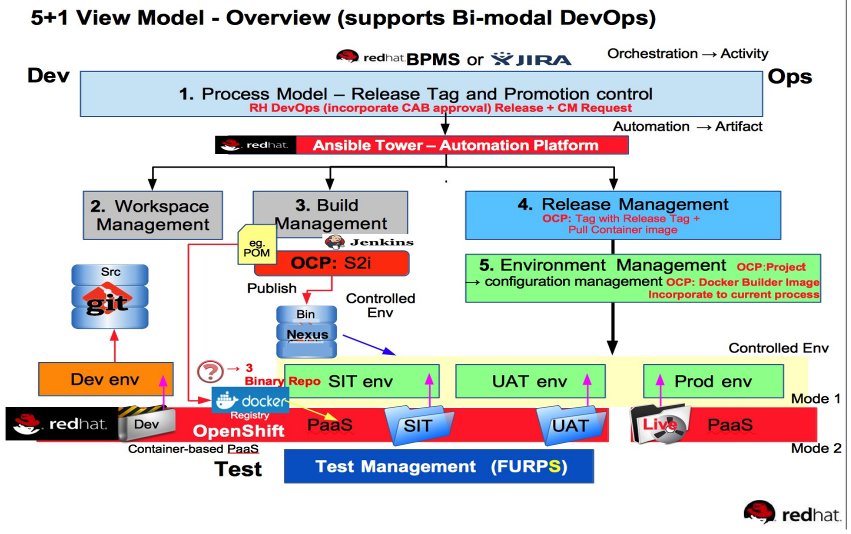
1 Manage Workspace with Git in the 5+1 View Model Approach

This article is focused on providing the design and mechanism to manage the SCM usage model for Git in the 5+1 View Model approach. A new concept of managing parallel development with the use of the DevOps process model will be introduced with the aim to minimize the learning curve of implementing the workspace management within the team and ease the SCM related activities which developers would need work on.

1 5+1 View Synopsis

The following provides a synopsis on the key areas necessary to be addressed as a level 1 entry for a typical DevOps journey. There are topics that are required to be addressed in greater details beyond the level 1 DevOps implementation. For example: 1) Dev Development activities such as Agile Scrum, requirement management, and etc ; and 2) Ops Operation activities such as system/infra monitoring and Continuous Assessment → feedback loop.


5+1 View Model - Overview (supports Bi-modal DevOps)
redhat BPMS or JIRA
Orchestration → Activity
Dev
Ops
1. Process Model – Release Tag and Promotion control
RH DevOps (incorporate CAB approval) Release + CM Request
Automation → Artifact
redhat Ansible Tower – Automation Platform
2. Workspace Management
3. Build Management
Jenkins
eg. POM
OCP: S2i
Publish
Controlled Env
Bin
Nexus
4. Release Management
OCP: Tag with Release Tag + Pull Container image
5. Environment Management OCP:Project
→ configuration management OCP: Docker Builder Image Incorporate to current process
Src
git
Dev env
→ 3 Binary Repo
docker Registry
Controlled Env
SIT env
UAT env
Prod env
Mode 1
redhat
Dev
OpenShift
PaaS
SIT
UAT
Live
PaaS
Mode 2
Container-based PaaS
Test
Test Management (FURPS)
redhat

The following diagram illustrate the 5+1 key areas to be addressed in a DevOps journey and how open source solution helps to address them.

The essence of the 5+1 view model focus on 6 key areas starting with the Process Model. This covers the orchestration of the DevOps overarching activities which ensure a streamline approach in moving the development artifacts across the various environment and finally to the production deployment. The promotion of the artifacts should automated to the fullest while ensuring proper tracking and audit for clients who are in a regulated industry.

The orchestration expands into the next area which is the Workspace Management. In this aspect, the DevOps practice would depend very much on the team's development maturity. Regardless, the vital point is to to be able to address parallel development with a SCM repository put in place. In today's context, many are talking about Continuous Integration (CI) which addresses the build and deployment of development builds into the Dev environment. This is usually achieved with web hooks to trigger build and deploy upon check-in of source code.

