Lenovo

Reference Architecture:

Git Strategy

Version 1.0

Feng Xia

March 23, 2020

DRAFT COPY

Abstract

Which git workflow to use in an enterprise environment is a question every development has to answer. Yet in practice there are many confusions that may result in surprising pitfalls unintended and even unnoticed by the team. In this article we identify 16 different strategys, and how they affect the content of a release. We will recommend one strategy as it gives control over the release contents.

Contents

List of Tables

List of Figures

Git is wonderful, but its workflow is very confusing. Surprised? You would think it sounds rather cliche that this problem even exists. But take a step back and consider this scenario:

1. We just did a release 5.0.0.
2. We have two features: 1 & 2.
3. Initially they were all targeting the next release, say 5.1.
4. We create a **staging** branch, 5.1-staging and point our CICD to this so to QA them (more on this later).
5. Feature 1 & 2 have been merged into 5.1-staging. Their source branches, eg. feature-1, have been deleted at merge request, select to delete source branch.
6. Time is slipping, 5.1 is up this Friday, but one feature is failing QA tests. So management is going to release 5.1 w/ feature 1 only, and moving feature 2 to release 5.2, which has another staging branch 5.2-staging.

All sound fair and common. But there are multiple questions. First, let's set the objective.

1 Objective

> AS a product owner, I want to control the release content, so that I can plan release schedule into the future and communicate w/ stake holders of availability of a feature based on these schedules.

No mystery here. Regardless Agile or not, team is to deliver features, and these features will fall into a release, and release is not Agile by nature however Agile desires the product to be *usable* by the end of each Sprint, the reality is not so, and further, the definition of a release has usually two more requirements than a product being *usable* (usable is an ambiguous term and is useless in determining a release):

1. It must pass a set of user defined tests (UAT).
2. We must know the content of the release in term of functions (aka. features).

It's the second requirement we are to discuss in this article.

2 Problem statement

The problem is that at any given time, there are multiple teams and developers working on various topics — some features, some bug fixings, thus code is constantly in fluctuation. This is directly contradicting us controlling the content of a release:

> I want to select the changes I want and nothing else, while the pool of changes to be selected from is constantly expanding in a rate higher than an individual can keep up w/.

Since all development result in a code change, and all code changes are ear marked in git, is there a git strategy to solve this problem?

Yes there is gotta be. Otherwise, git or any other version control tool will become useless for software development.

3 Two rules

In git, individual code change is ear marked in **commit**, and changes are *grouped* into **branches**. Here we are to use two types of branch for discussion — feature branch, and release staging. Feature branch is to hold the group of changes of a feature; release staging is hold the group of changes of a release. It is the release staging we are interested in controlling.

Assuming branch develop is our MASTER:

1. Feature branch **must go into a staging** or abandoned. No one can merge a feature directly into develop this is to prevent skipping CICD/QA.
2. **Release staging will go into develop eventually instead of living as independent branch indefinitely**. This is assuming that all releases are essentially backwards compatible. **This may not be true**, and we will discuss further in "Long term support" section.

4 Possible strategies

Now consider the original story that we have a past release of 5.0, and two upcoming releases 5.1 & 5.2. Since a release must have staging, we will have two staging branches: 5.1-staging and 5.2-staging.

There are two features in development, feature 1 & 2, correspondingly there are two feature branches.

Out of all these, there are only finite combinations of possibilities:

1. What are the origin of these two staging branches? Two possibilities:
    1. both from develop
    2. 5.2-staging is a branch off 5.1-staging, and 5.1-staging was branched off develop.
2. What are the origin of these two feature branches?
    1. both from develop
    2. both from 5.1-staging

Similarly we determine where these features are merged into, and where these staging branches are merged in eventually. So with these in mind, we have the following combinations:

Table 1: Possible git strategies by feature branch and release staging branch

Strategy	Staging from	Staging to	Feature from	Feature to	Applicable
1	develop	develop	develop	staging	
2			develop	develop	no
3			staging 1	staging 1	
4			staging 1	staging 2	
5	another staging	develop	develop	staging	
6			develop	develop	no
7			staging 1	staging 1	
8			staging 1	staging 2	
9	develop	staging	develop	staging	
10			develop	develop	no
11			staging 1	staging 1	
12			staging 1	staging 2	
13	another staging	yet another staging	develop	staging	
14			develop	develop	no
15			staging 1	staging 1	
16			staging 1	staging 2	

Strategy 2,6,10,14 are ruled out because they violate rule #1 □ feature code can not be merged directly to develop.

