LA-UR-23-24762

Title: CI/CD Image Build Pipelinin

Author(s): Cotton, Travis Bradley

Intended for: Cray User Group, 2023-05-07/2023-05-12 (Helsinki, Finland)

Issued: 2023-05-04 (Draft)

Los Alamos
NATIONAL LABORATORY

Los Alamos
NATIONAL LABORATORY

CI/CD Image Build Pipelining

Travis Cotton

Outline

Layered Image Building

- Layer Definition
- Tools in use

CI Pipeline

- Define CI Pipelines
- Simple Examples
- Automated Generation

Observations and Benefits

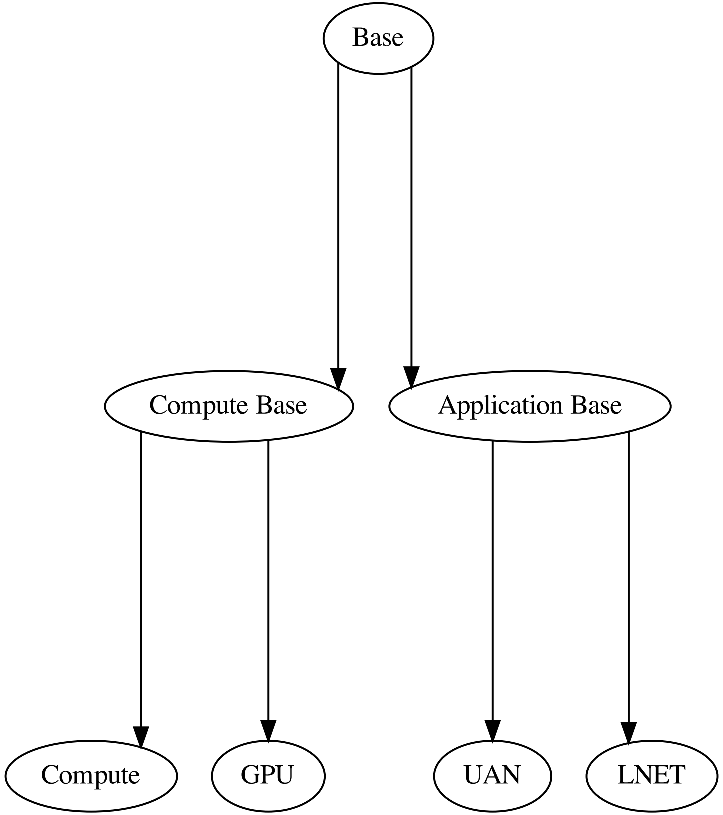
- Turnaround Time
- Portability and Scaling
- Consistency and Validation

Future Work

- Automated Testing

Part 1: What is a Layer?

- A distinct, self-contained piece or part of a larger image
- Has parent layer
  - Previously configured layer
  - Blank or scratch layer
- A defined configuration
  - Ansible playbook
  - List of packages



Tools in Use

- Containers!
  - A layer can be started as a container
  - Import from a parent, a.k.a a previously built container image or scratch
  - Treat configured layers as a container image
    - Push to a registry
    - Export to a tarball, squashfs, etc

- Buildah Containers!
  - On the fly containers
  - No Dockerfiles, or Daemons required
  - Easy to debug things

On the fly base image

- Start an empty container

```
image-testing:~ # CNAME=$(buildah from scratch)
```

On the fly base image

- Start an empty container
- Mount the container

```
image-testing:~ # CNAME=$(buildah
from scratch)

image-testing:~ # MDIR=$(buildah
mount $CNAME)
```

On the fly base image

- Start an empty container
- Mount the container
- Install packages

```
image-testing:~ # CNAME=$(buildah
from scratch)

image-testing:~ # MDIR=$(buildah
mount $CNAME)

image-testing:~ # zypper --
installroot $MDIR in bash
```

On the fly base image

- Start an empty container
- Mount the container
- Install packages
- Commit image

```
image-testing:~ # CNAME=$(buildah
from scratch)

image-testing:~ # MDIR=$(buildah
mount $CNAME)

image-testing:~ # zypper --
installroot $MDIR in bash

image-testing:~ # buildah commit --rm
$CNAME sles15-base
```

On the fly base image

- Start an empty container
- Mount the container
- Install packages
- Commit image
- View it!

```
image-testing:~ # CNAME=$(buildah
from scratch)

image-testing:~ # MDIR=$(buildah
mount $CNAME)

image-testing:~ # zypper --
installroot $MDIR in bash

image-testing:~ # buildah commit --rm
$CNAME sles15-base

image-testing:~ # buildah images |
awk ‘{print $1}’
REPOSITORY
localhost/sles15-base
```

On the fly base image

- Let’s add more stuff to our base image

On the fly base image

- Let’s add more stuff to our base image
- Start a container from our base image
- Give it a name

```
image-testing:~ # buildah from --name 
base-update sles15-base
```

On the fly base image

- Let’s add more stuff to our base image
- Start a container from our base image
- Give it a name
- Mount it again

```
image-testing:~ # buildah from --name
base-update sles15-base

image-testing:~ # MDIR=$(buildah
mount base-update)
```

On the fly base image

- Let’s add more stuff to our base image
- Start a container from our base image
- Give it a name
- Mount it again
- Add more packages

```
image-testing:~ # buildah from --name
base-update sles15-base

image-testing:~ # MDIR=$(buildah
mount base-update)

image-testing:~ # zypper --
installroot=$MDIR in coreutils
python3 zypper
```

On the fly base image

- Let’s add more stuff to our base image
- Start a container from our base image
- Give it a name
- Mount it again
- Add more packages
- Commit it

```
image-testing:~ # buildah from --name
base-update sles15-base

image-testing:~ # MDIR=$(buildah
mount base-update)

image-testing:~ # zypper --
installroot=$MDIR in coreutils
python3 zipper

image-testing:~ # buildah commit --rm
base-update sles15-base-v2
```

On the fly layer configuration

- We’ve only installed packages so far

On the fly layer configuration

- We’ve only installed packages so far
- Let’s run ansible against our container!

On the fly layer configuration

- First make an inventory
  - Make a ”Compute” group
  - Add our layer “compute-cont” to the group
  - Set the connection type to be “buildah”

```
#Inventory hosts file
[Compute]
compute-cont ansible_connection=buildah
```

On the fly layer configuration

- Inventory is done
- Let’s make a playbook with some roles to run
  - Run against the “Compute” group
  - Use four roles
  - Pretty standard stuff
- Call the playbook compute.yaml

```
---
- hosts:
    - Compute
  roles:
    - repos
    - pkgs
    - chrony
    - nfs
```

On the fly layer configuration

- Start a new container

```
image-testing:~ # buildah from --name 
compute-cont sles15-base-v2
```

On the fly layer configuration

- Start a new container
- Run ansible against this container

```
image-testing:~ # buildah from --name
compute-cont sles15-base-v2

image-testing:~/test-ansible #
ansible-playbook -i inventory/
compute.yaml
```

On the fly layer configuration

- Start a new container
- Run ansible against this container
- Hopefully it runs correctly…

```
image-testing:~ # buildah from --name
compute-cont sles15-base-v2

image-testing:~/test-ansible #
ansible-playbook -i inventory/
compute.yaml
…
ok=7    changed=4    unreachable=0
failed=0
```

On the fly layer configuration

- But if not, fix the error
  - Run against existing container
  - Or remove and start over
- Annoying or Mysterious errors
  - Jump into container and poke around!

```
image-testing:~ # buildah from --name
compute-cont sles15-base-v2

image-testing:~/test-ansible #
ansible-playbook -i inventory/
compute.yaml
…
ok=4    changed=2    unreachable=0
failed=1

image-testing:~/test-ansible #
buildah run --tty compute-cont bash
```

On the fly layer configuration

- If happy with your image, commit it
- And see our images so far

```
image-testing:~ # buildah commit --rm
compute-cont compute-v1

image-testing:~ # buildah images |
awk '{print $1}'
REPOSITORY
localhost/compute-v1
localhost/sles15-base-v2
localhost/sles15-base
```

On the fly layer configuration

- If happy with your image, commit it
- And see our images so far
- You can build more images if you want
- Any of the current images can be used as a parent

```
image-testing:~ # buildah commit --rm
compute-cont compute-v1

image-testing:~ # buildah images |
awk '{print $1}'
REPOSITORY
localhost/compute-v1
localhost/sles15-base-v2
localhost/sles15-base
```

Quick Recap

- Easily build base images
  - No complicated configurations needed
  - Easy to update
- Leverage Base Images
  - Import base and add configurations
  - Run ansible against layer container
  - Convenient to debug problems
- Still a very manual process
- Clunky ansible inventory

Let’s do a little programming

- Need a way to encapsulate the previous steps
- Things we need to know
  - Parent
  - Layer name/type
  - Ansible group(s)
  - Playbook and inventory to use
- Python is our language of choice because…
  - Pretty easy to use
  - Popular
  - I wanted to learn it

Let’s do a little programming

- Not going to paste a bunch of python code
- Highlight a few neat things
  - Ansible has python libraries available

Let’s do a little programming

- Not going to past a bunch of python code
- Highlight a few neat things
  - Ansible has python libraries available
  - Load up inventory

```
inventory =
InventoryManager(loader=loader,
sources=inv)
```

Let’s do a little programming

- Not going to past a bunch of python code
- Highlight a few neat things
  - Ansible has python libraries available
  - Load up inventory
  - Add group(s)

```
inventory =
InventoryManager(loader=loader,
sources=inv)

inventory.add_group(“Compute”)
```

Let’s do a little programming

- Not going to past a bunch of python code
- Highlight a few neat things
  - Ansible has python libraries available
  - Load up inventory
  - Add group(s)
  - Add container as host

```
inventory =
InventoryManager(loader=loader,
sources=inv)

inventory.add_group(“Compute”)

inventory.add_host(host=compute-cont,
group=“Compute”)
```

Let’s do a little programming

- Not going to past a bunch of python code
- Highlight a few neat things
  - Ansible has python libraries available
  - Load up inventory
  - Add group(s)
  - Add container as host
  - Run playbooks

```
inventory =
InventoryManager(loader=loader,
sources=inv)

inventory.add_group(“Compute”)

inventory.add_host(host=compute-cont,
group=“Compute”)

pbex =
PlaybookExecutor(playbooks=pbs,
inventory=inventory, …)
```

Let’s do a little programming

- Buildah can be easily wrapped
- Or you can steal wrappers from ansible-bender…

```
from ansible_bender.utils import run_cmd
```

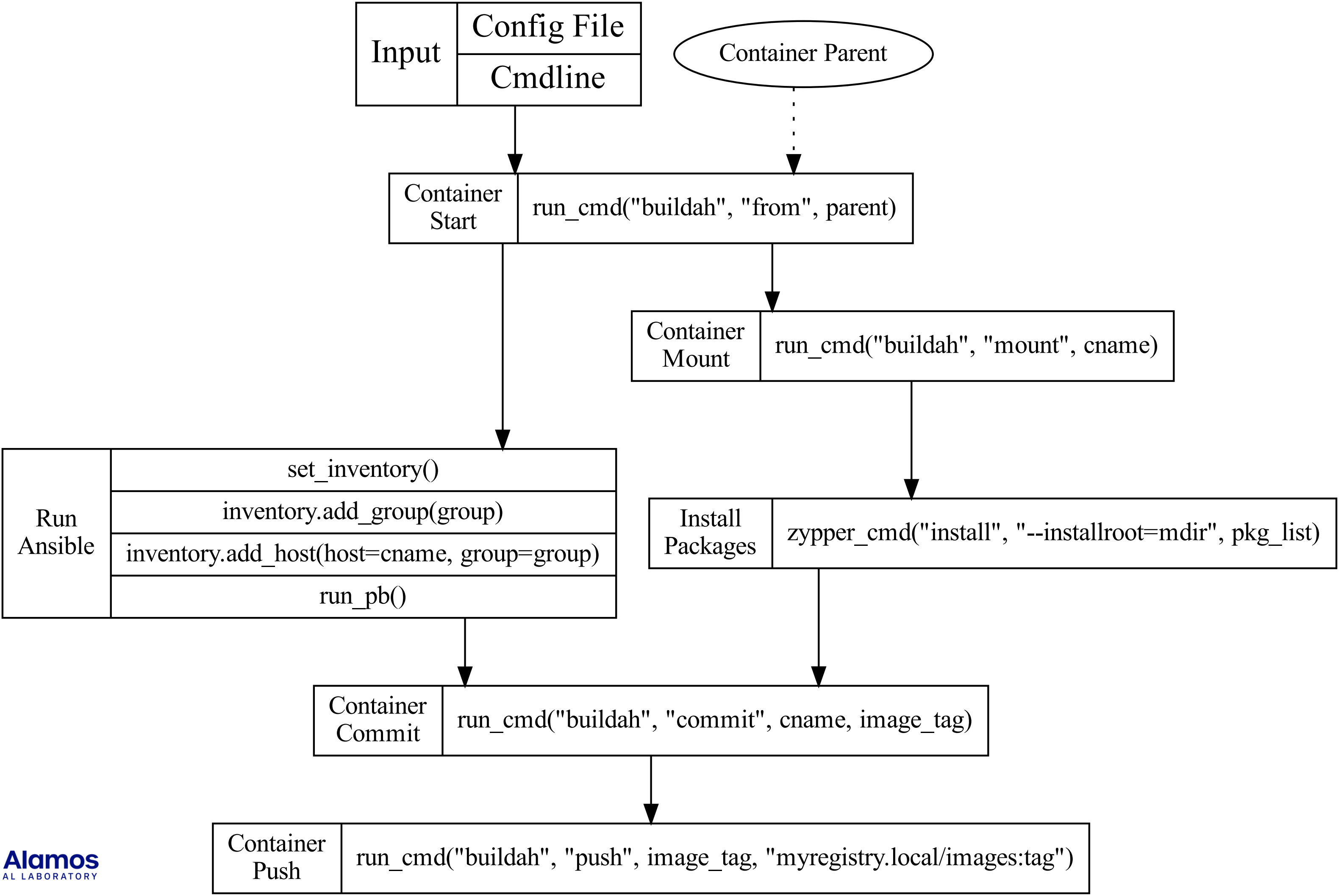
Let’s do a little programming

- Buildah can be easily wrapped
- Or you can steal wrappers from ansible-bender…
- And use the “run_cmd” function

```
from ansible_bender.utils import
run_cmd

cmd = [“buildah”, “from”, “scratch”]

cname = run_cmd(cmd,
return_output=True)
```



