This solution is logically divided into two parts - *continuous integration* responsible for code development and management, as well as building and testing container images; *continuous deployment* responsible for supplying Kubernetes manifest code and applying manifests to one or more Kubernetes clusters.

Although the stated goal is to deploy the *tag-rc-2* branch, the continuous integration (CI) pipeline will consider all branches and all changes. A Jenkins Multibranch Pipeline can be configured to monitor the *trunk* branch for changes and take action on specific events, such as a pull request:

on:

# Trigger a workflow on pull request for trunk branch

pull\_request:

branches:

- trunk

Using Jenkins Pipeline stages developers who need to be involved with the CI process and want action automatically based on predefined conditions to produce a deterministic set of outcomes.

The Jenkins pipelines will be configured into multiple stages, which will:

1. Pull updates from the repository corresponding to the triggering branch
2. Build a container image, such as a Docker container, from the repository code
3. (optional) Perform testing of the image to ensure consistency
4. Authenticate to a container repository
5. Push the image to the container repository
6. Update a values file with data for deployment, such as the image tag

By default Jenkins is self-contained with GITHUB, however, Jenkins slaves are supported to address scaling and cost considerations at scale. Jenkins plugins can be configured to execute a background process that communicates with Github to watch for triggers and can support organization-wide or per-repository Actions as required. Jenkins slaves can be configured with Cloud technologies, such as AWS Compute Autoscaling Groups or GCP Compute Managed Instance Groups, to scale out and scale in automatically in response to build activity.

Because all branches will be considered by the Multibranch pipeline, metadata can be used to further refine decisions to ensure that only the intended changes result in a new container image. Examples include branch labels, pull request comment text, and release events.

Continuous Deployment (CD) will be informed by CI changes and take action as necessary to update the Kubernetes cluster. Software from the Jenkins project will be configured to monitor the Github repository for changes, such as a values file describing the image tag to be deployed, and take action to rationalize the actual state to the desired state when a deviation is found. This model is flexible to allow for container images to be built and pushed to a repository that are not to be deployed by ensuring that only changes documented in a values file are considered. Working in conjunction with the CI process earlier, a Jenkins “stage” could be written to build, test, and push a container image from every branch that contains the string “feature” but lacking an update to the values file these images would not be deployed.

Jenkins consumes YAML-formatted template files in a Helm-compatible structure to render deterministic Kubernetes manifests that are applied to one or more target clusters. The software can be run on the same Kubernetes cluster where changes will be applied, for a self-contained environment, or remotely from the target cluster(s) as dictated by the security posture of a business.

Additional “stages” can be configured in Jenkins for *prune* and *heal* which respectively remove resources no longer contained in the manifest and update the live Kubernetes objects to match expected. These two actions strongly support an *as code* model by ensuring that the code in a monitored software repository always matches the code running on Kubernetes.

Jenkins Pipelines are lightweight and can support many concurrent changes simultaneously without scaling, however, scaling options are also available to ensure that no artificial limits are imposed on the CD pipeline.

When deploying non-unique tags, such as by strings *latest* or *tag-rc-2*, it’s important to note that updating a values file with the tag string will result in only a single deployment, the first, since subsequent changes to the image are undetected by Jenkins. This can be solved via CI pipeline by either updating an additional field, such as an image SHA label, that is unique to each image built or by incrementing a version number, such as a patch version with each changed image. When such a field is monitored by Jenkins, changes to the field will result in update actions being taken to freshen the image used. Methods for solving this requirement should conform to business processes and expectations.

In this model, stemming from each code commit, a series of automated logic gates are traversed ending either in no change to the cluster, such as when a branch name is not considered by Jenkins, or when all stages are passed an update action being taken on a Kubernetes cluster to ensure that current and desired states are a match. To minimize friction, these stages should match the natural patterns of a development team. For example, a team who uses a release review process but does not review pull requests might be better served by Jenkins that monitor only releases.