5 Strategy 1

Strategy	Staging from	Staging to	Feature from	Feature to
1	develop	develop	develop	staging

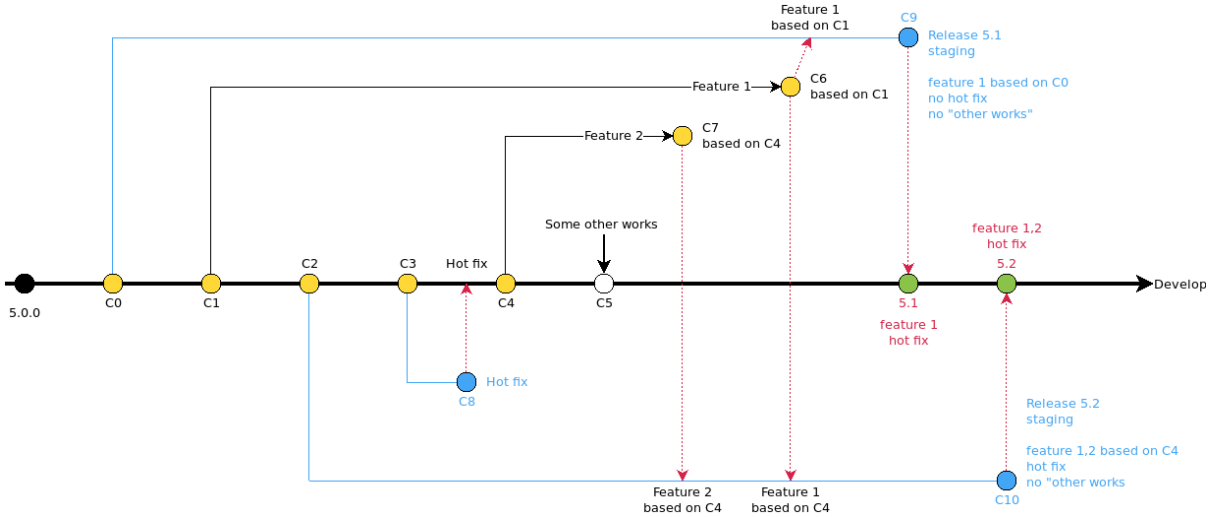

Figure 1: Release staging (dev□dev), feature (dev□staging)

In the diagram I have two feature branches and two staging branches. All four were created from develop.

Pros:

- ☞ Developer always create his/her feature from develop. Thus they don't have to be aware which release this feature is going into. Logistically it's a managerial decision anyway.

```
1 But, this is to creating a big problem! See first item in "cons
    ↪ "
2 section below.
```

Cons:

- ☞ Developer's starting point is arbitrary □ dev don't have control what is in C0 and C4, nor do they really pay attention to them when they were creating a branch at that moment.

What this means is that feature 1 is fine in its own branch, but may break as soon as it's merged to 5.2-staging, because C4 is after both C0 and C1, thus having **unknown** contents when developer was working on feature 1.

Mitigation is to control the starting points, for example, having both features starting from C0 if we have multiple features, the latter ones will have to consult the very first feature branch for its starting point.

☞ staging-5.2 will also have hot fix content because of feature 2 merge. This may not be desired if I want feature branch to only have feature content.

Because features are created from develop, C4 from which feature 2 branch was created is essentially a random location (you don't necessarily know what is before C4 when you decided to create a 5.2-staging brach from it), thus making the content of 5.2-staging random as well. Same is true for C0 that we don't necessarily consider it be part of 5.1 release at all, but it is pulled in because of the branching strategy.

Therefore, contents of both feature staging become arbitrary if not controlled carefully when the feature branch is created. Considering that dev owns feature branch, it's too much a risk to rely on personal code of conduct to keep staging branch *clean/minimal*.