Input

Config File

Cmdline

Container Parent

Container Start

run_cmd("buildah", "from", parent)

Container Mount

run_cmd("buildah", "mount", cname)

Run Ansible

set_inventory()

inventory.add_group(group)

inventory.add_host(host=cname, group=group)

run_pb()

Install Packages

zypper_cmd("install", "--installroot=mdir", pkg_list)

Container Commit

run_cmd("buildah", "commit", cname, image_tag)

Container Push

run_cmd("buildah", "push", image_tag, "myregistry.local/images:tag")

Layered Image Build Recap

- Containerized layer builds
  - Import from Parent layer
  - Multiple Layers can use the same parent
- Multiple ways to build a layer
  - Package managers
  - Ansible playbooks
  - Custom scripts
- Programmatic Builds
  - Using Python
  - Easily add groups and inventory info
  - Can easily chain multiple layers together

Part 2: CI Pipelines

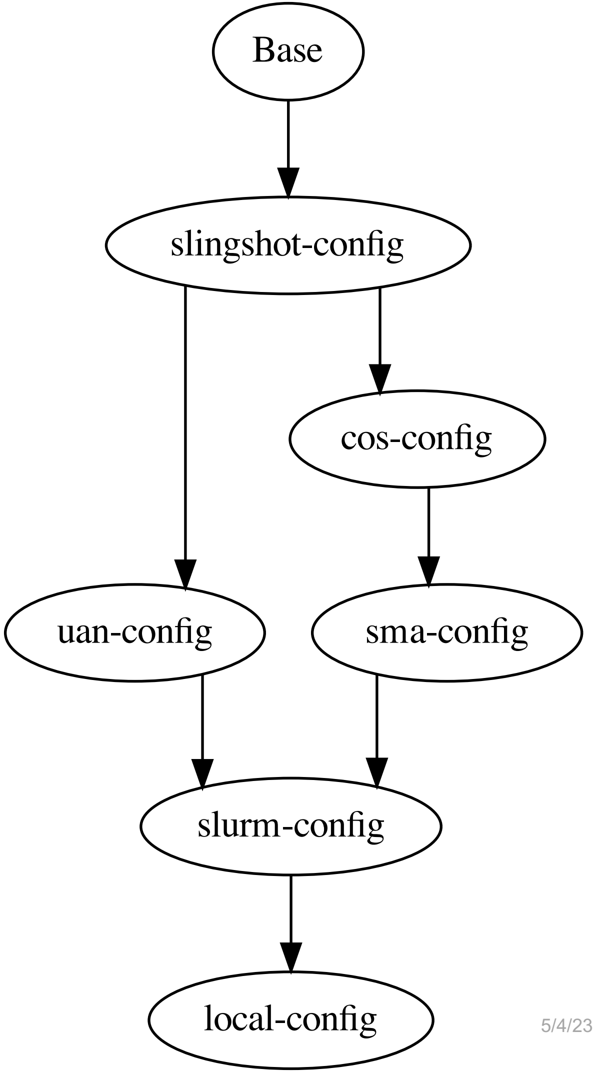
- Let Layer Builds be simple
  - Import from parent
  - Configure current layer
  - Commit layer as new image

Part 2: CI Pipelines

- Let Layer Builds be simple
  - Import from parent
  - Configure current layer
  - Commit layer as new image
- Let the pipeline handle the coordination
  - Who’s parent is who’s
  - What configuration to use
  - What to do with a configured layer

Part 2: What is a CI Pipeline?

- CI = Continuous Integration
  - An automatic way to integrate new configs into production branches
- Pipelines are the logic that dictates if changes can be integrated
- In this context a CI pipeline
  - Ensures configs results in a successful image build
  - Defines the layer dependencies that results in a complete image
- We are using gitlab – for now
  - https://about.gitlab.com/topics/ci-cd/



Los Alamos NATIONAL LABORATORY

CI pipeline – simple example

- Let's start with a simple example
  - Single repo
  - These things are pretty verbose, bear with me

```
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 >
```

CI pipeline – simple example

- Let’s start with a simple example
  - Single repo
  - These things are pretty verbose, bear with me
- Will run stages in order

```
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 >
```

CI pipeline – simple example

- Let’s start with a simple example
  - Single repo
  - These things are pretty verbose, bear with me
- Will run stages in order
- Script tells it what to do

```
stages: [full, layer1, layer2]

inventory-job:
  stage: full
  script:
    - imgbuild -name sles15-base -parent scratch
    - imgbuild -name layer1 -parent sles15-base
    - imgbuild -name layer2 -parent layer1
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - inventory/**

layer1-job:
  stage: layer1
  script:
    - imgbuild -name layer1 -parent sles15-base
    - imgbuild -name layer2 -parent layer1
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - < list of layer 1 roles >

layer2-job:
  stage: layer2
  script:
    - imgbuild -name layer2 -parent layer1
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - < list of layer 2 roles >
```

CI pipeline – simple example

- Let’s start with a simple example
  - Single repo
  - These things are pretty verbose, bear with me
- Will run stages in order
- Script tells it what to do
- Rules decide when to run

```
stages: [full, layer1, layer2]

inventory-job:
  stage: full
  script:
    - imgbuild -name sles15-base -parent scratch
    - imgbuild -name layer1 -parent sles15-base
    - imgbuild -name layer2 -parent layer1
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - inventory/**

layer1-job:
  stage: layer1
  script:
    - imgbuild -name layer1 -parent sles15-base
    - imgbuild -name layer2 -parent layer1
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - < list of layer 1 roles >

layer2-job:
  stage: layer2
  script:
    - imgbuild -name layer2 -parent layer1
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - < list of layer 2 roles >
```

CI pipeline – simple example

- Let’s start with a simple example
  - Single repo
  - These things are pretty verbose, bear with me
- Will run stages in order
- Script tells it what to do
- Rules decide when to run
- Changes are the files we trigger on

```
stages: [full, layer1, layer2]

inventory-job:
  stage: full
  script:
    - imgbuild -name sles15-base -parent scratch
    - imgbuild -name layer1 -parent sles15-base
    - imgbuild -name layer2 -parent layer1
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - inventory/**

layer1-job:
  stage: layer1
  script:
    - imgbuild -name layer1 -parent sles15-base
    - imgbuild -name layer2 -parent layer1
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - < list of layer 1 roles >

layer2-job:
  stage: layer2
  script:
    - imgbuild -name layer2 -parent layer1
  rules:
    - if: $CI_MERGE_REQUEST_ID
  changes:
    - < list of layer 2 roles >
```

CI pipeline – simple example

- Our simple repo might look something like this
- A list of packages for our base layer
- A playbook for each layer
  - Each with a list of roles

```
.
├── base_packages
├── inventory
│   └── hosts
├── layer1.yaml
├── layer2.yaml
└── roles
    ├── chrony
    ├── cve_fixes
    ├── nfs
    ├── nhc
    ├── pkgs
    ├── repos
    └── ssh
```

CI pipeline – simple example

- The layer playbooks
- Our changes for each stanza have a source
  - The base packages file and inventory for the base layer
  - Each layer playbook's list of roles

```
---
# Layer1
- hosts:
    - Compute
  roles:
    - repos
    - pkgs
    - chrony
    - nfs

---
# Layer 2
- hosts:
    - Compute
  roles:
    - ssh
    - cve_fixes
    - nhc
```

CI pipeline – simple example

- Layer1 for example will have this
- If anything in these folders changes, the pipeline will start from layer1
- Layer2 will also be built
- But we didn’t need to rebuild the base

```
script:
  - imgbuild -name layer1 -parent sles15-base
  - imgbuild -name layer2 -parent layer1
changes:
  - roles/repos/**
  - roles/pkgs/**
  - roles/chrony/**
  - roles/nfs/**
```

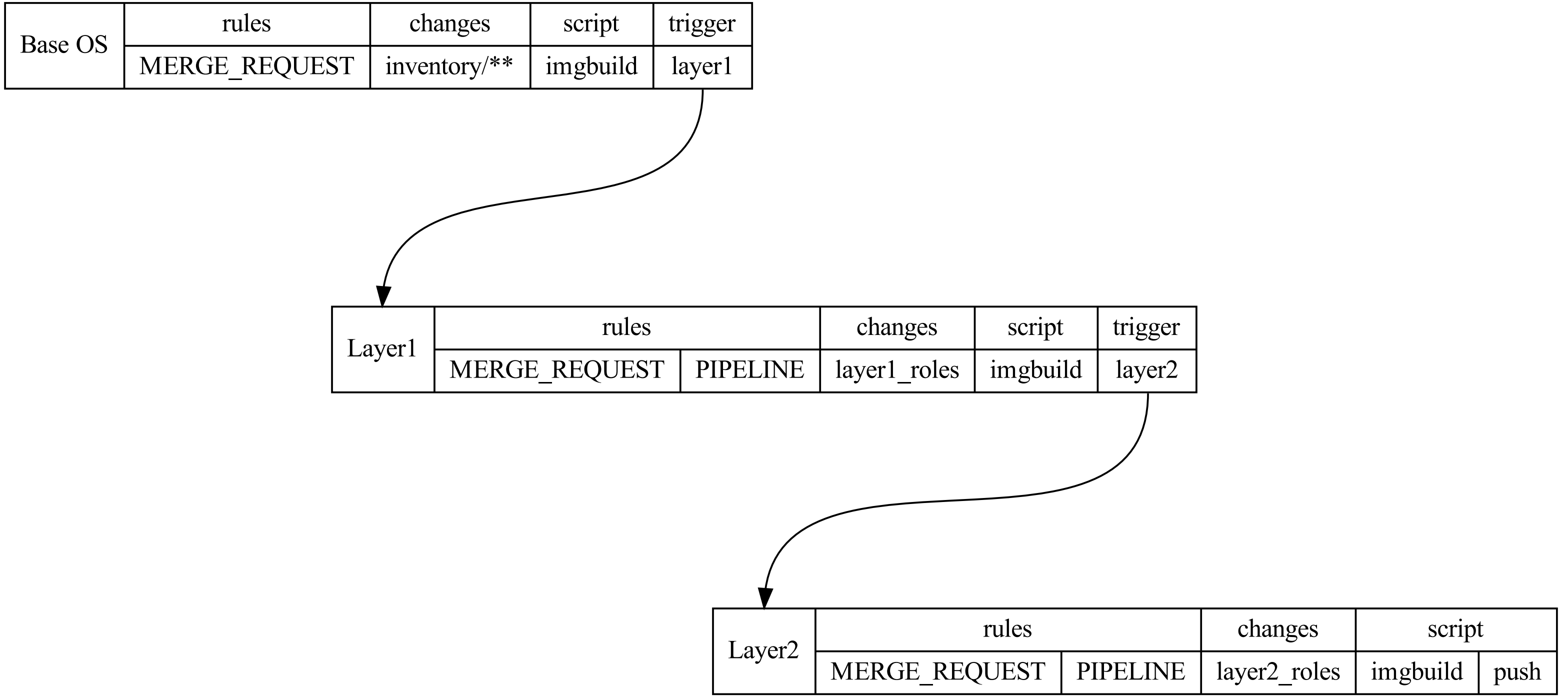
CI pipeline – multi-repo example

- What if I don’t want to use a single repo?
- Gitlab has the “trigger” keyword
  - Start a pipeline in another repo
- Instead of lengthy configs, we’ll attempt this with pictures

