Published in Better Programming

Wenqi Glantz Follow
Jan 18 · 10 min read · ✦ · Listen

DevOps Self-Service Centric GitHub Actions Workflow Orchestration

How to orchestrate GitHub Actions workflows driven by image immutability

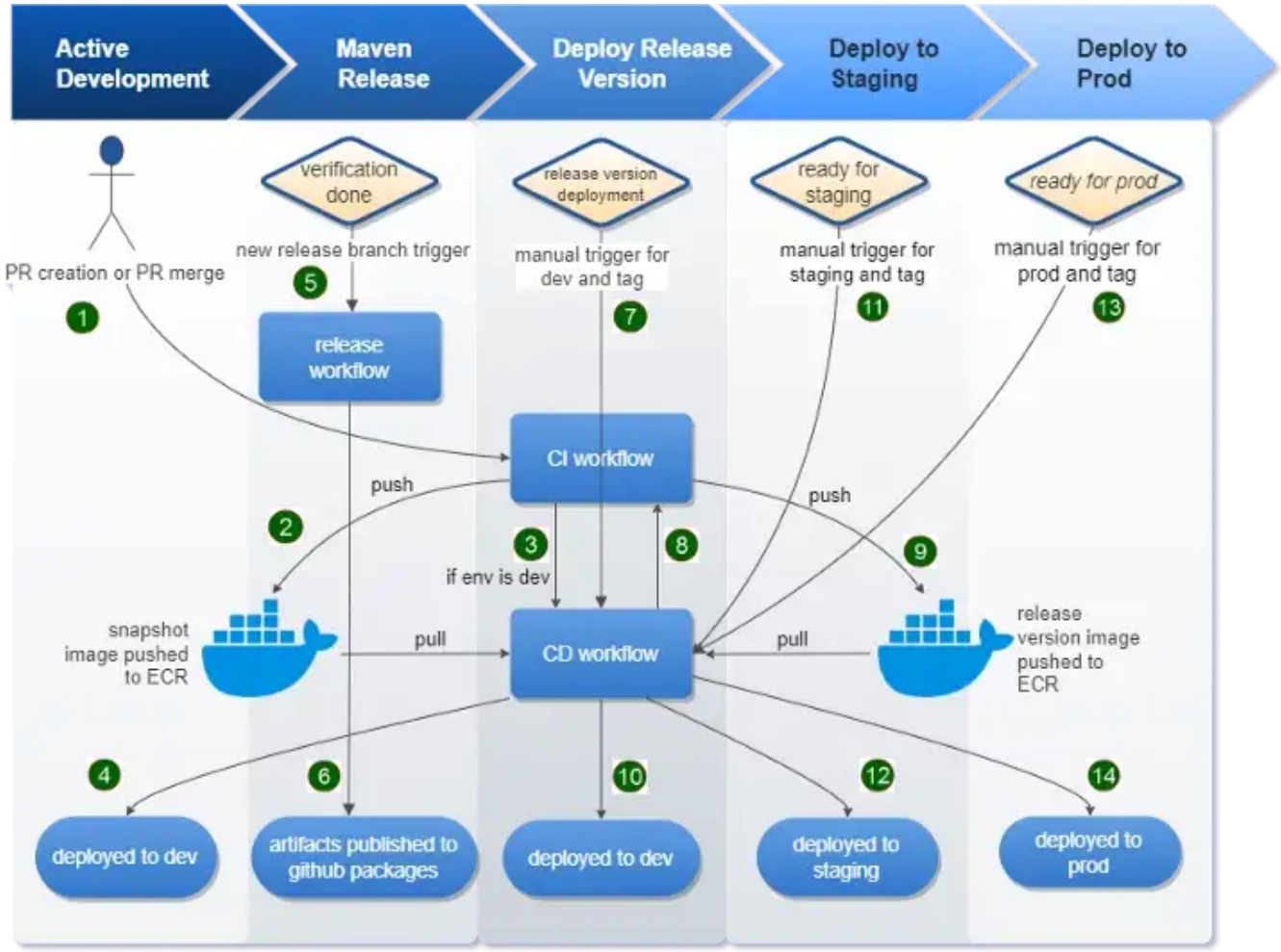

Diagram by author

GitHub Actions

The adoption of GitHub Actions has been increasing in recent years. According to a survey by The Software House for the state of frontend, GitHub Actions takes the front seat in CI/CD tools, with over 56% in 2022 compared to 35% in 2020. This shows that more developers shifted to GitHub Actions as their CI/CD tool in their day-to-day.