6 Strategy 3&4

Strategy	Staging from	Staging to	Feature from	Feature to
3	develop	develop	staging 1	staging 1
4	develop	develop	staging 1	staging 2

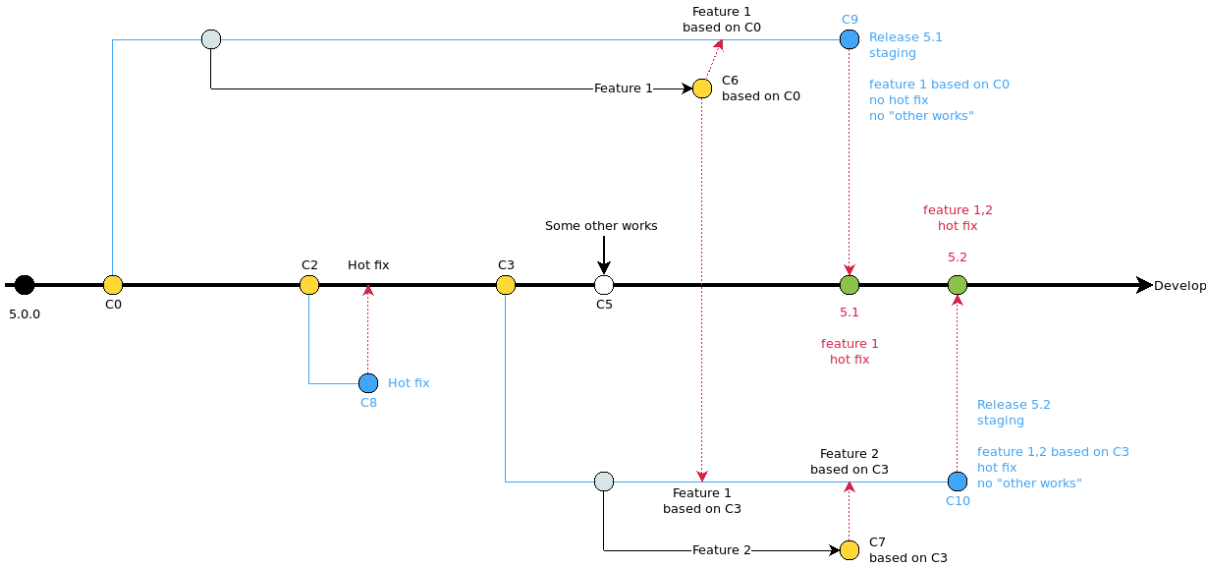

Figure 2: Release staging (dev□dev), feature (staging□staging)

In this diagram I am covering two strategies:

1. feature is either folding into the same stating it was branched off (case 3) — feature 2 was developed for 5.2, and will be merged to 5.2-staging.
2. feature is merged onto yet another staging (case 4) — feature 1 was developed for 5.1, but later was decided to be 5.2.

Cons:

- ☞ As in strategy 1, staging branches were created from two different develop commits, making its contents arbitrary.
- ☞ In case 4, feature 1 may stop working when merged into 5.2-staging because contents between C0 and C3 were unknown to feature 1's developer.
- ☞ Developer has to be aware which release the feature is intended so to create the feature branch off the **proper** staging, even though this doesn't have any material impact on the code state. Thus this is an unnecessary noise for developer and adds no value but a falsified impression that we are feature code is already *contained* within a release, which it is not.

7 Strategy 5

Strategy	Staging from	Staging to	Feature from	Feature to
5	another staging	develop	develop	staging