CI pipeline – multi-repo example

- Same setup as before
  - Base
  - Layer1
  - Layer2
- Each in their own special repo
- Pipelines will trigger when
  - A merge request is made
  - A parent pipeline triggers

CI pipeline – multi-repo example - Base

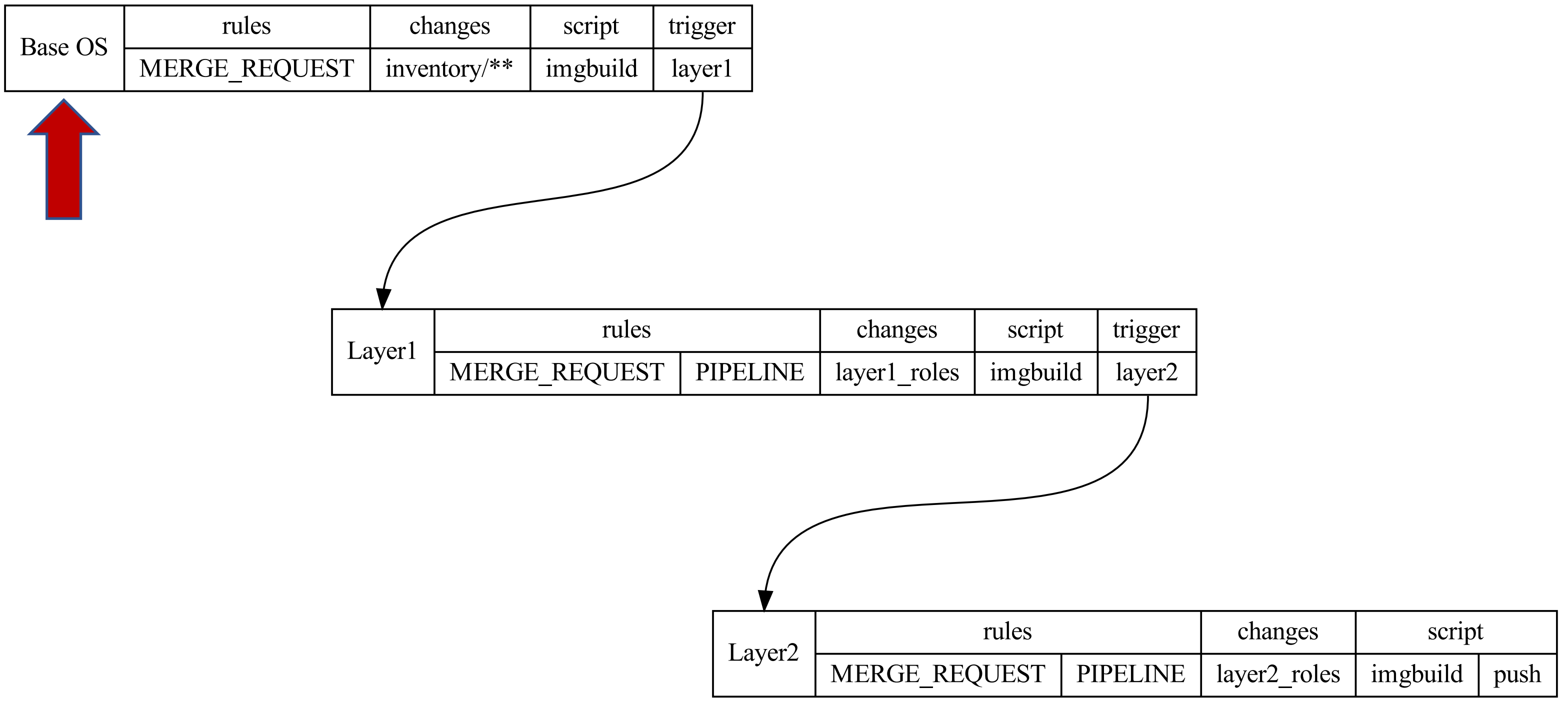


Base OS	rules	changes	script	trigger
	MERGE_REQUEST	inventory/**	imgbuild	layer1

Layer1	rules		changes	script	trigger
	MERGE_REQUEST	PIPELINE	layer1_roles	imgbuild	layer2

Layer2	rules		changes	script	
	MERGE_REQUEST	PIPELINE	layer2_roles	imgbuild	push

CI pipeline – multi-repo example - Base

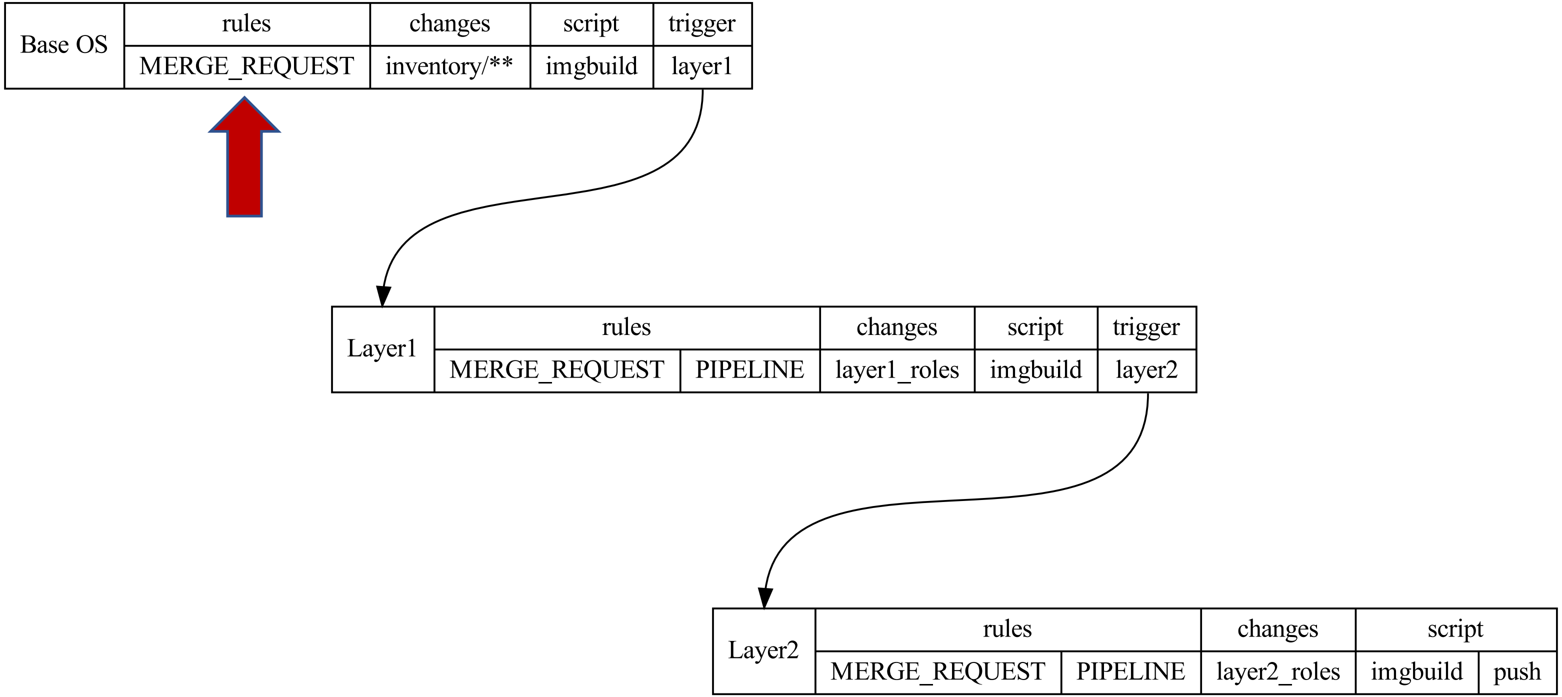


Base OS	rules	changes	script	trigger
	MERGE_REQUEST	inventory/**	imgbuild	layer1

Layer1	rules		changes	script	trigger
	MERGE_REQUEST	PIPELINE	layer1_roles	imgbuild	layer2

Layer2	rules		changes	script	
	MERGE_REQUEST	PIPELINE	layer2_roles	imgbuild	push

CI pipeline – multi-repo example - Base

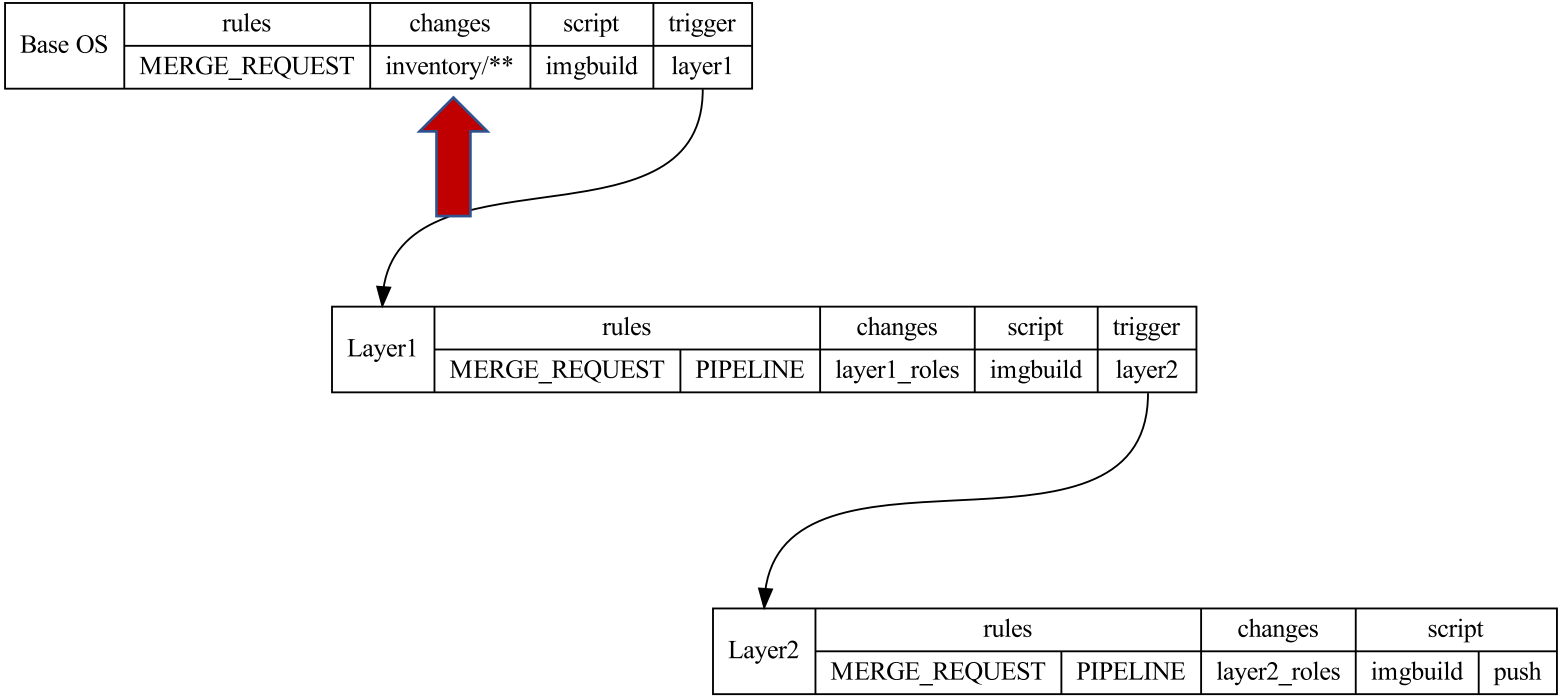


Base OS	rules	changes	script	trigger
	MERGE_REQUEST	inventory/**	imgbuild	layer1

Layer1	rules		changes	script	trigger
	MERGE_REQUEST	PIPELINE	layer1_roles	imgbuild	layer2