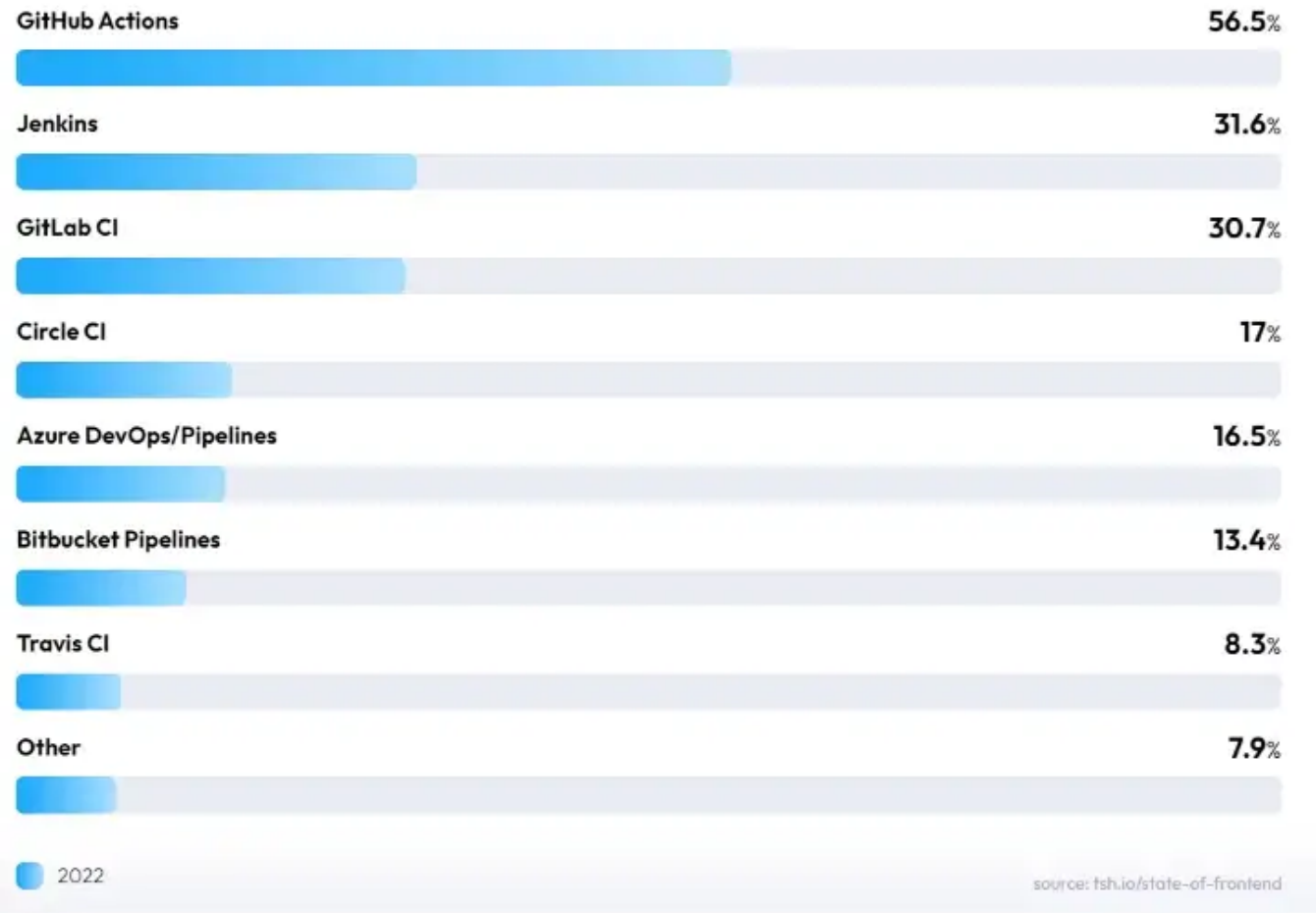

Image source: https://tsh.io/state-of-frontend/

Guided by the DevOps self-service pipeline architecture, we explored Terraform project structure and its reusable modules in our previous story. This article will dive into how to orchestrate application pipelines with GitHub Actions. We will focus on the red highlighted rectangle in the diagram below.