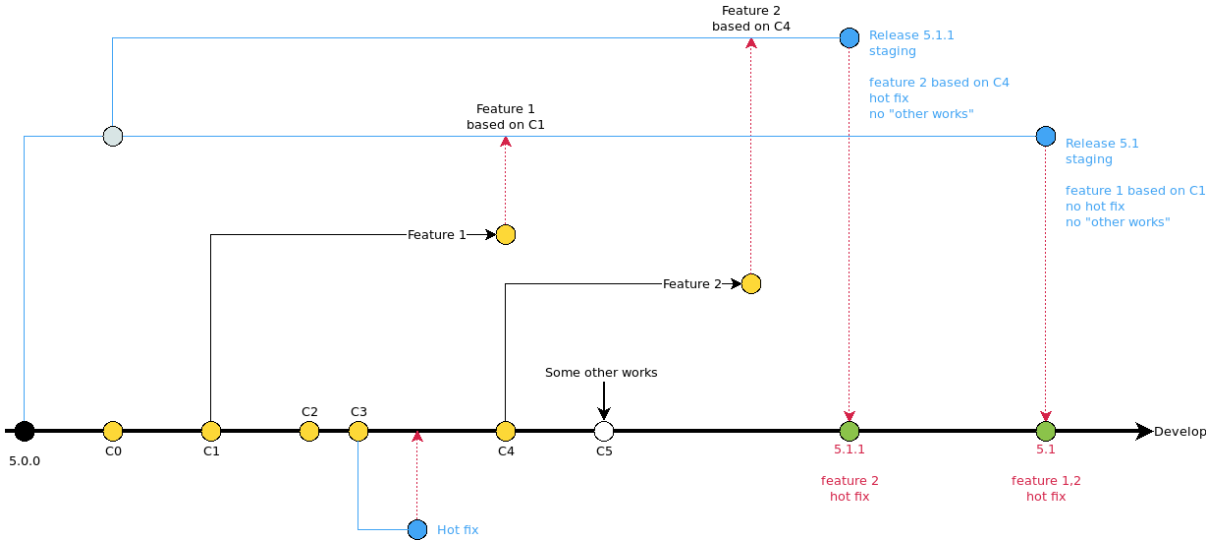

Figure 3: Release staging (staging→dev), feature (dev→staging)

In this strategy, 5.1.1-staging is created from 5.1-staging. This is simulating that we started w/ 5.1 release development, but later decided to do a 5.1.1 release first.

Cons:

1. This strategy suffers the same problems as strategy 1, that we will have arbitrary contents in the staging depending on what the feature is bringing w/ it — feature 2 will drag in all the changes of C0 to C4 into 5.1.1-staging.

8 Strategy 7

Strategy	Staging from	Staging to	Feature from	Feature to
7	another staging	develop	staging 1	staging 1

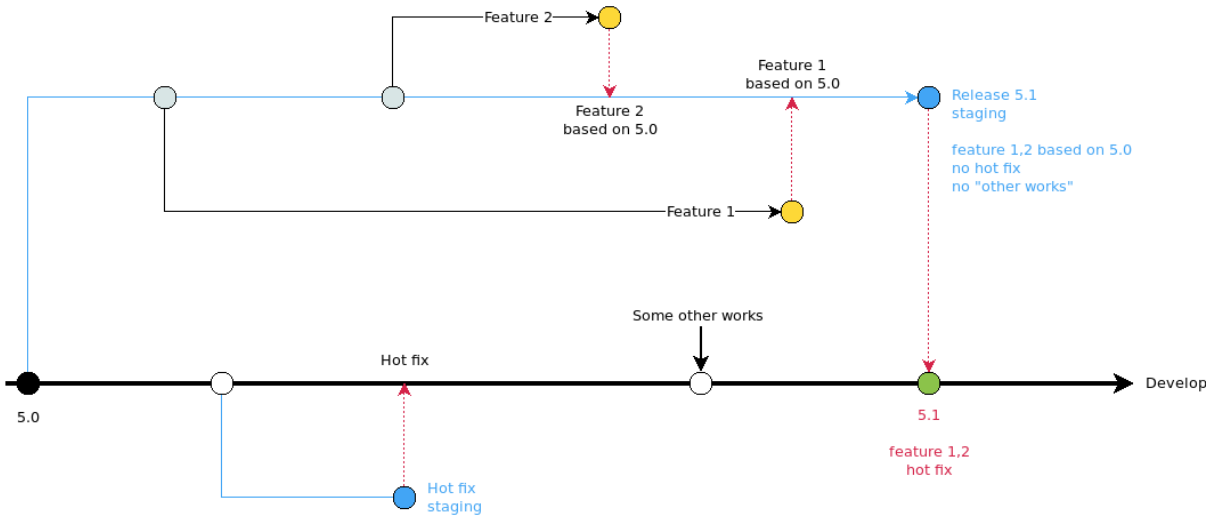

Figure 4: Release staging (staging□dev), feature (staging□same staging)

Pros:

- ☞ This is a clean solution that we have had a 5.0 release, thus any staging is off the 5.0 commit, creating a common base for all features.
- ☞ Features are confined to its release staging during development.
- ☞ Contents of the staging is controlled. Nothing on develop will leak into the staging by surprise.