Layer2	rules		changes	script	
	MERGE_REQUEST	PIPELINE	layer2_roles	imgbuild	push

CI pipeline – multi-repo example - Base

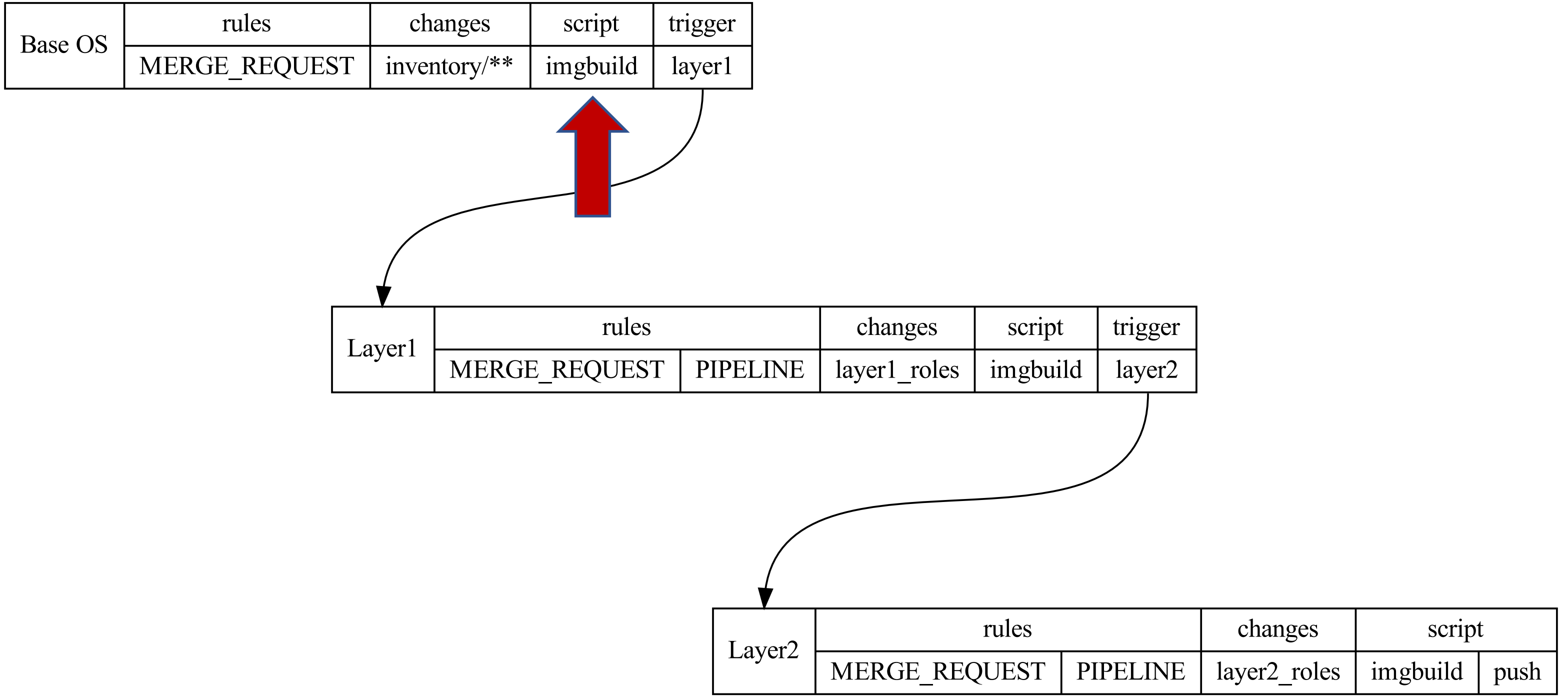


Base OS	rules	changes	script	trigger
	MERGE_REQUEST	inventory/**	imgbuild	layer1

Layer1	rules		changes	script	trigger
	MERGE_REQUEST	PIPELINE	layer1_roles	imgbuild	layer2

Layer2	rules		changes	script	
	MERGE_REQUEST	PIPELINE	layer2_roles	imgbuild	push

CI pipeline – multi-repo example - Base

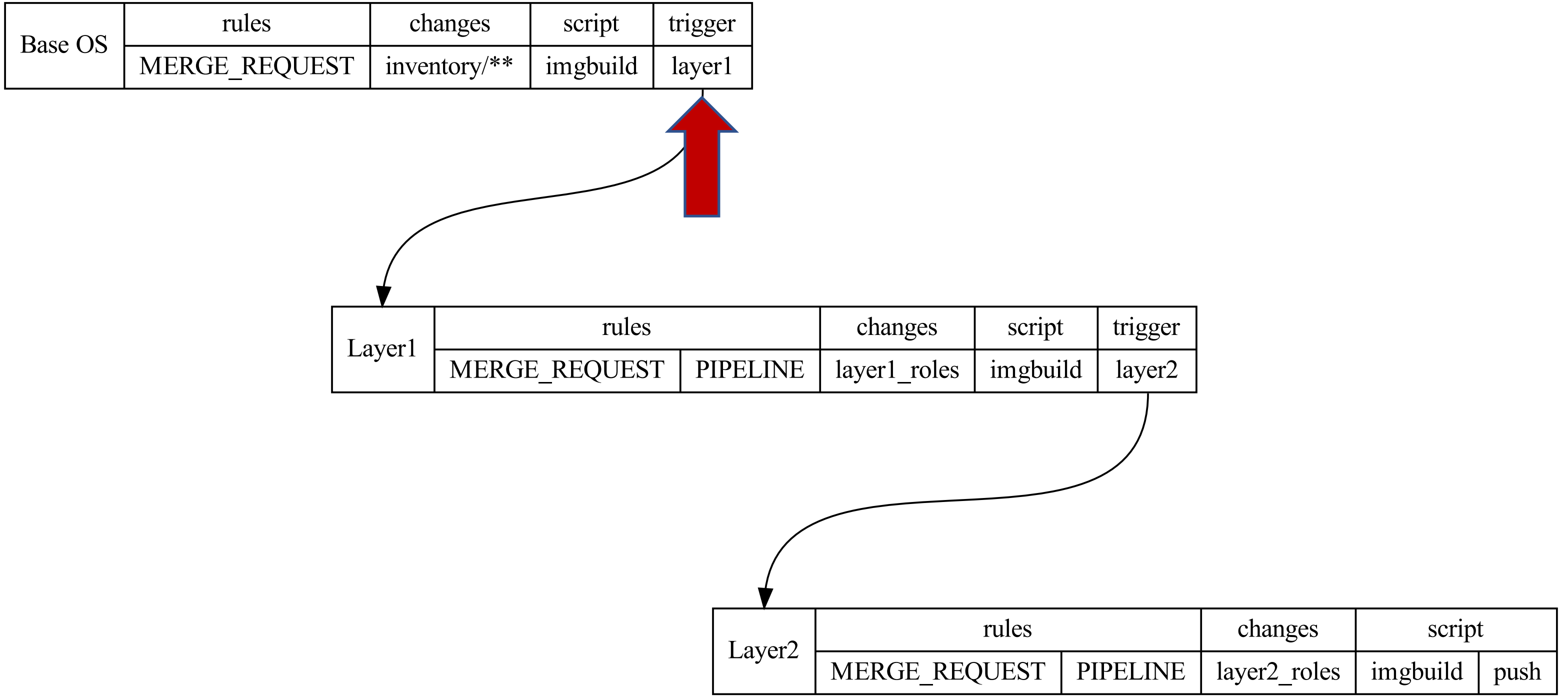


Base OS	rules	changes	script	trigger
	MERGE_REQUEST	inventory/**	imgbuild	layer1

Layer1	rules		changes	script	trigger
	MERGE_REQUEST	PIPELINE	layer1_roles	imgbuild	layer2

Layer2	rules		changes	script	
	MERGE_REQUEST	PIPELINE	layer2_roles	imgbuild	push

CI pipeline – multi-repo example - Base

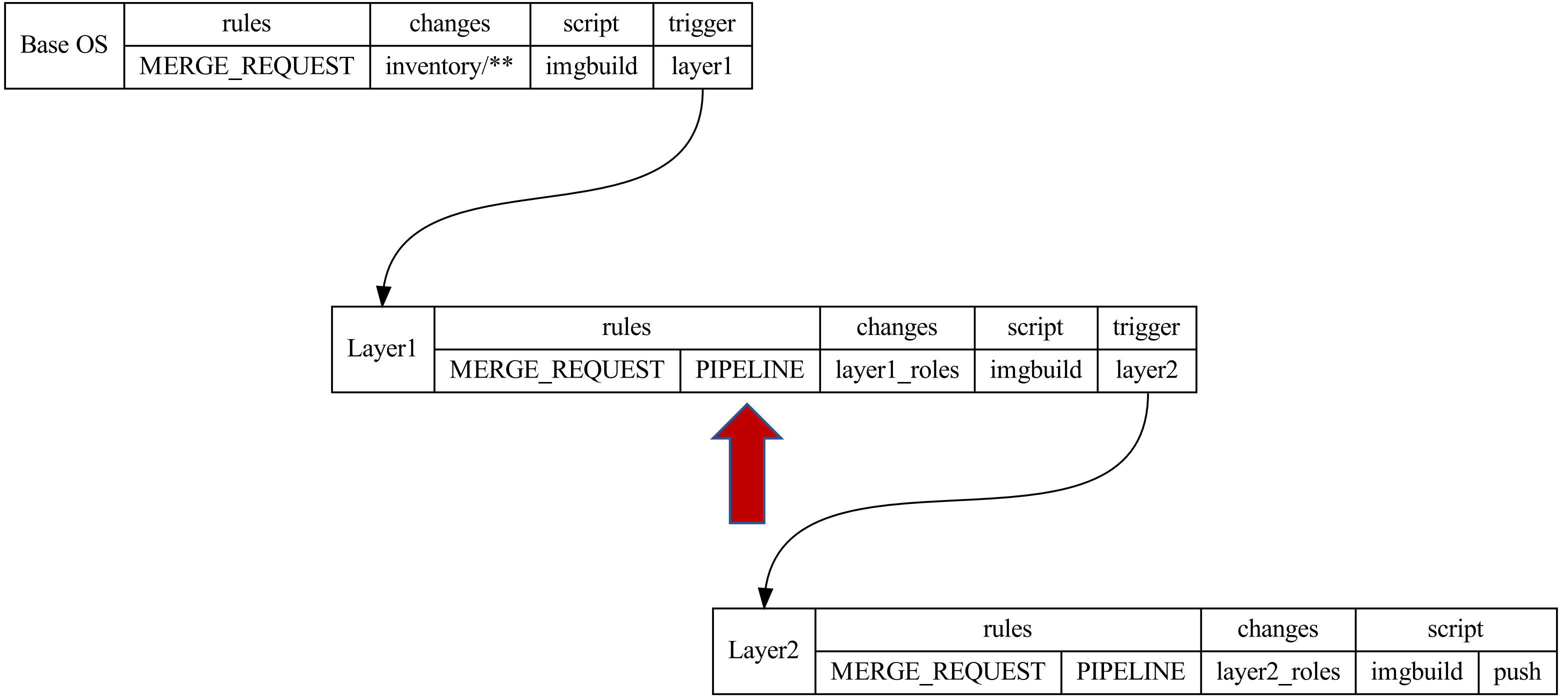


Base OS	rules	changes	script	trigger
	MERGE_REQUEST	inventory/**	imgbuild	layer1

Layer1	rules		changes	script	trigger
	MERGE_REQUEST	PIPELINE	layer1_roles	imgbuild	layer2

Layer2	rules		changes	script	
	MERGE_REQUEST	PIPELINE	layer2_roles	imgbuild	push

CI pipeline – multi-repo example - Base



Base OS	rules	changes	script	trigger
	MERGE_REQUEST	inventory/**	imgbuild	layer1

Layer1	rules		changes	script	trigger
	MERGE_REQUEST	PIPELINE	layer1_roles	imgbuild	layer2

Layer2	rules		changes	script	
	MERGE_REQUEST	PIPELINE	layer2_roles	imgbuild	push

CI pipeline – examples recap

- The pipeline will auto-trigger on
  - Merge request
  - And if defined changes are detected
- Any changes will be validated
  - Breaking changes will cause build failure
  - Prevent merge
  - Notify assignees
  - No sneaking in changes

CI pipeline – examples recap

- This can grow to be pretty complex
    - Rearranging can be painful
        - Parent-child relationship can change
    - Keeping track of changes to the pipeline
        - Map the right configs to layers
        - Adding new layers
- Let’s automate it!