2 Workspace Management

The DevOps platform manages the workspace management aspect with a simplified SCM usage model. This model focus on the use of a ***development branch*** *(referred to as* ***branch*** *from this point)* and ***tag branch*** *(referred to as* ***tag*** *from this point)* . There is no concept of a ***trunk***. Instead, the tracking of the production baseline code-base is managed in the existence of a ***tag*** and this information (production baseline) is maintained the DevOps Process Model implemented using the Jira Software.

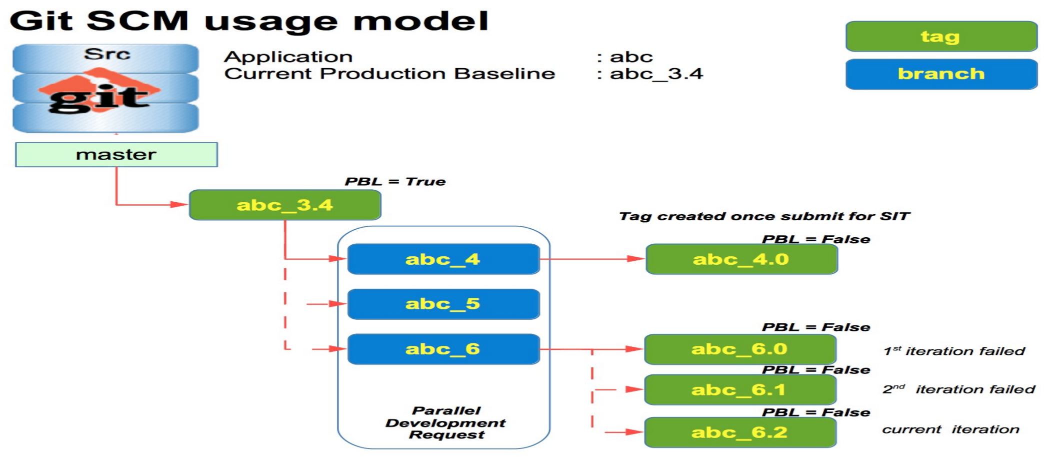
The ***tag*** basically stores all source code that is deployed to SIT/UAT/PRD. The name of the ***tag*** will indicate the release label that is issued by the Process Model and maintained in Jira. This release tag will also be used to tag the build runtime. (application container image in the Registry of OpenShift (OCP) or binary published in Nexus that represents the bin SCM)

The ***branch*** will be used for development purposes. The ***branch*** will be created automatically from the Process Model when the release-issue is approved to proceed to Work-in-Progress state. The ***branch*** is created from the ***tag*** that represents the current production baseline. The ***branch*** will be created using the release tag main prefix in the form of [application name]_[current release number]. When the release-issue is promoted to "Review" state, a ***tag*** representing the release tag in the form of [application name]_[current release number]_[minor number] will be created.

The Git repository is first initialized with the application name on the remote repository.

The next step is to create a **tag** branch from the master branch. This **tag** branch will be locked for readonly using the *custom hooks* to be created on the application Git Repository. The *custom hooks* will read from a file which contains the release tag that are classified as readonly and return an exit -1 if true.

The following diagram depicts the branching outcome as the application goes through multiple parallel development request release.