9 Strategy 8

Strategy	Staging from	Staging to	Feature from	Feature to
8	another staging	develop	staging 1	staging 2

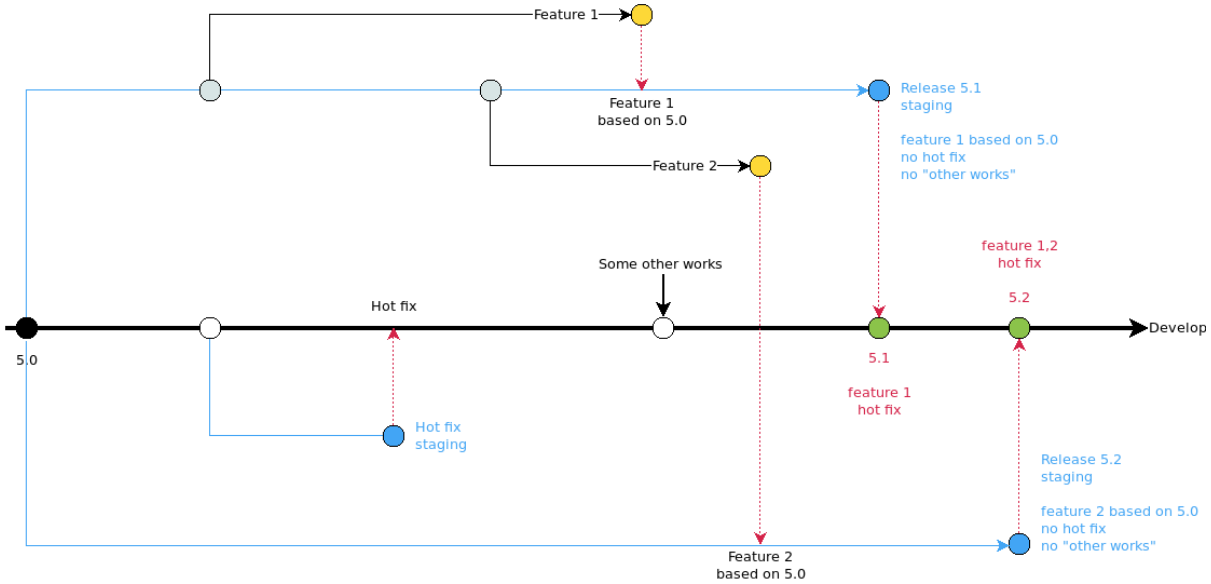

Figure 5: Release staging (staging→dev), feature (staging 1→staging 2)

Similar to strategy 4, that we are changing mind during a release (5.1-staging), that we are shift one feature to another release (5.2-staging) instead.

Pros:

- ☞ Because of the common root law of two staging branches, shifting feature has no ill effect at all — both staging has clean contents from feature development, and feature 2 will work the same on 5.2-staging because it is based on 5.0 in any case.

10 Strategy 9

Strategy	Staging from	Staging to	Feature from	Feature to
9	develop	staging	develop	staging