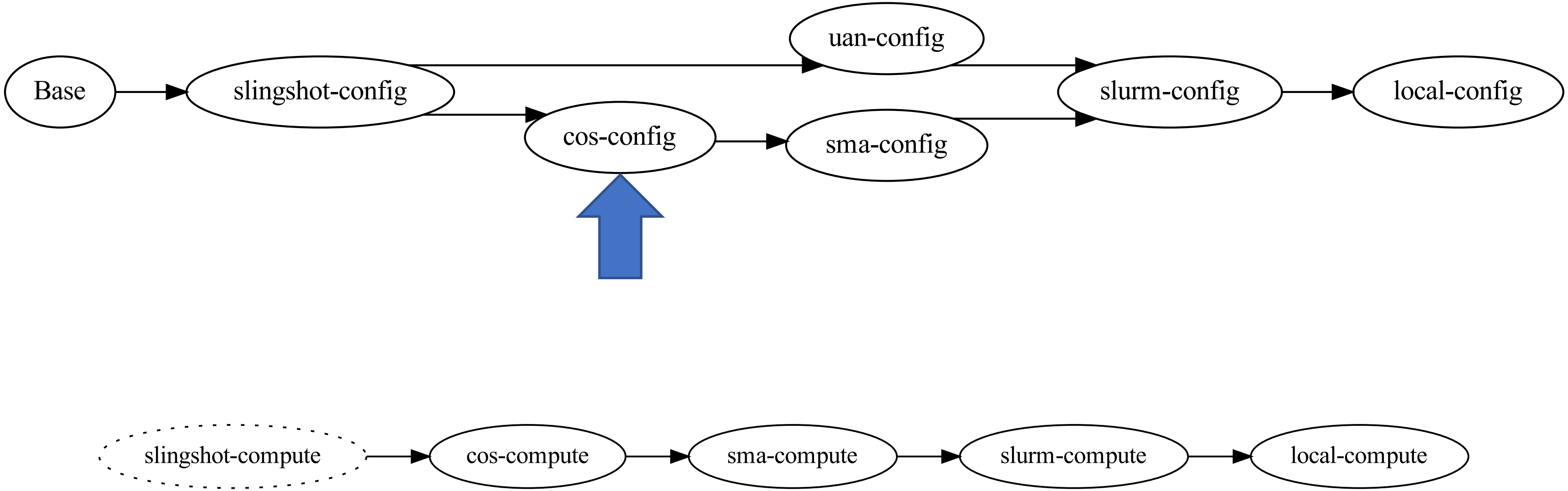
CI pipeline – Automated

- Dynamically write the pipeline configs
  - Pipeline configs are written in yaml
  - Easy to build and write in python
- Treat parent child relationships as a DAG
  - Start the pipeline in the correct place
  - Layer configs can apply to multiple image types
  - Use correct configurations for layers

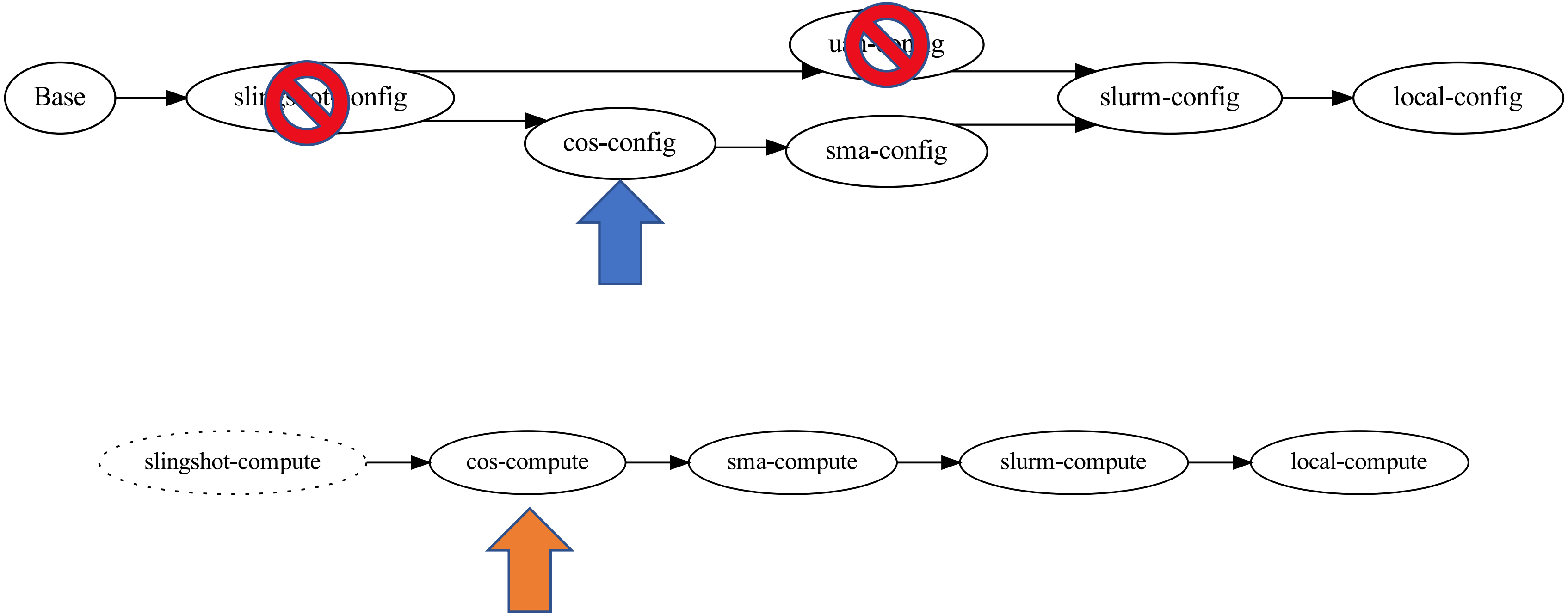
CI pipeline – Automated

- Use a staging repo
  - Triggered by layer repos
  - Responsible for writing pipeline
  - Holds layer dependency DAG

CI pipeline – Automated



CI pipeline – Automated



CI pipeline – LANL

- Separate Inventory for each system
- Generic repos for each layer
  - Can apply to all systems
  - Can apply to multiple image types
- Three trigger types
  - Inventory
  - Roles
  - Base

Base type trigger

- Still under development
  - Currently only one base type layer
  - Plan is to move towards group base layers
    - Application
    - COS

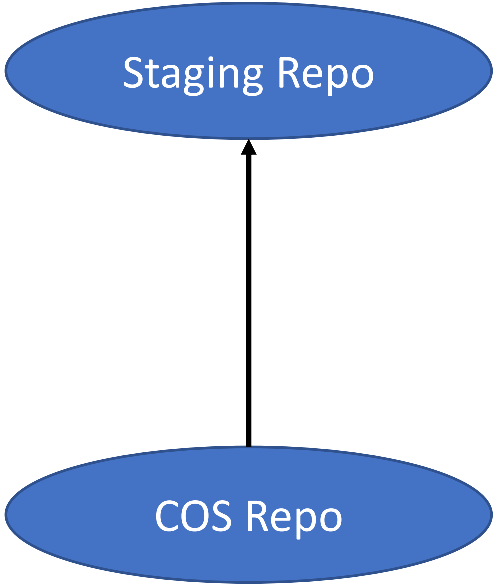
Role type trigger

- Looks for source in DAG
- Match changes to roles/tasks used in defined playbooks
- Write pipeline for affected layer types