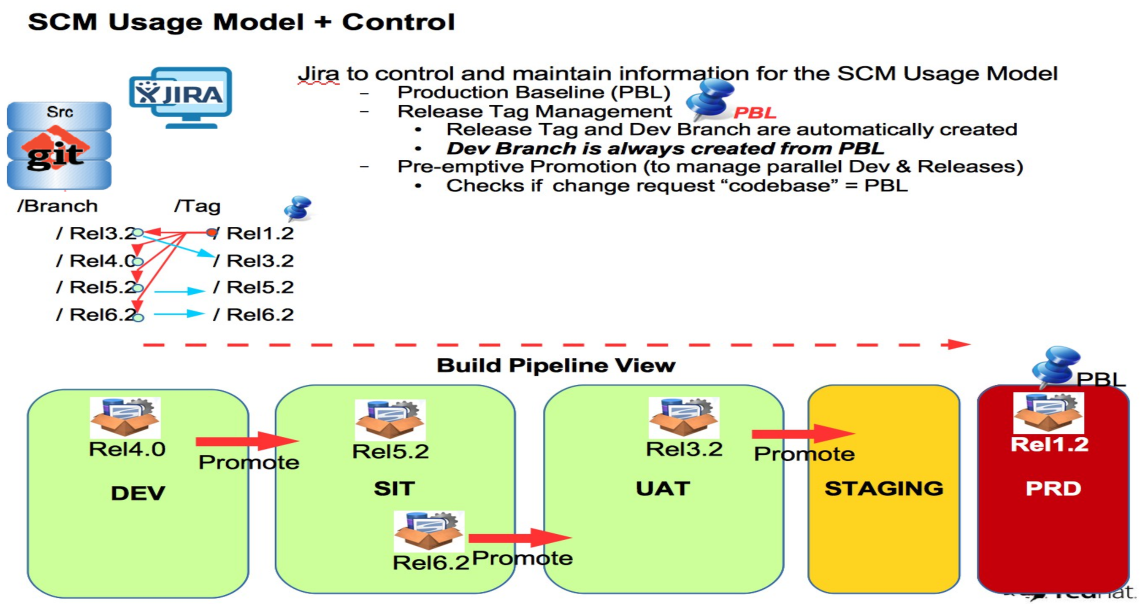


When the release-issue is promoted to PRD then to the Closed state, the Process Model will update the production baseline with this release tag. In this event, the following scenarios will occur:

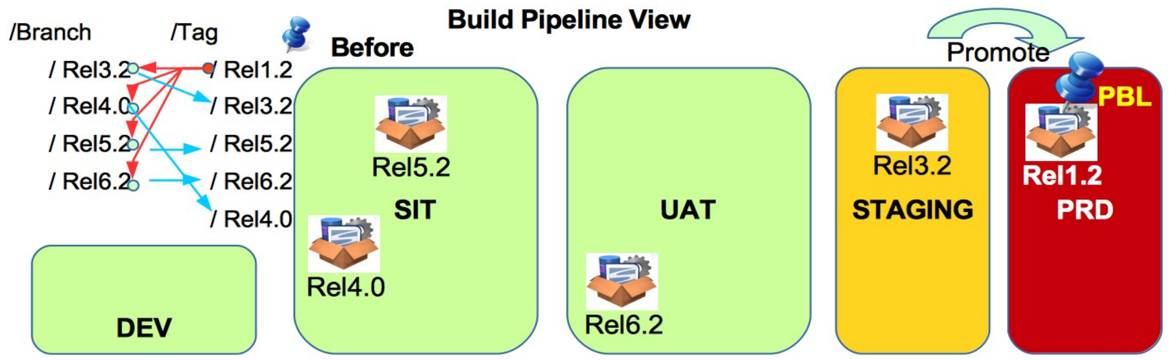
- a new release-issue is raised , the branch will be created using this release tag from the ***tag***

- existing development ***branch*** would need to merge all source code from the ***tag*** labeled by this release tag
    - The Process Model will not allow any Release-issue to be promoted whenever there is a change in the production baseline. Instead, a message will be prompted for the code merge with the respective release tag information. The acknowledgement of this message and promoting the release-issue to the "re-work_codemerge" state is an electronic sign-off to agree to perform the code merge activities.
    - Currently there is no effective way to automate the code merge for the following scenarios:
        1. Non-conflicting changes
            - automatic code merge capability is usually provided by a client utility or some scripts. ***Note***: *the draw back is that some auto merge might result in non-working codes.*
        2. Conflicting changes
            - the developer would need to decide which line of conflict code will be the final one.