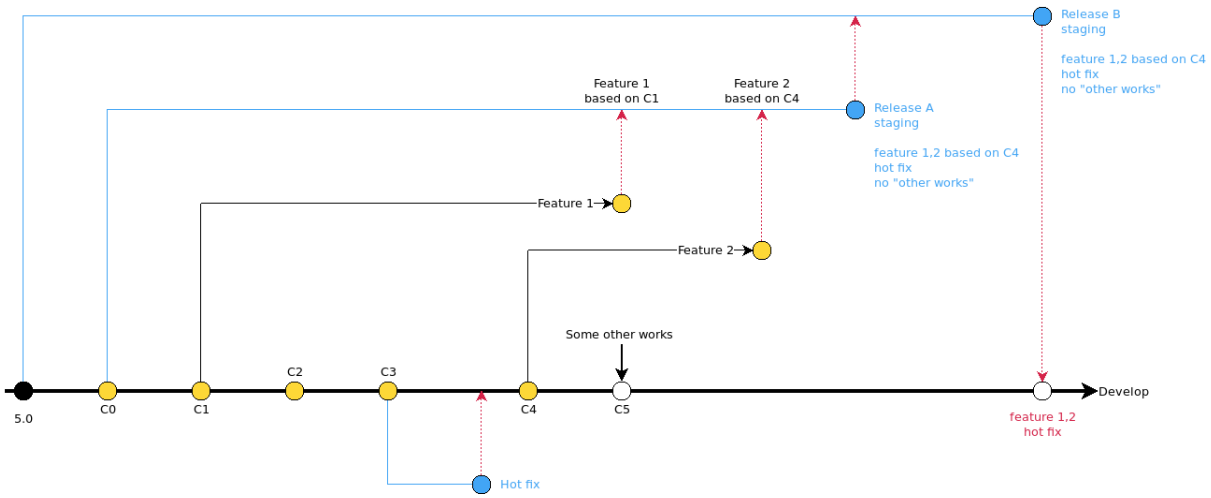

Figure 6: Release staging (dev□staging), feature (dev□staging)

Up to this point you can certainly see the similarity between strategy 1, 5 and 9. Even though we are merging staging into another staging instead of develop, this strategy suffers all the problems strategy 1&5 suffer. After all, the core of the objective is to have a known content in a release, which this fails to achieve.

11 Strategy conclusion

I'm gonna skip strategy 10-16 because they are redundant after the discussions so far. It's clear that the key to the issue are two:

1. root of the staging
2. root of the feature

If we can't control these two, we lost control of the contents in each branch. Therefore, the only viable strategy is 7 or 8:

1. stagings use the same root, ideally a past release
2. root of feature is staging (if we satisfies first point, all stagings are essentially the same!)

12 Move feature to new staging

Common request is to move a feature to a new release. The feature may be in progress or having been merged. How to do this properly?

From strategy 8, we already see that repointing MR is necessarily sufficient because moving feature 2 will bring commits that were developed on 5.1-staging. For all we know, these commits can be intermittent works by feature 1. Therefore, we will be leaking feature 1 works to the 5.2-staging.

The clean method is to fully revert feature 2 on 5.1-staging, **reimplement** it from scratch on 5.2-staging, then MR.

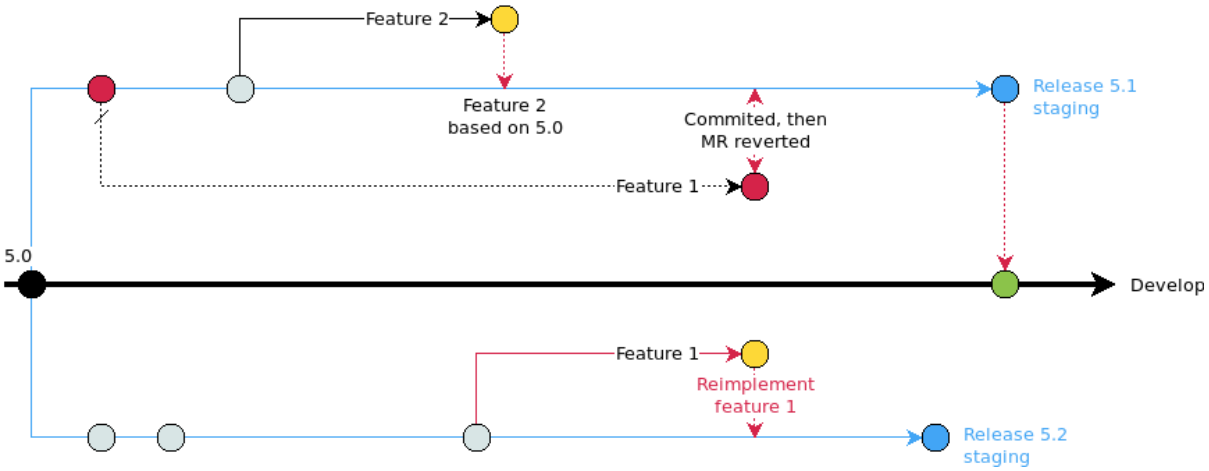

Figure 7: Move feature to a new release staging

13 Long term support (LTS) releases

A release is a snapshot in time. But that hardly holds in real life because there will be hot fixes — things you have to patch, say, security, that will ultimately change the content of a released release. What is the strategy for hot fixes?