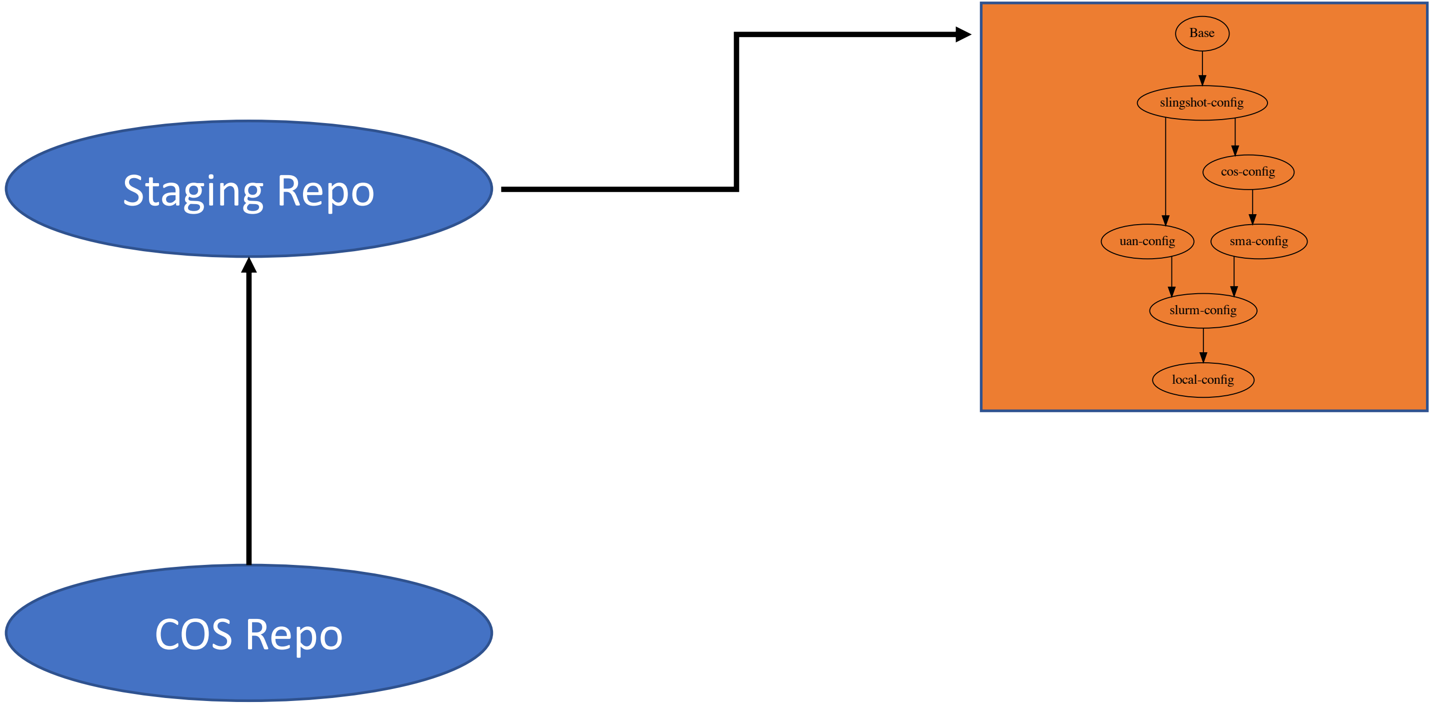
A merge request is made

COS Repo

Staging repo is triggered



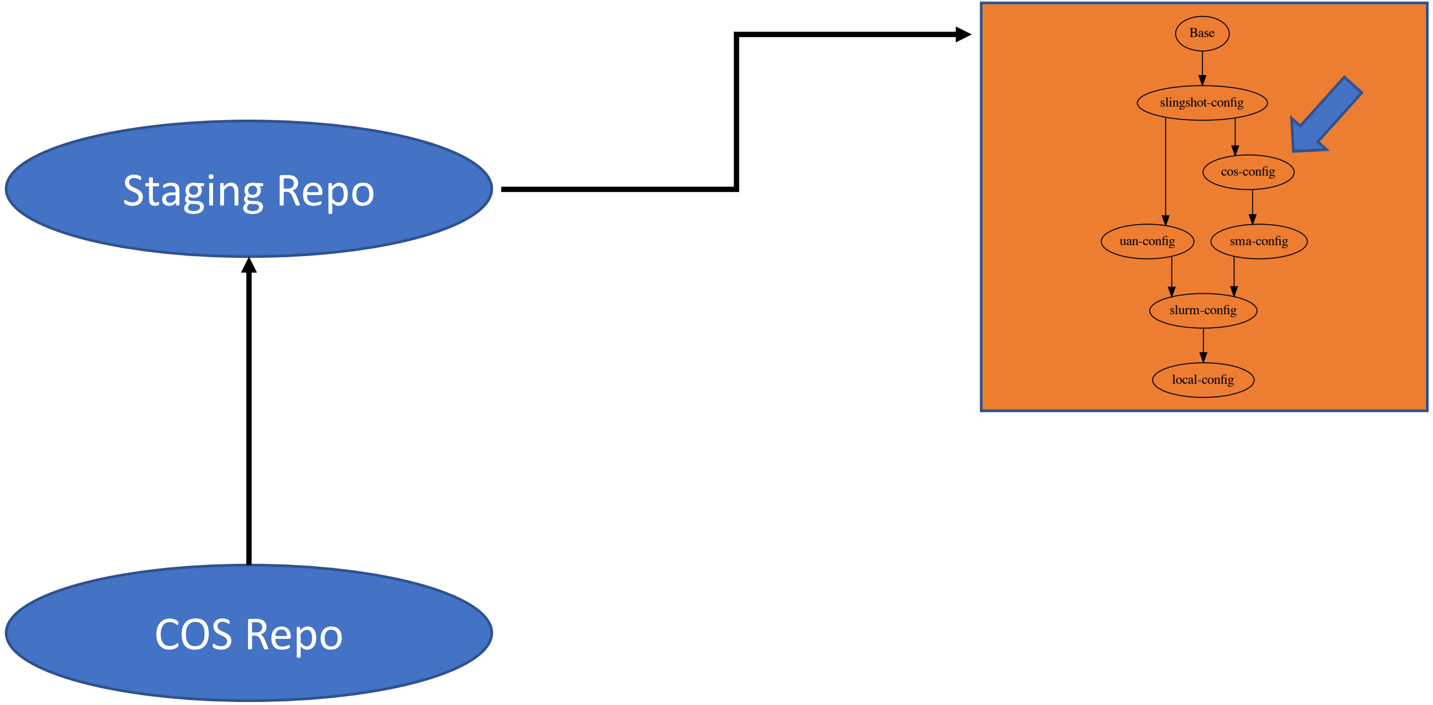
Search DAG



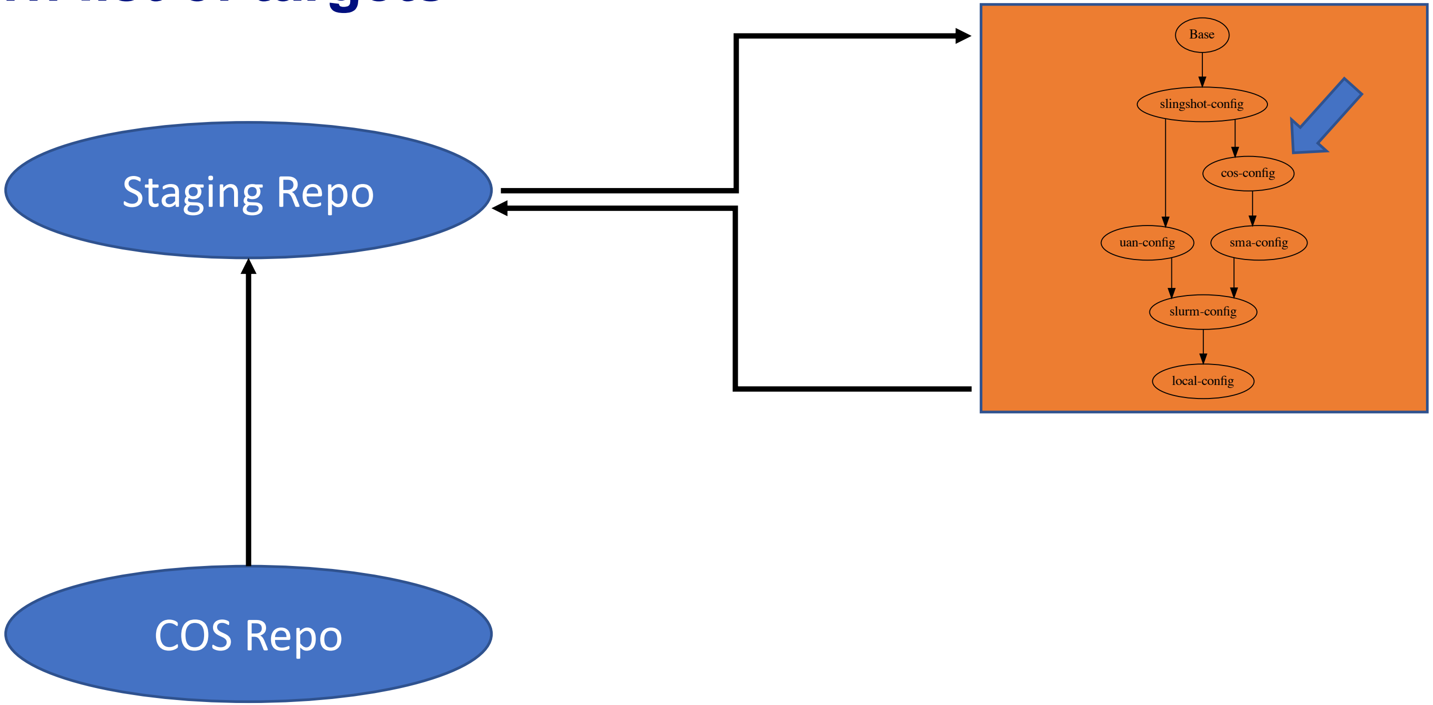
Staging Repo

COS Repo

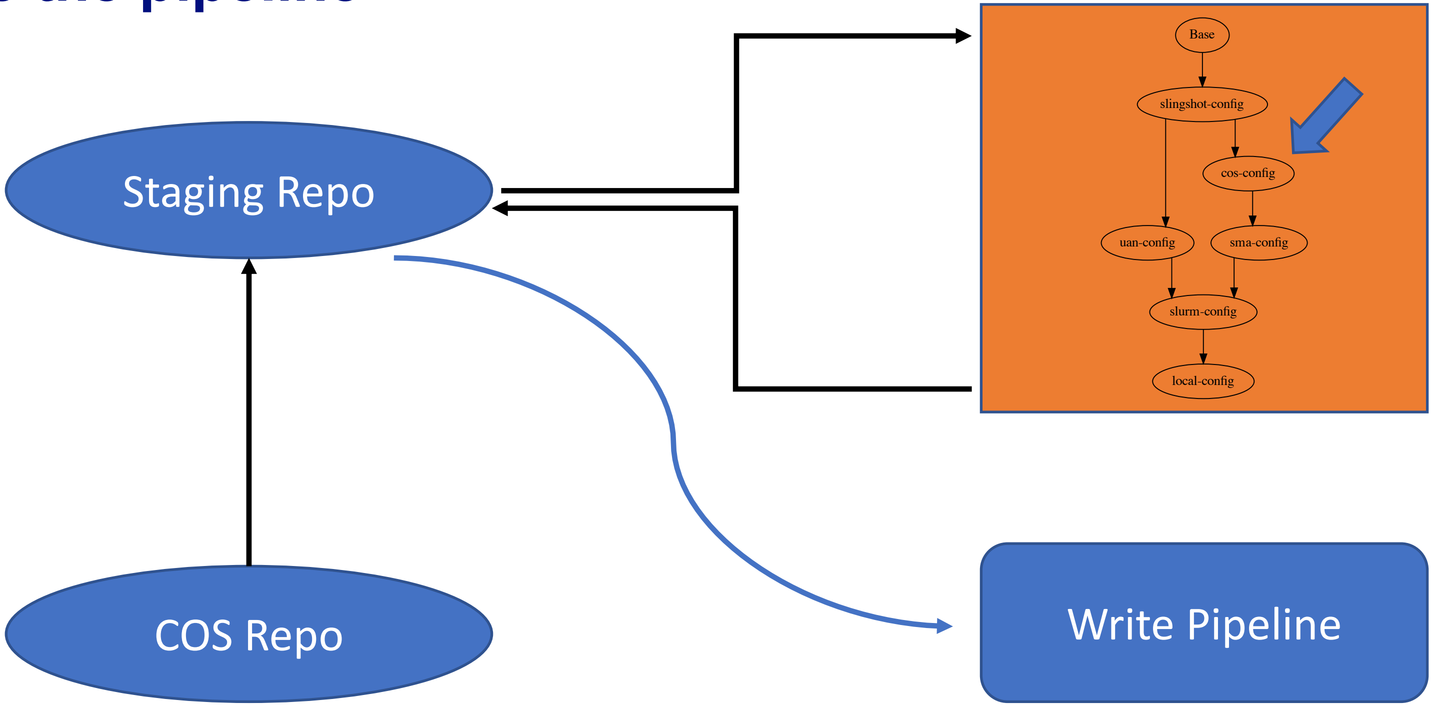
Target is found



Return list of targets



Write the pipeline



Staging Repo

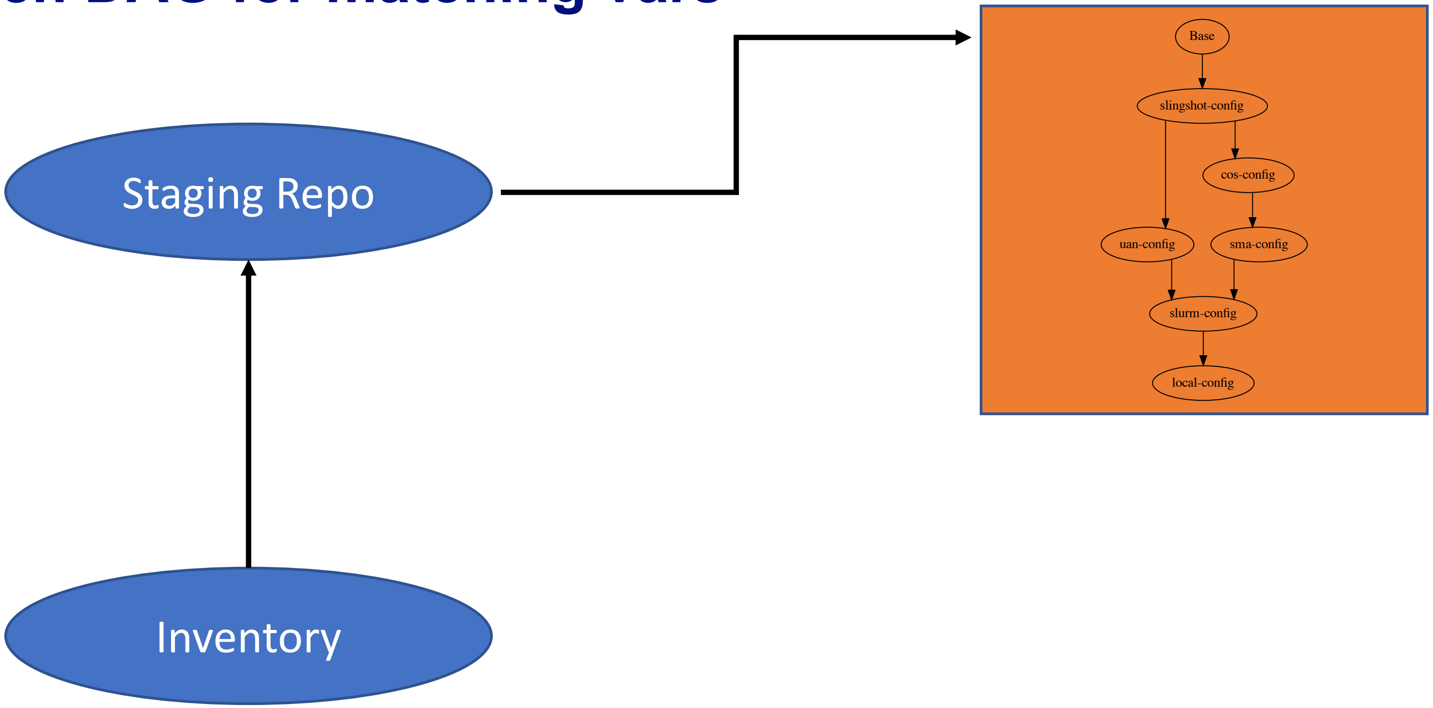
COS Repo

Write Pipeline

Inventory type trigger

- Also still a work in progress
  - Still file based
  - Variables defined from other variables still needs work
- Map changed variable files to roles repos
  - Don’t inherently know where to start the pipeline
  - Iterate through layer dependency DAG
    - Check each layer type’s defined playbooks
    - Search for changed variables used in roles/tasks
    - Search templates
- If no match is found, no layer pipeline