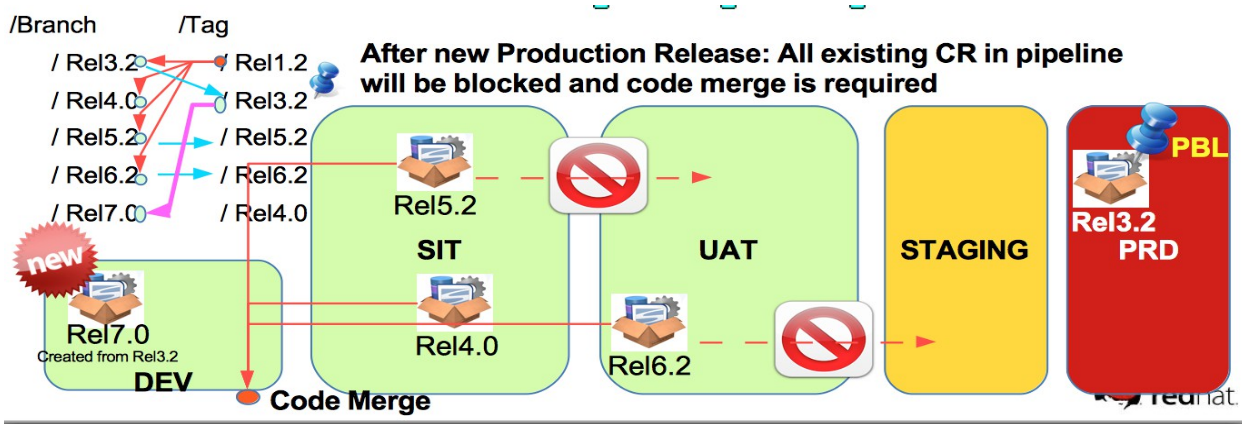
The following section will illustrate the creation of ***branch*** and ***tag.***


SCM Usage Model + Control
Jira to control and maintain information for the SCM Usage Model
– Production Baseline (PBL)
– Release Tag Management
PBL
• Release Tag and Dev Branch are automatically created
• Dev Branch is always created from PBL
– Pre-emptive Promotion (to manage parallel Dev & Releases)
• Checks if change request "codebase" = PBL
Src
git
JIRA
/Branch
/Tag
/ Rel3.2
/ Rel4.0
/ Rel5.2
/ Rel6.2
/ Rel1.2
/ Rel3.2
/ Rel5.2
/ Rel6.2
Build Pipeline View
Rel4.0
DEV
Promote
Rel5.2
SIT
Rel6.2
Promote
Rel3.2
UAT
Promote
STAGING
PBL
Rel1.2
PRD

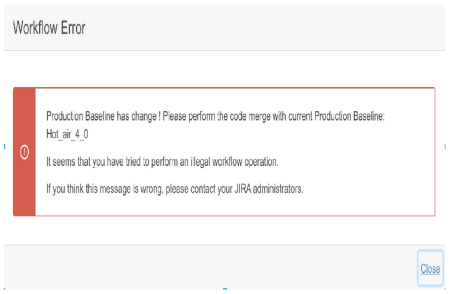
In the above example, Rel1.2 is the Production baseline. This is the tag used to create the concurrent release-issue request raise for Rel3.2, Rel4.0, Rel5.0, and Rel6.0. The above diagram also depicts the stage each of the parallel release request. Do note that Rel3.2, Rel5.0, and Rel6.0 are also created in the …/**tag** since they have been promoted to the "review" state each promotes the request to SIT and beyond. However, Rel4.0 is not found under …/**tags** until it is promoted to the "review" state.



With the promotion of Rel4.0 to SIT, the Rel4.0 release tag is now created in …/**tags** with various release-issue requests promotion. In this example, Rel3.2 has completed the performance test and ready for promotion to the production.



When Rel3.2 is deployed to Production, the Production baseline will now be pinned to Rel3.2(info maintained in the Process Model). Any attempt to promote any release-issue request in the build pipeline will result in a denial with the Process Model prompting the user to proceed to transit to the "re-work_codemerge" state.


Workflow Error

Production Baseline has change ! Please perform the code merge with current Production Baseline: Hot_air_4_0

It seems that you have tried to perform an ilegal workflow operation.

If you think this message is wrong, please contact your JIRA administrators.

At this point of time, should any new Release-issue request is approved for development (transit to "work-in-progress" state), the …/**branches** to be created will be from the new production baseline …/**tags**/Rel3.2.