13.1 Hot fixes

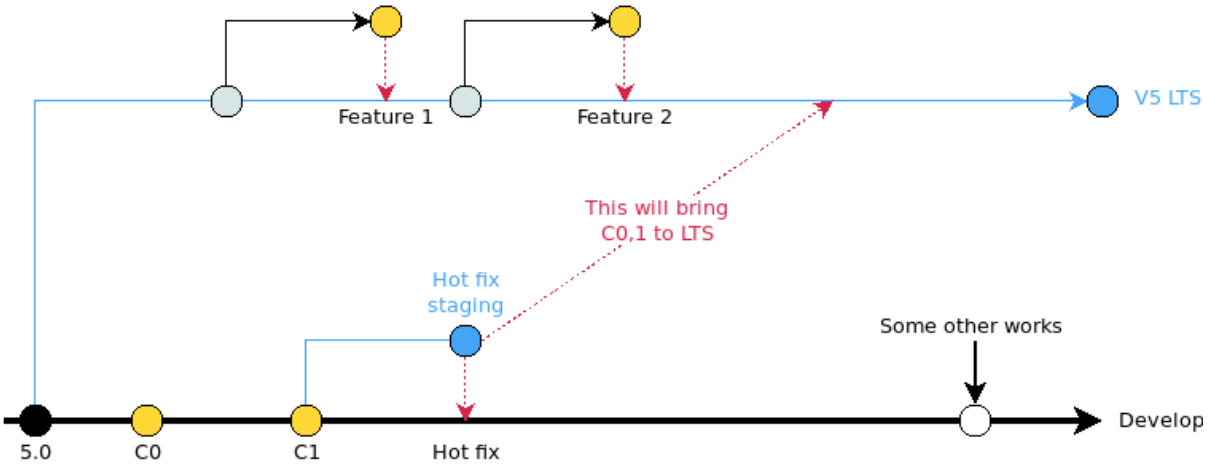

Figure 8: LTS and hot fixes based on Strategy 7

Coming from strategy 7, you can see that same problem will occur again if we are to merge hot fixes into my LTS, that unintended commits C0 and C1 will be included in LTS. Remedy is simple, hot fix branch must be rooted in 5.0 like all 5.x release branches!