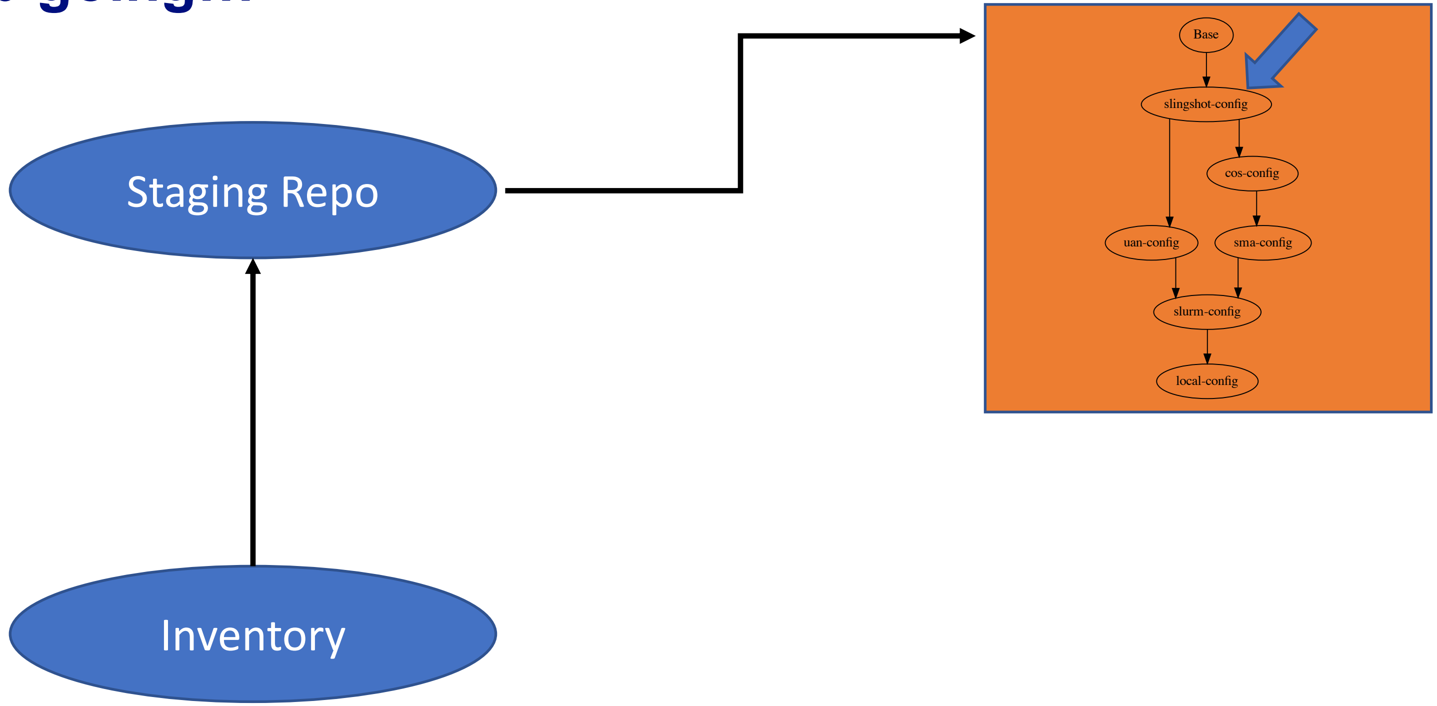
Search DAG for matching vars



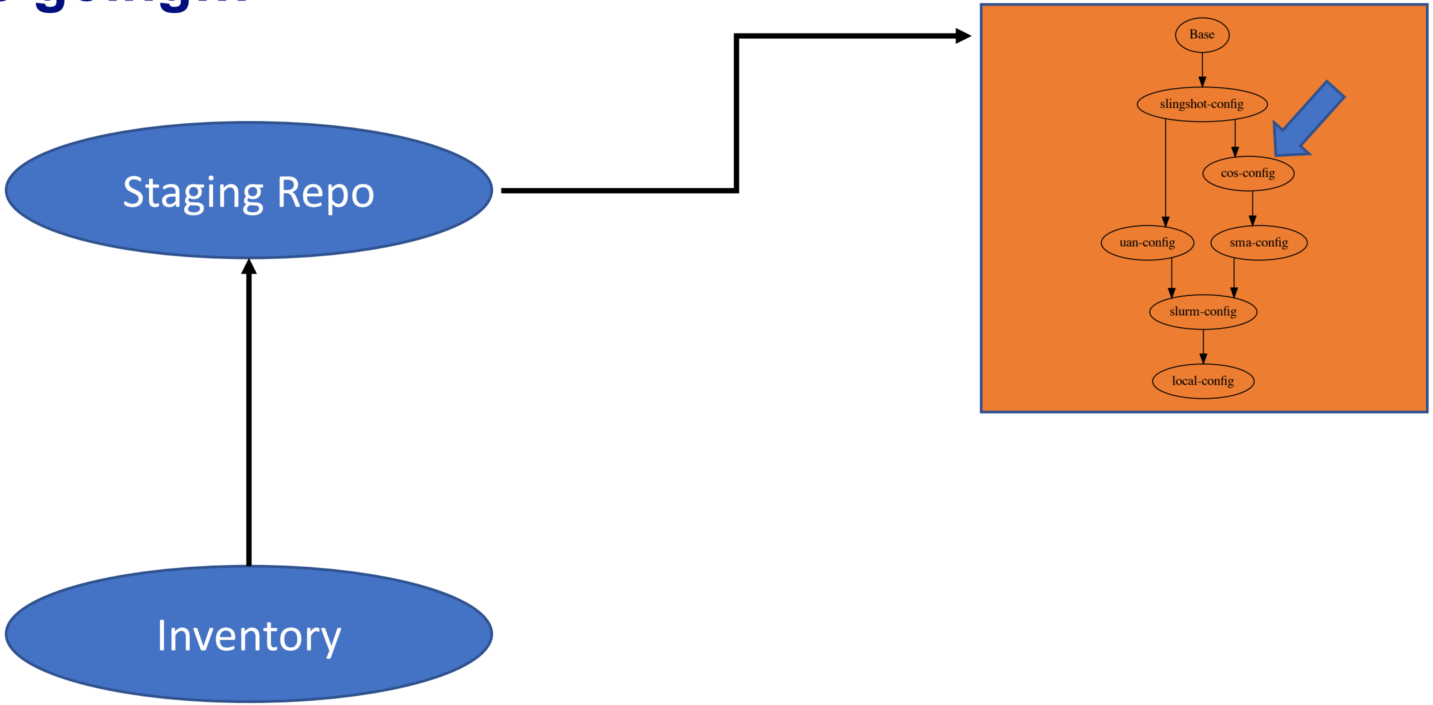
Staging Repo

Inventory

Keep going…


Staging Repo
Inventory
Base
slingshot-config
cos-config
uan-config
sma-config
slurm-config
local-config