3 Design Mechanism

The following are the commands in Git which are deployed to support the simplified SCM usage
model :

1. **Script to create branch :** git_create_branch.sh
2. **Usage:** git_create_branch.sh p1 p2 p3 p4 p5 p6

```
#!/bin/bash
#
# passing arguments
#
echo "Workpath: $1"
echo "Git Repo: $2"
echo "PBL/src Br : $3"
echo "Release tag: $4"
echo "Tgt Branch : $5"
echo "Lock Tgt Br: $6"
#
# #######################
#
git_repo=$(echo $2| cut -d'/' -f 4 | cut -d'.' -f 1)
echo "Repo name: $git_repo"
pwd
#
# Create working folder [workpath]/[release tag]
#
cd $1
mkdir $4
cd $4
#
# Create git local repo
#
git clone $2 -b $3
pwd
cd $git_repo
pwd
#
# Create a new branch "new_branch" with the same state as "old_branch"
# cmd: git checkout -b new_branch old_branch
#
git branch $5
git checkout $5
touch $(echo $4.tag)
git add .
git commit -m $4
git ls-remote --heads origin
git push -u origin $5
#
# lock branch
#
if [ $6 == "lock" ]
then
  echo "good lock !!!!!"
  echo "Update pre-receive hooks list file with : $5"
#
# ansible remote file update
#
fi
#
```

To achieve the ***tag*** **locking** mechanism for **GitLab on-premise** system, we leverage **Ansible Tower** to manipulate *custom hooks* feature provided by GitLab.

The idea is to create a custom ***update*** *hook file* and a text file containing all the tag names we want to lock, under the *custom_hooks* folder inside the GitLab repository file system. Please note that this is not inside the *repository* itself, rather it is in the GitLab server filesystems. For this case, we are working with default GitLab repository path in /var/opt/gitlab/git-data/repositories/<<user/groupname>>/<<repo_name>>.git.

We are creating the *update* hook file by copying over a ready-to-use *update* hook file from Ansible Role, if the file is not there yet. And then we put in the tag branch name to lock to the custom lock file named locked_branch_list_file.txt. The custom *update* hook file will look up to this text file for branch names that must not be edited.

Ansible Role default variable file:

```
---
# defaults file for gitlab_lock
repository_url: "git@10.11.12.13:ansible/dummy.git"
gitlab_host: "{{ (repository_url.split('@') | last).split(':') | first }}"
gitlab_user: "{{ ((repository_url.split('@') | last).split(':') | last).split('/') |
first }}"
gitlab_repository_folder_name: "{{ ((repository_url.split('@') | last).split(':') |
last).split('/') | last }}"
gitlab_branchname_to_lock: dummy_1_0
gitlab_repository_path: "/var/opt/gitlab/git-data/repositories/{{ gitlab_user }}"
custom_hooks_dir: "{{ gitlab_repository_path }}/
{{ gitlab_repository_folder_name }}/custom_hooks"
hooks_file: update
locked_branch_list_file: locked_branch_list_file.txt
locked_branch_list_prepend: refs/heads/
```

Ansible Role task file:

```
---
- name: "Create (if not exist) {{ custom_hooks_dir }} folder"
  file:
    path: "{{ custom_hooks_dir }}"
    mode: 0755
    owner: root
    group: root
    state: directory
```

```
- name: "Create (if not exist) {{ hooks_file }} script file"
  copy:
    dest: "{{ custom_hooks_dir }}/{{ hooks_file }}"
    src: "{{ hooks_file }}"
    mode: 0755 # must be executable
    owner: root
    group: root

- name: "Put {{ locked_branch_list_file }} to {{ locked_branch_list_file }} under
{{ custom_hooks_dir }} folder"
  lineinfile:
    path: "{{ custom_hooks_dir }}/{{ locked_branch_list_file }}"
    line: "{{ locked_branch_list_prepend }}{{ gitlab_branchname_to_lock }}"
    state: present
    create: yes

- name: "Make sure {{ locked_branch_list_file }} is read-only"
  file:
    path: "{{ custom_hooks_dir }}/{{ locked_branch_list_file }}"
    mode: 0400
    owner: git
    group: git
```

