distinct, having separate repositories for image types and also branching out very early in the configuration process. This can be seen in Fig. 5, where the base OS is shared among all image types, the first layer after the base (slingshot in LANL's case) branches by image type and the result is that any future layers must be distinct by image type, even if they share many of the same configurations. This can be further exacerbated by having multiple compute systems, where many of the configurations are the same, but system distinctness appears in earlier layers forcing uniqueness in layers where it may not be needed.

Fig. 6. =
Theoretical layer dependency. The idea is to push more generic, shareable configurations to earlier layers, which would reduce build times and configuration drift.



An alternative is to push as many similar configurations as possible to the earlier layers and any specific image type configurations to later layers. This can have the benefit of speeding up image builds in cases where the desired changes are further down the dependency chain and/or specific to an image type. If we add in multiple systems then it should be possible to have a generic layer for a node type (i.e. Compute) and only add system specific configurations on top of the generic layer. This will make it easier to share configurations between not only different image types but also between multiple systems. A theoretical example can be seen in Fig. 6.

In either case, or in future cases, the automated pipeline should be able to handle a variety of configurations and structures, and the automated nature of the pipeline can alleviate many of the annoying aspects of manually building and configuring cluster images.

V. Summary

An automated pipeline has many advantages, forcing consistency and preventing unanticipated changes to production systems. The underlying pieces that allow this kind of automation, leveraging container technology and tools to build images in layers, and having the ability to import and build off previous layers reduces the amount of time it takes to push changes into production while reducing risk. The automated detection of changes, and where those changes occur, allows images to build from where said changes have occurred, instead of completely rebuilding the entire image from scratch. It also has the advantage of only building specific image types if changes only affect the respective image type.

Another advantage of the pipeline process is the ability to "plugin" to external pipelines. These external pipelines can do a myriad of tasks; like automated testing of images, booting images on a testbed system to validate changes made, or regression testing. This allows a plug-n-play set of features that can be expanded on, can be done locally and however a site chooses.

Finally, the automated pipeline can be used to generify the image build process away from a cluster specific task. Allowing multiple systems to share layers and then further customize to a specific system later in the build chain. This will reduce system configuration drift from system to system and force consistency between multiple systems, which can affect the time it takes to push out security patches, product updates, and general system changes.

While this work is in the initial stages, if there is sufficient community interest it is the goal of this team to release an open source version of this product.

REFERENCES

[1] Merkel, D., "Docker: lightweight linux containers for consistent development and deployment'" Linux Journal, vol. 2, 2014.

[2] Heon, Matt et. al, "Podman - : A tool for managing OCI containers and pods (v1.0 and beyond. Currently at v3.0.1)", Zenodo. https://doi.org/10.5281/zenodo.4735634.

[3] D.J. Walsh, N. Dahyabhai, et. al, "Buildah - a tool that facilitates building Open Container Initiative (OCI) container images", https://github.com/containers/buildah/blob/main/README.md

[4] T. Tomecek, "buildah connection - Interact with an existing buildah container", https://docs.ansible.com/ansible/latest/collections/containers/podman/buildah_connection.html.

[5] "GitLab Runner", https://docs.gitlab.com/runner/.

[6] "Predefined variables reference", https://docs.gitlab.com/ee/ci/variables/predefined_variables.html

[7] A. A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

[8] R. Priedhorsky, R.S. Canon, T. Randles, A.J. Younge, "Minimizing privilege for building HPC containers", SC '21: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, November 2021 Article No.: 32