**Semantic versioning strategies which I have hands on experience in different projects**

While working for different companies in different projects we need work on application or infra repository versioning or tagging. Here I’m trying to summarize couple of such strategies in which I worked personally. I hope it will the readers to get some idea and may they can adapt the same to their projects also.

**Application repositories:** Most of the application which I used to with Developers are Java spring boot. In one of the legacy project we used to maintain semantic versioning very well. The project have 15 repositories and all of them uses the same versioning strategy.

The git repo will have three types of branches: development branch, release branches, and feature branches. Development branch will always have the latest commits and it’s the main branch. Feature branches use to make changes and later to merge to the development branch. Release branches get create as part of the weekly releases.

Release branches we used to create using a naming convention as ‘release-<main version>.<sub version>.x . These applications use maven as build tool and hence we used to maintain the versions using pom.xml file. Pom file will have release version and a snapshot version. The snapshot is the future release version. When there is a release occurs our maven release script will look for main version in pom file and will create a release branch and the generated artifact will get tagged with patch version as zero. For example if the snapshot version mentioned as 5.46.0-snapshot, the build will create a branch release-5.46.x and build the artifact from source in this branch. Once the build completes the artifact will create with tag 5.46.0 and it will get pushed to our nexus. Also, the snapshot in pom file will get incremented as 5.46.1-snapshot.

If there is bug fix or patching needed for any released version we used to pull latest code from development and create changes in feature branch. Once the automated code analysis ran successfully we will cherry pick these latest commits to the release branch for which we need to release the patch version. So, suppose the current running version is 5.46.0 , we will checkout branch release-5.46.x and will do cherry pick of the bug fix commits to this branch and will run the build job in this release branch only. So we will get an artifact with 5.46.1 version and the snapshot version in pom file will get increment to 5.46.2-snapshot. In development branch the main version in pom file will always be next minor version snapshot, in this case it will be 5.47.0-snapshot . For a major version change, if there is a decision to update the core version then we update the major version number and corresponding snapshot version in pom file and merge to development branch prior to the build.

**Infrastructure-terraform repository versioning strategies:** In terraform repos we used to have master branch and feature branches to make changes. Once the feature got developed and ran a successful terraform plan, we merge it to the master branch. Then we used to draft a release with appropriate tag which we used to use as parameters in our pipelines. We use Jenkins to run our infra pipelines and apply the changes.

Feature branches we always used to create with a task ID but the release tags which we create from master branch will always semantic version strategy: <major version>.<minor version>.<patch version>

**Helm chart versioning:**

It is better to maintain semantic versioning for our helm charts also. We only need to change or update the chart version when there is a configuration, or template changes are there. For docker image tag updates, no need to change the helm chart versions. In most cases Chart.yaml file will have identical values for chart version and app version. But for some applications used to have docker image tag as app version and it can update based on latest docker image tag.