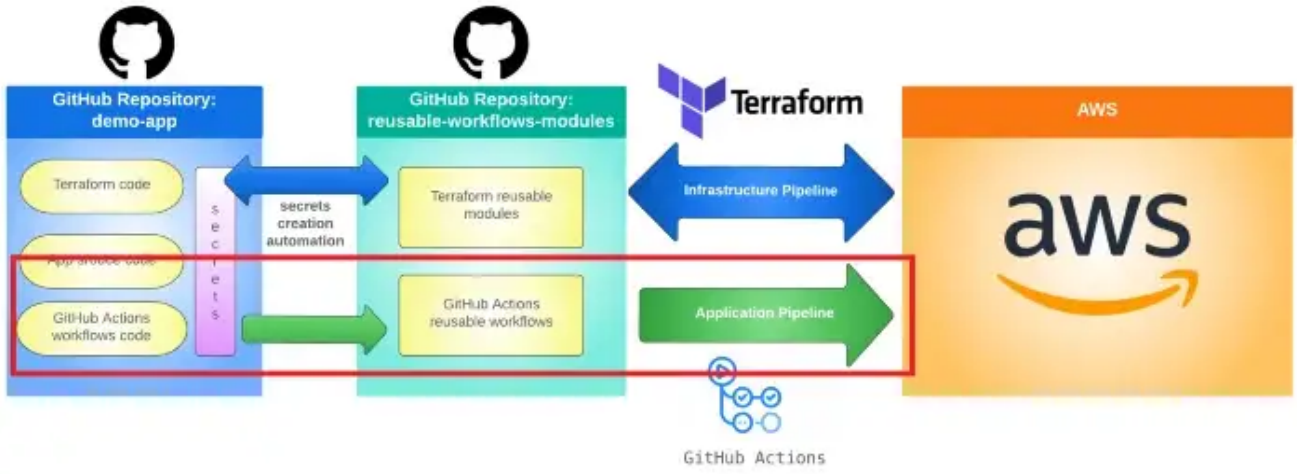

diagram by author

GitHub Actions Reusable Workflow

In DevOps' self-service centric practice, reusability is key. Just like reusable Terraform modules, we can have reusable GitHub Actions workflows in our application CI/CD pipelines.

From a stateful/stateless perspective, reusable workflows are stateless. Caller workflows from many different applications can call one reusable workflow.

I published an article a few months ago titled A Deep Dive into GitHub Actions' Reusable Workflows, in which I explored the detailed steps to make a GitHub Actions workflow reusable, how to call a reusable workflow, and how to pass input parameters and secrets to the reusable workflow. Sample code was shared in that article, and it is listed at the bottom of this article. I highly recommend you look at that article if you are new to GitHub Actions' reusable workflow.

Once you start using reusable workflows in your projects, you will never look back, as reusable workflows save so much time and effort in rolling out your apps' CI/CD workflows and eliminating maintenance headaches.

Container Image Immutability

Container image immutability refers to creating and using container images that cannot be modified after they are built. Once an image is built and pushed to a registry, it should not be modified. Instead, a new image should be built with any desired changes and deployed.

The main benefit of container image immutability is its predictability. When image immutability is enforced, you can be sure that your application behaves as predicted across environments such as staging and prod. Image immutability also offers you the peace of mind to roll back to a previous image tag in case of errors because you know the previous image tag is immutable and has not been tampered with by builds after its release. Overall, immutability helps in ensuring consistency, security, and reproducibility of container images.

Image Immutability Challenges

Triggering CI before each CD for each environment does not guarantee image immutability. Why? Each CI could produce a different image even though you have not changed your source code. If you have dependency upgrades and auto-merge automated by tools such as GitHub dependabot, your source code varies depending on when your latest dependency upgrades took place. Because of that, the image

built a week ago and the image built today could be two different images despite no manual code changes in between.

To enforce image immutability, we have to separate CI from CD, making them into two isolated workflows. CI builds and pushes the image to ECR, and CD needs to be smart enough to know exactly which image tag to use to pull that immutable image. Teams who have been used to using date timestamp or Git SHA as part of the image tag now face a dilemma: how do you know which image tag to use when you trigger CD? You could try to trace your CI workflow debug log to figure out what image tag was published to ECR, but it defeats the purpose of automation. Any manual effort in this flow is not a desirable solution.

It ultimately boils down to — how do you ensure your image tag is immutable?

Read on to find out.

Maven Release Automation

Assume you are developing a Spring Boot app using Maven as a build tool, and your app will be deployed to three environments: development, staging, and prod. Maven Release automation is the key to maintaining container image immutability. Let’s take a closer look at how Maven Release works.