Ansible Role template for *update* hook file:

```
#!/bin/sh
#
# A hook script to block unannotated tags from entering.
# Called by "git receive-pack" with arguments: refname sha1-old sha1-new

# --- Command line
refname="$1"
oldrev="$2"
newrev="$3"

# --- Safety check
if [ -z "$GIT_DIR" ]; then
echo "Don't run this script from the command line." >&2
echo " (if you want, you could supply GIT_DIR then run" >&2
echo "  $0 <ref> <oldrev> <newrev>)" >&2
exit 1
fi

if [ -z "$refname" -o -z "$oldrev" -o -z "$newrev" ]; then
```

```
echo "usage: $0 <ref> <oldrev> <newrev>" >&2
exit 1
fi

# --- Check types
# if $newrev is 0000...0000, it's a commit to delete a ref.
zero="0000000000000000000000000000000000000000"
if [ "$newrev" = "$zero" ]; then
newrev_type=delete
else
newrev_type=$(git cat-file -t $newrev)
fi

# refname sample: refs/heads/dummy_1_0, refs/heads/locked_branch
if grep "^${refname}$" ./locked_branch_list_file.txt ; then
  echo "*** THIS BRANCH IS LOCKED AS A RELEASE BRANCH!" >&2
  exit 1
fi

# --- Finished
exit 0
```

Jira post-function script to call Ansible Role:

```
package rmf_scripts.release

import com.atlassian.jira.issue.ModifiedValue
import com.atlassian.jira.issue.util.DefaultIssueChangeHolder
import com.atlassian.jira.component.ComponentAccessor
import rmf_scripts.util.AnsibleTowerLauncher
import static rmf_scripts.CustomFields.*

log.setLevel( org.apache.log4j.Level.DEBUG )
def cfm = ComponentAccessor.getCustomFieldManager()
def issueManager = ComponentAccessor.getIssueManager()
def app = issue.getCustomFieldValue(cfm.getCustomFieldObject(v_CR_App_name)) as String
def applicationIssue = issueManager.getIssueObject(app)
def app_name =
applicationIssue.getCustomFieldValue(cfm.getCustomFieldObject(v_amt_App_name)) as String
def p_SCM_src_repo =
applicationIssue.getCustomFieldValue(cfm.getCustomFieldObject(v_amt_scm_src_branch)) as
String
def CR_release_branch =
issue.getCustomFieldValue(cfm.getCustomFieldObject(v_CR_release_name_main)) as String
```

```
def CR_release_tag = issue.getCustomFieldValue(cfm.getCustomFieldObject(v_CR_release_tag))
as String

// set the CR_otag to the current tag
def textCf22 = cfm.getCustomFieldObject( v_CR_otag )
if (textCf22) {
    textCf22.updateValue(null, issue, new ModifiedValue(issue.getCustomFieldValue(textCf22),
CR_release_tag), new DefaultIssueChangeHolder())
}

log.debug "app_name: ${app_name}"
log.debug "p_SCM_src_repo: ${p_SCM_src_repo}"
log.debug "CR_release_branch: ${CR_release_branch}"
log.debug "CR_release_tag: ${CR_release_tag}"

log.debug "Call Ansible - Create Release_Tag Branch"
def extraVars = '{"extra_vars": {' + '\"repository_url\": \"' + p_SCM_src_repo +
                '\", \"from_branch\": \"' +  CR_release_branch  +
                '\", \"target_branch\": \"' +  CR_release_tag  + '\" } }'
log.debug "Extra Vars: ${extraVars}"
def gitStatus = AnsibleTowerLauncher.callTower(AnsibleTowerLauncher.CREATE_BRANCH_ID,
extraVars)
log.debug "Git release_tag branch creation status: ${gitStatus}"

// Lock down the Release_Tag branch
log.debug "Call Ansible - LOCK Release_Tag Branch"
extraVars = '{"extra_vars": {' + '\"repository_url\": \"' + p_SCM_src_repo +
                '\", \"gitlab_branchname_to_lock\": \"' +  CR_release_tag  + '\" } }'
log.debug "Extra Vars: ${extraVars}"
gitStatus = AnsibleTowerLauncher.callTower(AnsibleTowerLauncher.LOCK_GITLAB_BRANCH_ID,
extraVars)
log.debug "Git release_tag branch LOCKING status: ${gitStatus}"
```

4 Summary

In conclusion, a simplified SCM usage model defined in the 5+1 DevOps process model is designed to simplify the workspace management activities for the development team. Thus, getting the developers to focus on design and coding. Multiple SCM related tasks can now be avoided except for 1) checkout branch to local workspace, and 2) merge from tag/branch , all else would be managed by the process model (branching and tracking of production baseline tag)