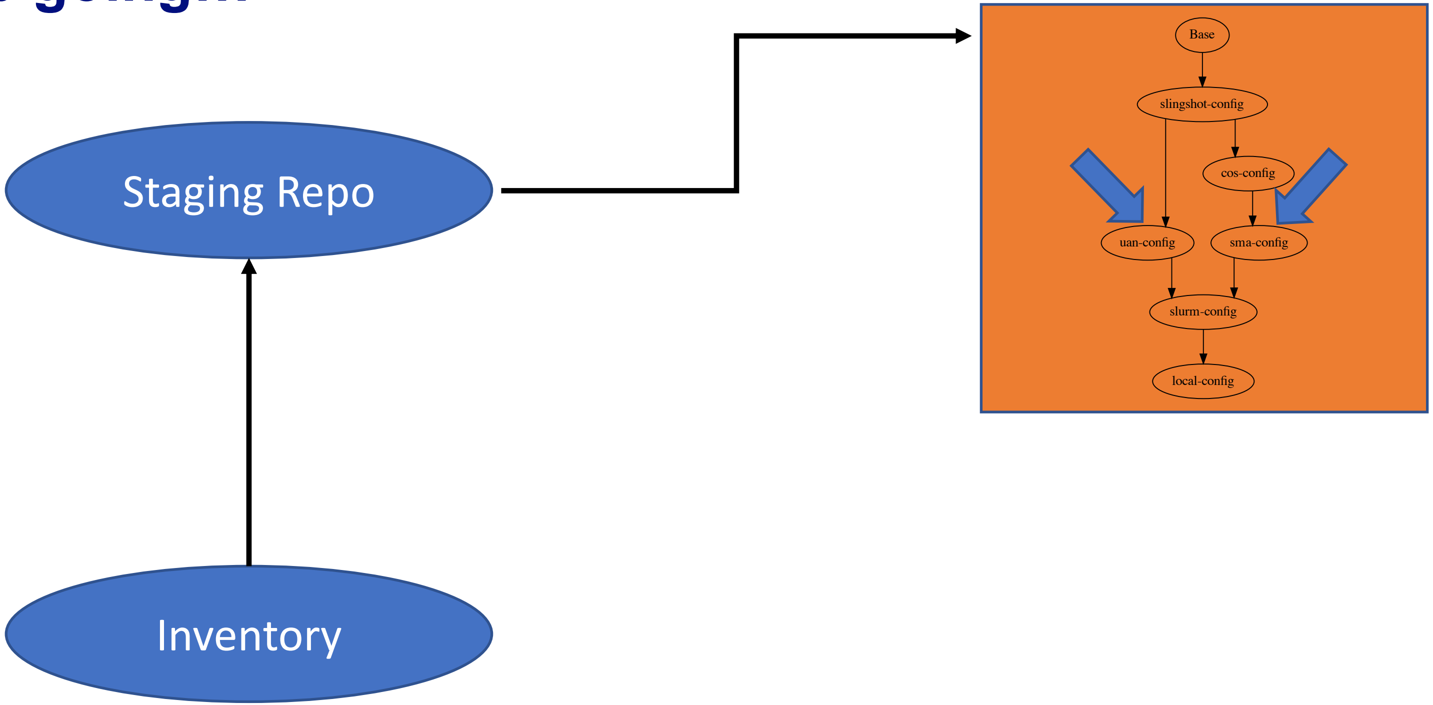
Keep going…



Staging Repo

Inventory

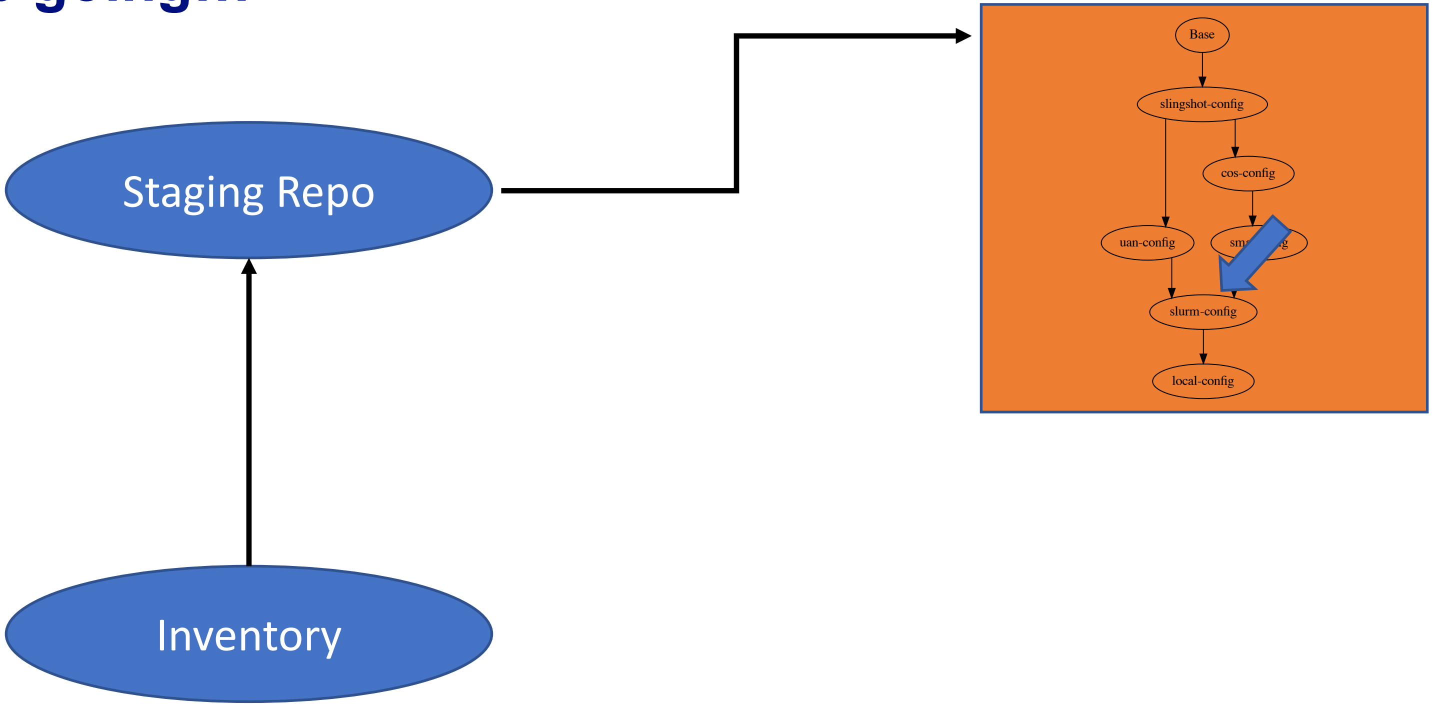
Keep going…



Staging Repo

Inventory

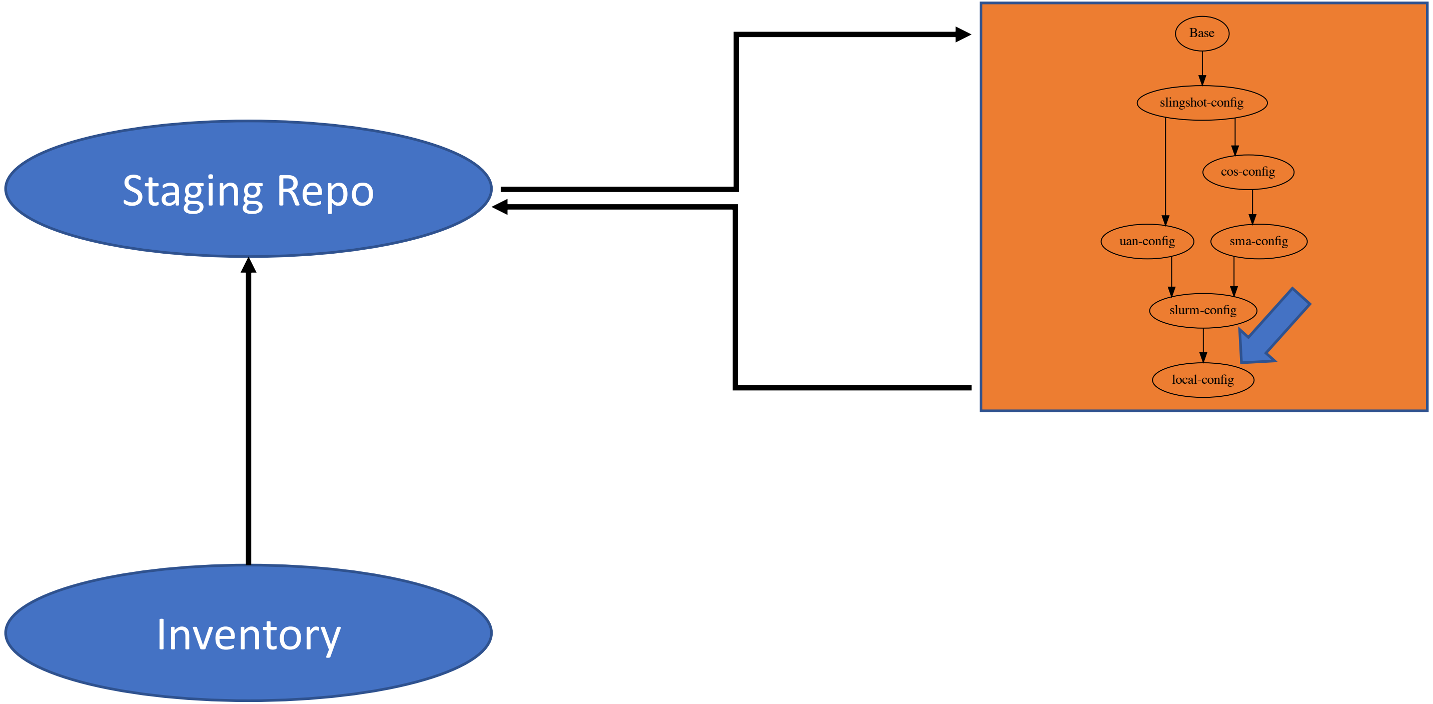
Keep going…



Staging Repo

Inventory

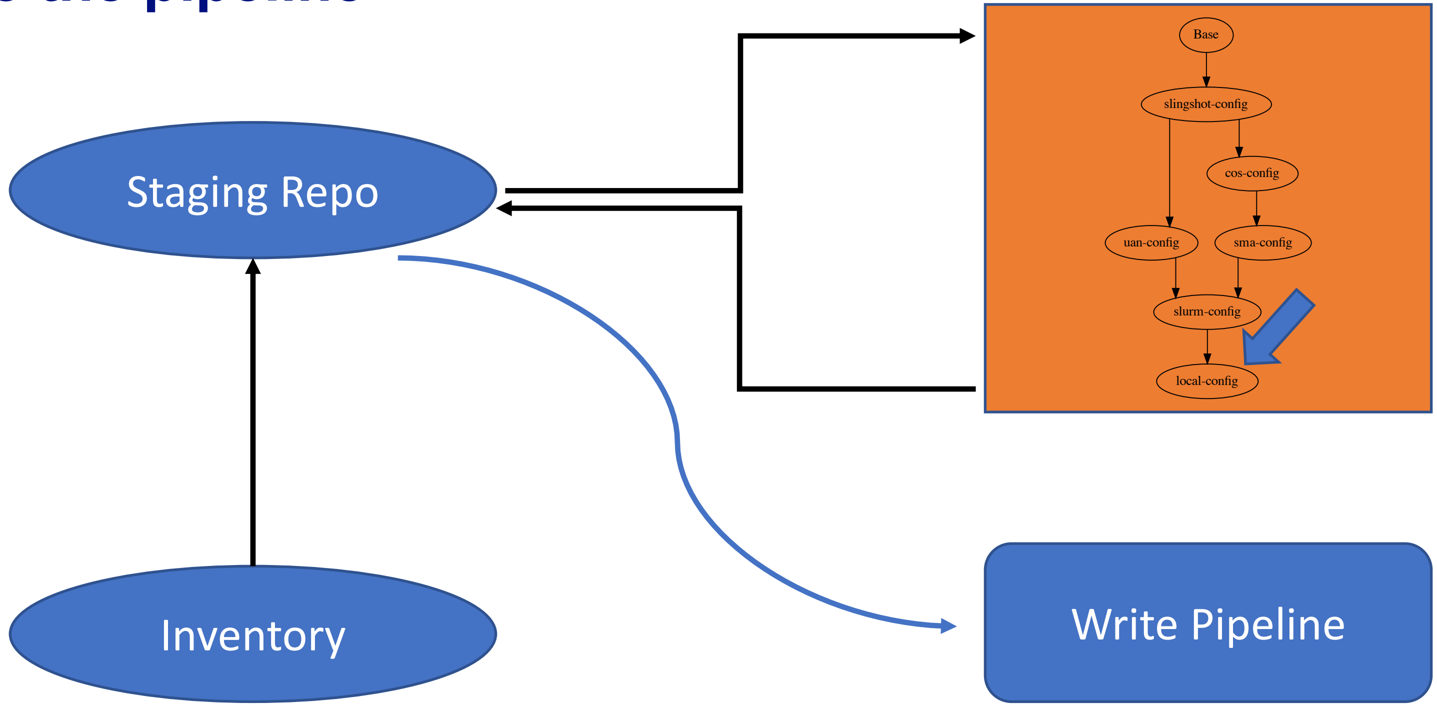
Until a match is found



Staging Repo

Inventory

Write the pipeline



DAG definition file

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

How the DAG is built and used

- Uses the definition file as source
  - Can have separate files for each system
  - Joined together during build
- Use networkx
  - Adds edges from parent -> child
  - Adds layer info (playbooks, repo, etc) to each node
  - Used during search for matching repos/changes
- Recursively iterate over neighbors

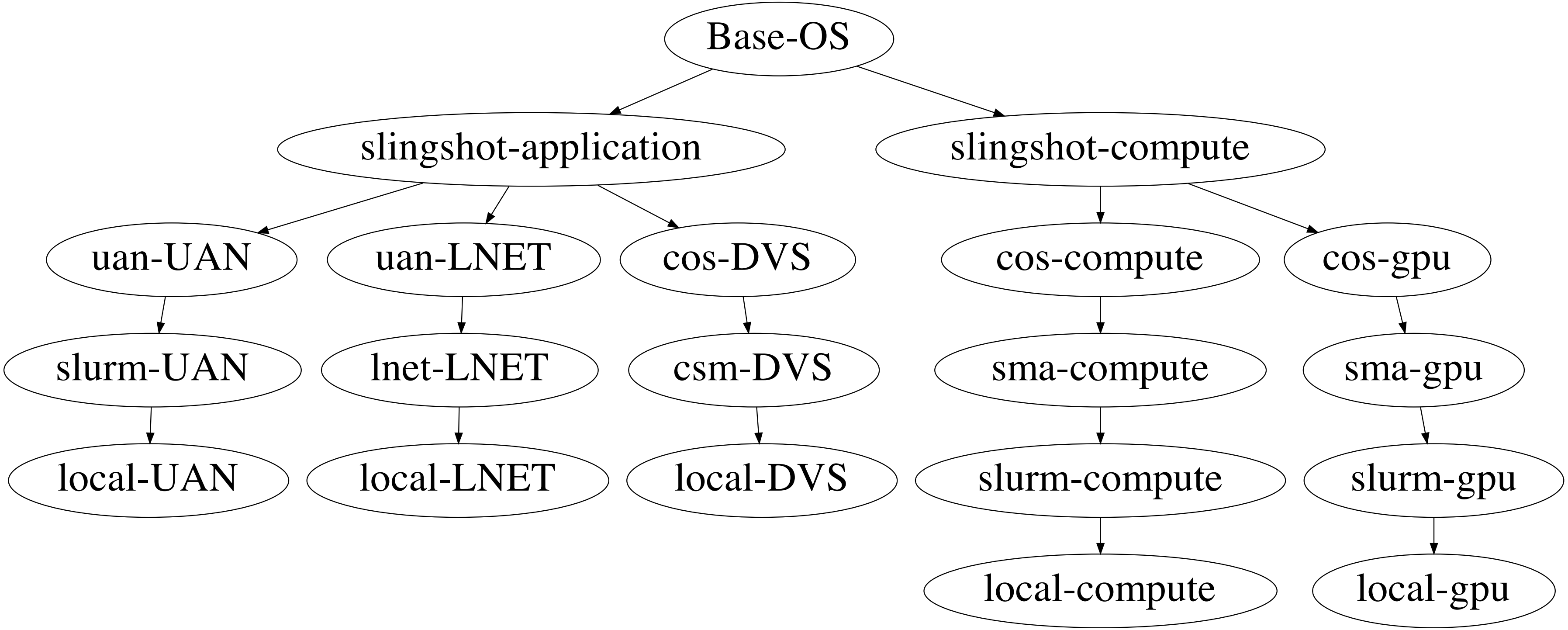
Automated Pipeline - recap

- Keep source of truth in one place
- Don’t need to manage complicated pipeline configs
- Configs are written for you
- Only write pipelines for layers where changes matter

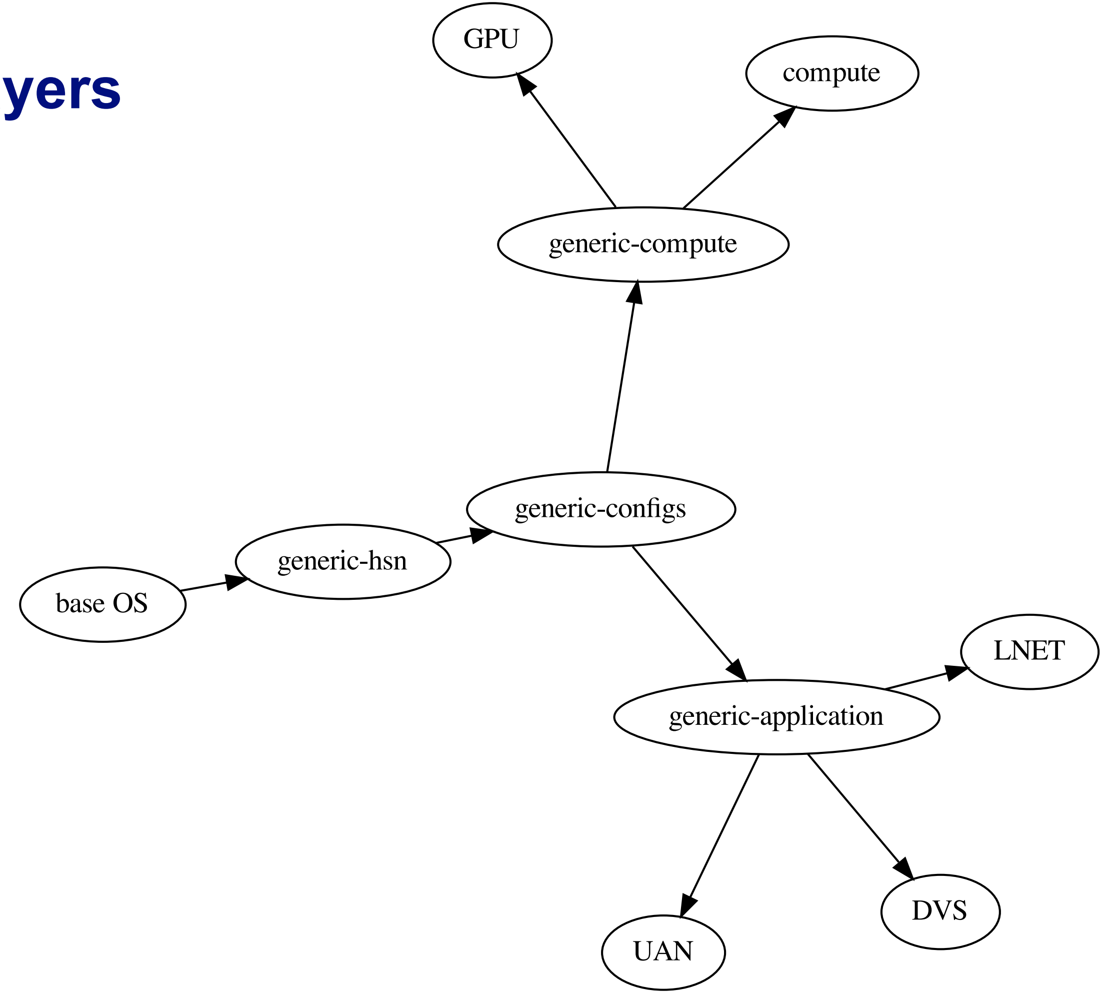
Part 3:Observations and Benefits

- Turnaround Time
  - Don’t build layers unnecessarily
  - Target layers where it matters
  - Easy to debug
    - Run ansible repeatedly, quickly
  - Build layers for multiple systems
  - Current CFS based DAG branches early in layer dependency DAG
    - Would like to push generic configs to earlier in DAG

CFS layers – simplified a little



Example layers



Part 3:Observations and Benefits

- Portability and Scaling
  - The goal is to be able to run it anywhere (that makes sense)
    - In Kubernetes
    - In a podman container
    - Some other fancy container thing
  - Can run as many layer builds as available runners
    - Can potentially scale up and down
    - Would like to do layer builds separate from systems

Part 3:Observations and Benefits

- Consistency and Validation
  - Changes can’t sneak in anymore
  - Helps mitigate human errors
  - Layers unchanged unless configs updated
  - Shared layers are the same across multiple systems
  - Breaking changes are caught in the pipeline

Part 3:Observations and Benefits

- It works!

Part 3:Observations and Benefits

- It works!
  - Mostly

Part 3:Observations and Benefits

- It works!
  - Mostly
  - On test systems
- Debugging ansible with buildah is so nice
  - Currently working on a debug tool
- Currently pushing directly to s3
  - Feels icky and scary
  - Use something like quay

Future work

- Expand base type images
- Validate Inventory type changes
  - Move away from file based
  - Complicated variable definitions
- Integrate other pipelines
  - Ansible linting
  - Ansible syntax checks
  - Automated testing – boot images from pipeline and run tests
  - Auto-deploy with rolling updates?
- Open source it? Would you like to know more?

Questions?