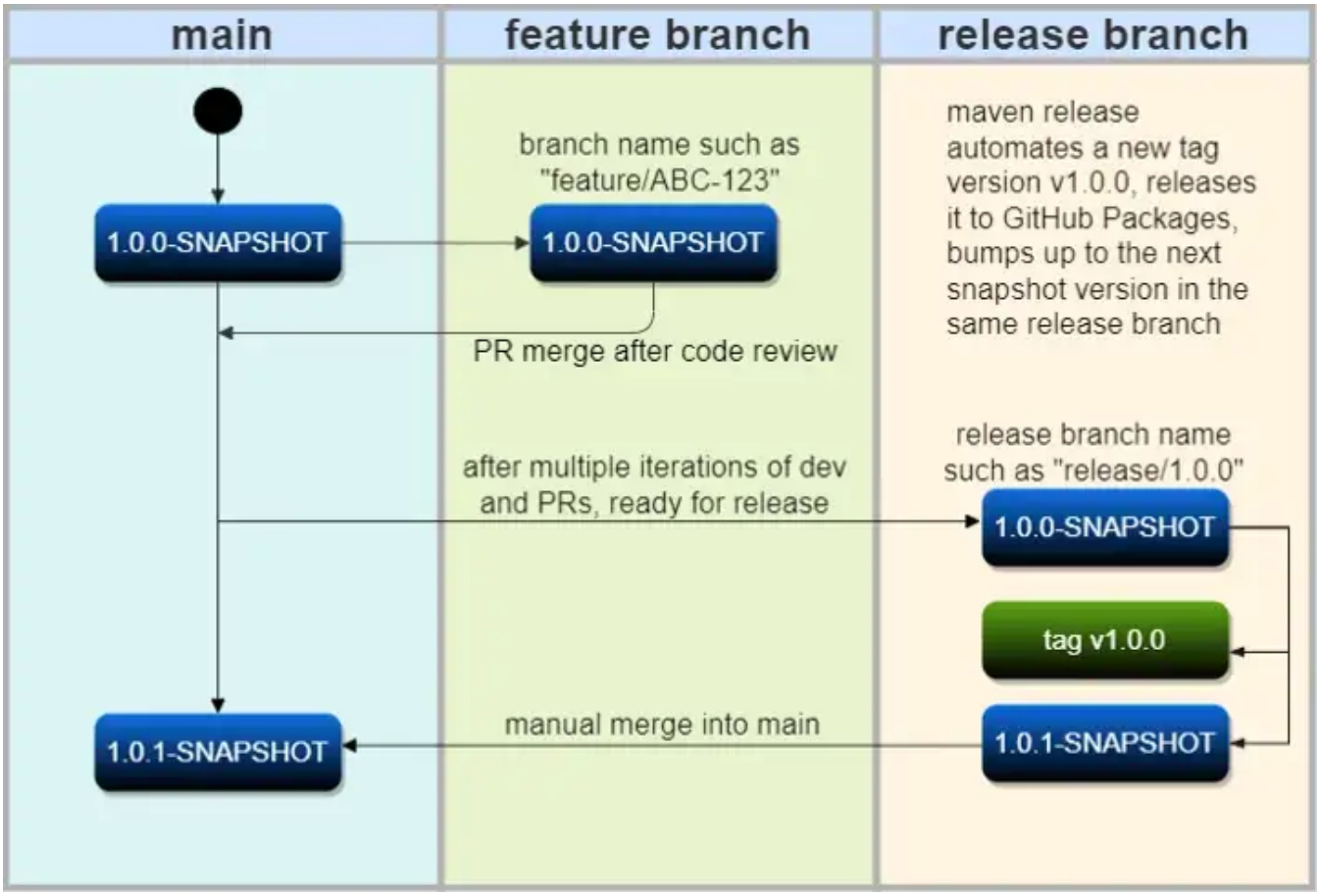

diagram by author

The diagram above should be pretty self-explanatory. Maven Release automates tag release and bumps up pom version to the next development version. If you express the above Maven Release flow in a GitHub Actions reusable workflow, you'll have the following sample release workflow.

```yaml
name: Release workflow for Spring Boot microservices or shared libraries

on:
  workflow_call:
    inputs:
      # working-directory is added to accommodate monorepo.  For multi repo, defaults t
      working-directory:
        required: false
        type: string
        default: '.'

jobs:
  release:
    runs-on: ubuntu-latest

    # accommodating monorepo, this sets the working directory at the job level, for mul
    defaults:
      run:
        working-directory: ${{ inputs.working-directory }}

    # default to dev env for publishing release version to ECR as AWS credential is tie
    environment: 'dev'

    # run release flow only if the triggering branch starts with "release/"
    if: startsWith(github.ref, 'refs/heads/release/')
    steps:
      - name: Harden Runner
        uses: step-security/harden-runner@ebacdc22ef6c2cfb85ee5ded8f2e640f4c776dd5
        with:
          egress-policy: audit # TODO: change to 'egress-policy: block' after couple of

      - name: Checkout Code
        uses: actions/checkout@d171c3b028d844f2bf14e9fdec0c58114451e4bf

      - name: Cache local Maven repository
        uses: actions/cache@9b0c1fce7a93df8e3bb8926b0d6e9d89e92f20a7
        with:
          path: ~/.m2/repository
          key: ${{ runner.os }}-maven-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-maven-

      - name: Setup jdk
        uses: actions/setup-java@19eeec562b37d29a1ad055b7de9c280bd0906d8d
        with:
          java-version: 17
          distribution: 'adopt'
          cache: maven
          # this action creates a maven settings.xml file as well, server-id correspon
```

```
          # this action creates a maven settings.xml file as well, server-id correspond
          server-id: github

      # create a git user to push to github automated pom snapshot release, next versio
      - name: Configure Git user
        run: |
          git config user.email "actions@github.com"
          git config user.name "GitHub Actions"

      - name: Release JAR
        run: mvn -B release:prepare release:perform
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          # recommended by github as sometimes github may throw 500 internal server err
          MAVEN_OPTS: -Dhttp.keepAlive=false -Dmaven.wagon.http.pool=false -Dmaven.wago

      - name: Rollback if failure
        if: failure()
        run: mvn -B release:rollback
```

Keep in mind that Maven Release version is immutable. So, how is Maven Release related to container image immutability? You guessed it — release version number as container image tag! How does that work exactly? Let's continue the exploration.

GitHub Actions Workflow Orchestration

Let's say you have developed/compiled a list of stateless reusable workflows for your app for CI, CD, and release. How do you tie these workflows together so they can be orchestrated to compose that masterpiece tune for your CI/CD?

Let's start with this workflow orchestration diagram below:

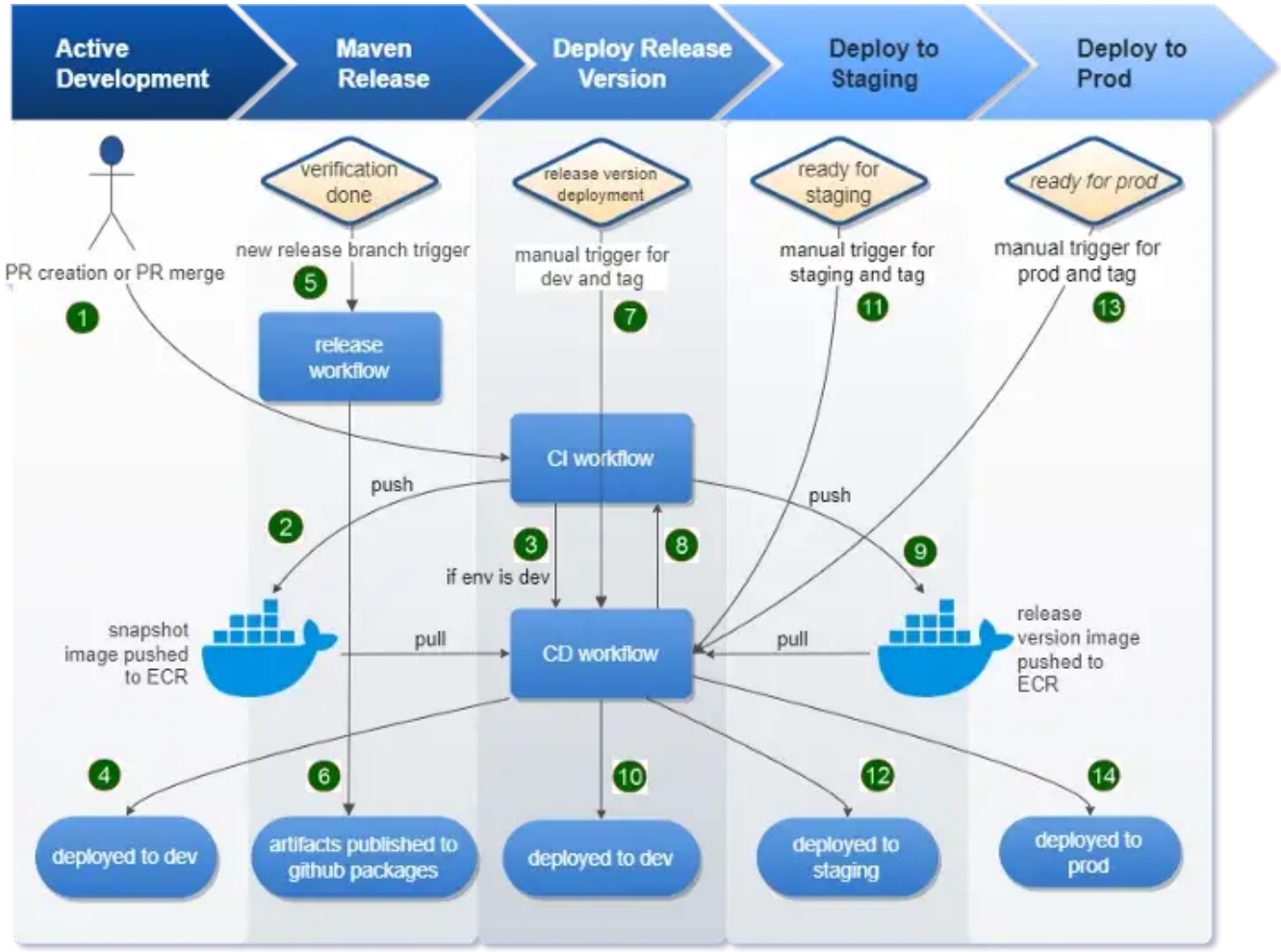

diagram by author

Can you hear there is a symphony going on? :-) Let's trace through the steps:

1. When you raise a new pull request during active development, it auto-triggers your CI workflow.

2. The CI workflow does the build, test, docker image build, and push to your container registry, such as ECR, and it also completes the image scan for vulnerability. Please note that the image tag here uses the project version defined in your application pom.

3. After multiple rounds of PR review and code fixes, your PR finally got approved, and you are ready to merge your PR to your default branch. Upon PR merge, it auto-triggers your CD workflow.

4. The CD workflow pulls your docker image from the container registry and deploys it to your predefined destination, such as ECS.

5. After many iterations of development, your app is ready to be promoted to staging environment. You create a release branch for a release candidate (RC)

version, which triggers the release workflow to automate Maven Release.

6. RC artifact gets published to GitHub Packages or any other artifact registry of your choice with proper configuration in place.
7. Once Maven Release is successful, you need to trigger CD workflow to deploy the RC version to Dev via the manual trigger, where you can select the dev environment and the tag (not the main branch). Notice you need to select the tag in the dropdown, which was just released by Maven Release at step 6.
8. CD, in turn, triggers the CI workflow as you now need to build the new RC version image.
9. The CI workflow does the build, test, docker image build, push to ECR, and scans images for vulnerability.
10. CD pulls the RC version image from ECR and deploys it to Dev.
11. When you deploy the RC version image to Staging, you trigger CD via the manual trigger by selecting Staging env and the RC tag (not the main branch).
12. CD pulls the RC version image from ECR and deploys it to Staging. Steps 5–12 could iterate multiple rounds depending on your application development status, bug fixes, etc., until you are finally ready to release the final version (without RC). Go through steps 5–12 to release your final release version and deploy it to Dev and Staging.
13. When you deploy the release version image to Prod, you trigger CD via the manual trigger by selecting Prod env and the final release tag (not the main branch). I suggest having GitHub deployment protection rule configured to ensure proper approval chain takes place before the Prod deployment can be actually triggered.
14. CD pulls the release version image from ECR and deploys it to Prod.

A few key observations:

CI workflow is only called twice in this whole lifecycle. The first is from PR creation/merge during the active development phase. The second call is from the CD workflow after the Maven Release, which has removed the snapshot, CI

workflow is triggered to build the RC version image or the final release image, which gets deployed to all environments.

You could have multiple iterations of snapshot image push and RC version push, but each Maven Release will be incrementing their numbering for RC version or release version, making each of such release version immutable. With the Maven Release, it's guaranteed you can not release the same version (whether RC or release version) to your artifact registry twice. Otherwise, you will be running into a 409 error.

Maven Release Candidate (RC)

During multiple Staging deployment rounds, you should not keep bumping up the patch release number in your Semantic Versioning (SemVer, major.minor.patch). The release candidate (RC) version is introduced for your Staging deployments for as many rounds as needed without impacting your SemVer when your app is released to Prod.

Let's take a different angle and look at how those RC versions are managed between the main branch and your release branches. Hopefully, it gives you a better idea on how Maven Releases ties into container image immutability.

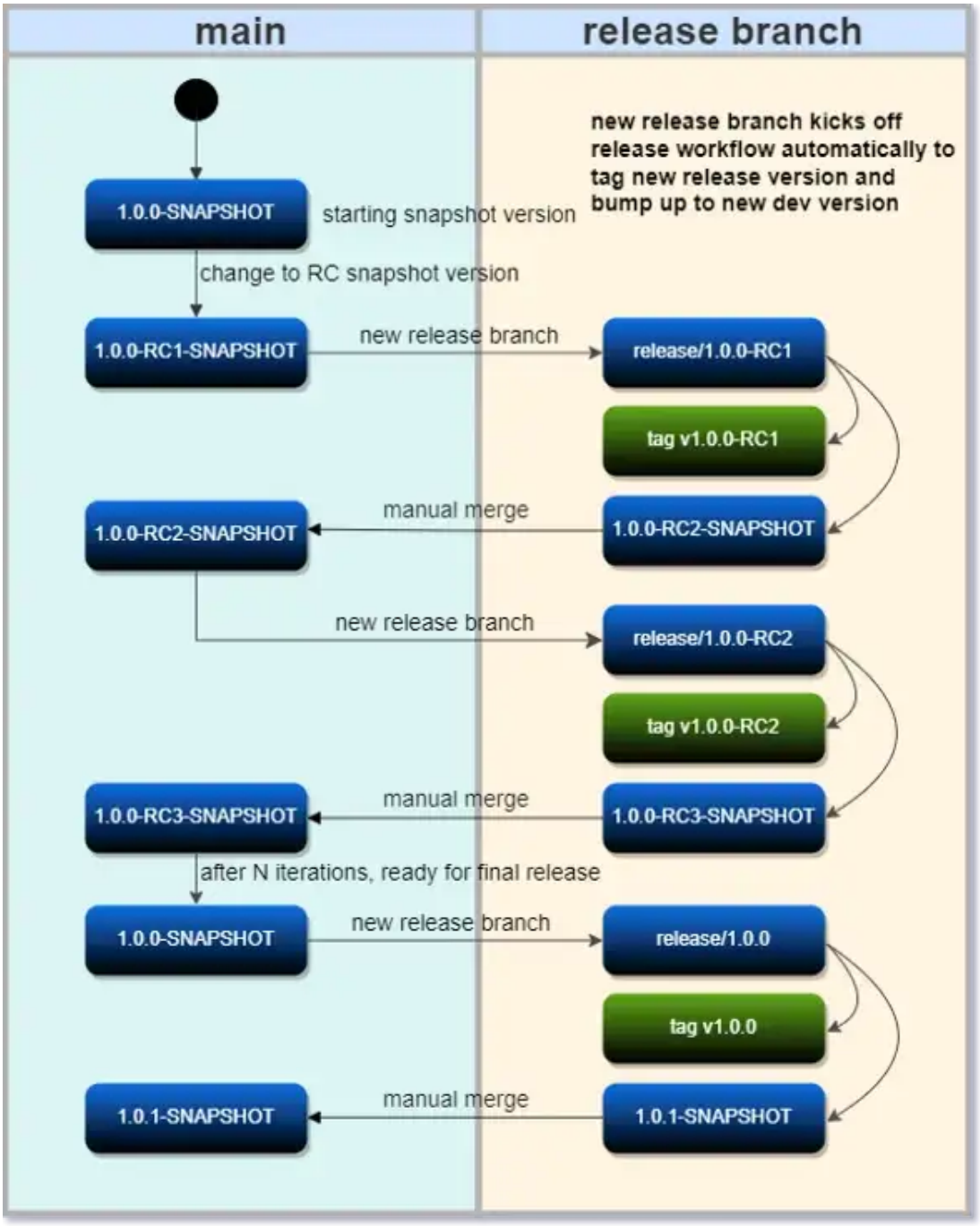

diagram by author

Image Immutability Implementation in Action

As you can see from the orchestration diagram above, there are three workflows involved: `ci.yml`, `cd.yml`, and `release.yml`. These workflows are all caller workflows, which calls reusable workflows. Let's take a look at what the caller workflows look like:

`ci.yml` is straightforward with no tricks. It is a simple workflow calling a reusable workflow, and it's triggered by creating a PR or pushing code into the PR.

```yaml
# This CI workflow can be triggered by PR creation or code push in PR, or manual trigge

name: CI workflow for building, testing microservice, and publishing image to ECR

on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment to run the workflow against'
        type: environment
        required: true
  pull_request:
    branches: [ main ]

permissions:  # added using https://github.com/step-security/secure-workflows
  contents: read

jobs:

  build-and-test:
    permissions:
      id-token: write # need this for OIDC
      contents: read
    uses: wenqiglantz/reusable-workflows-modules/.github/workflows/java-maven-build-tes
    with:
      env: ${{ github.event.inputs.environment }}
    secrets: inherit
```

ci.yml hosted with ❤ by GitHub view raw

`cd.yml` is a bit more involved. See the following reasons:

- It's triggered by either a PR merge (in dev) or a manual trigger.
- If it's for dev environment or PR merge, it also triggers the `build-and-test` job, which is what CI workflow does. Why do we need this condition and job in CD workflow? The dev environment will be constantly updated with the latest image, which would have the SNAPSHOT in the pom version. If we don't perform the `build-and-test` step before the `deploy-to-ecs` step, there is no guarantee that developer A's image will not be overwritten by developer B's image, as they both share the same image tag, the SNAPSHOT version.
- If it's for environments other than dev, the `build-and-test` job gets skipped because of that conditional statement on line 22 (see below the code). Line 34

ensures this `deploy-to-ecs` job is run only when PR is merged in dev, or for other environments via manual trigger, and PR is not raised by dependabot. This is where the image immutability implementation is in action. Take a closer look at the orchestration diagram above. Staging and Prod only trigger CD workflow, which pulls the RC or release version image from ECR and deploys it.

```yaml
# This CD workflow pulls the docker image from ECR and deploys it to the environment sp

name: CD workflow for deploying microservice to ECS Fargate

on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment to run the workflow against'
        type: environment
        required: true
  pull_request:
    types: [closed]  # when PR is merged, CD will be triggered

permissions:  # added using https://github.com/step-security/secure-workflows
  contents: read

jobs:

  # only run build-and-test job for dev env, this is to ensure the latest code from the
  build-and-test:
    if: github.event.inputs.environment == null || github.event.inputs.environment == '
    permissions:
      id-token: write # need this for OIDC
      contents: read
    uses: wenqiglantz/reusable-workflows-modules/.github/workflows/java-maven-build-tes
    with:
      env: ${{ github.event.inputs.environment }}
    secrets: inherit

  deploy-to-ecs:
    needs: build-and-test
    # runs deploy job only when PR merged in dev, or in other envs via manual trigger,
    if: (github.event.pull_request.merged || inputs.env != null) && github.actor != 'de
    permissions:
      id-token: write # need this for OIDC
      contents: read
    uses: wenqiglantz/reusable-workflows-modules/.github/workflows/java-api-deploy-to-e
    with:
      env: ${{ github.event.inputs.environment }}
    secrets: inherit
```

cd.yml hosted with ❤ by GitHub view raw

How does CD workflow know which version number to use as an image tag? Look in our CI and CD reusable workflows, where `PROJECT_VERSION` is discerned from the

code checked out (branch code for Dev env and tag code for Staging and Prod) by running `mvn help:evaluate` to extract its project version and then configure it as an environment variable for the image tag push and pull.

```
- name: Set project version as environment variable
  run: echo "PROJECT_VERSION=$(mvn help:evaluate -Dexpression=project.version -
```

`release.yml` is again really straightforward, triggered by either manual trigger or the creation of a branch with a naming convention starting with `release/`. After that, perform a simple call to the reusable workflow. See the sample below:

```
# This workflow runs maven release to publish artifact to GitHub Packages

name: Release workflow to run maven release to publish release version to GitHub Packag

on:
  workflow_dispatch:
  push:
    branches:
      - release/*
```



```
    uses: wenqiglantz/reusable-workflows-modules/.github/workflows/release.yml@main
```

release.yml hosted with ❤ by **GitHub** **view raw**

This concludes our workflow orchestration driven by container image immutability. I would love to hear any feedback to improve this orchestration. If you have a different way of handling workflow orchestration with image immutability, leave a comment for our readers and me.

All sample code can be found in my GitHub repositories:

- https://github.com/wenqiglantz/reusable-workflows-modules
- https://github.com/wenqiglantz/customer-service-reusable-workflows-example

Summary

This article focused on GitHub Actions workflow orchestration driven by container image immutability. We explored multiple topics such as reusable workflows, image immutability, the benefits and the challenges of image immutability in the pipelines, Maven Release automation, and how to tie all workflows together in the lifecycle of an application's CI/CD/release. I hope you found this article helpful.

Happy coding!

References

The State of Frontend 2022

Aleksandra Dąbrowska Report's Editor-in-Chief The last two years haven't been the easiest and prompted a lot of changes...

tsh.io

Top 5 CI/CD Tools to Look Out for in 2021

Automation and continuous integration/continuous development (CI/CD) can have a huge positive impact on how developers...

jfrog.com

Best practices for operating containers | Cloud Architecture Center | Google Cloud

This article describes a set of best practices for making containers easier to operate. These practices cover a wide...

cloud.google.com

Why I think we should all use immutable Docker images

medium.com

Dev Ops Github Actions Platform Engineering Reusable Workflow