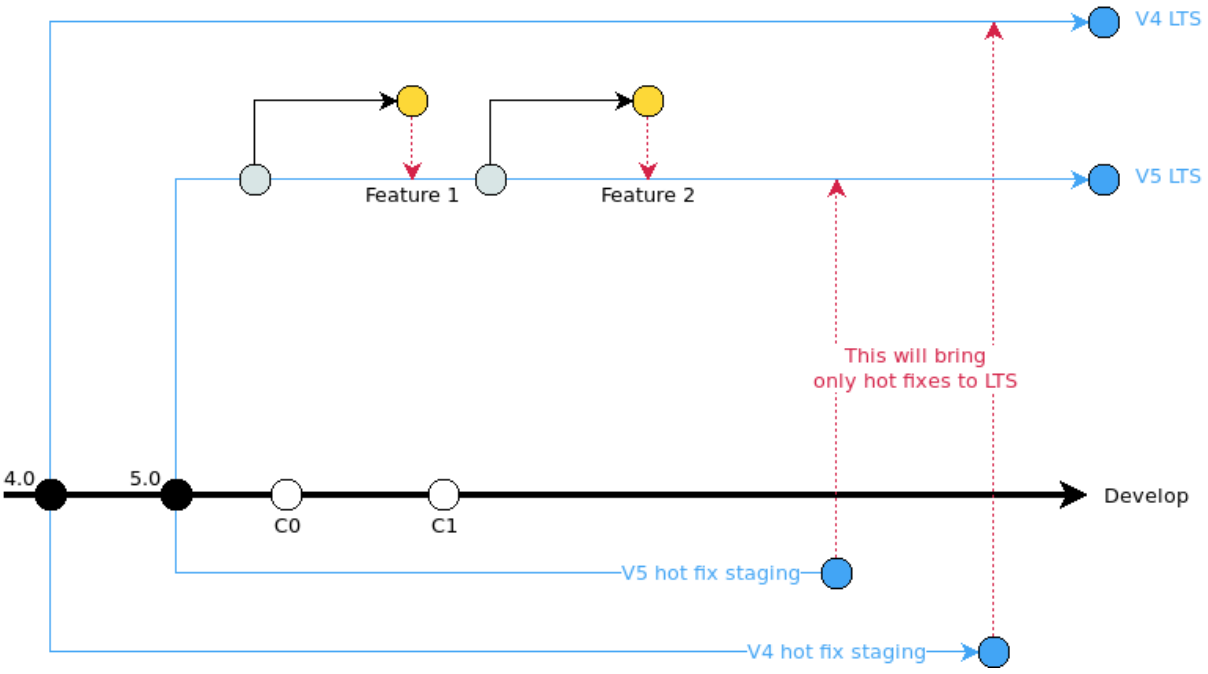

Figure 9: LTS and hot fixes

If we had two releases, 4 & 5, we will create two parallel hot fix branches tracking each.

Now, what if I found a security bug in V4, fixed it, can I apply the same fix to V5? or vice versa? From V4→5, yes; but from 5→4, no. See next section on “Bugs”.

13.2 Bugs

Bugs are essentially the same as hot fixes but w/ a critical difference: we don’t know bugs are critical until we discover and fix them; we know hot fixes are critical because they are usually determined by some external sources such as vendor of a library we are using, or of the operating system we are deploying into.

Once bug is found and fixed, we are facing the question whether this same bug is **critical enough** that needs to be applied in other releases.

Following the hot fix discussion, bugs found in V4 can be applied to V5 without ill feeling because V4’s `C0` is already part of V5. But bringing V5’s bug fix into V4 will then bring `C1` into V4! Therefore, bugs and hot fixes can only roll forward, but never backward — you can apply a bug fix of a older release to a newer release, but not the other way around.

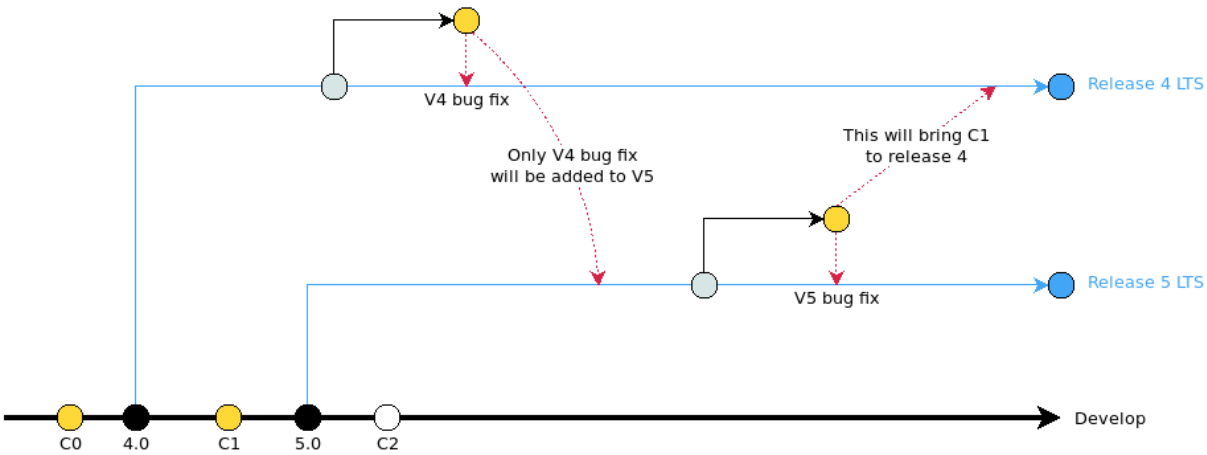

Figure 10: LTS and bug fixes

13.3 CICD/QA, UAT, and life cycle of a release

When does a release start, when does it end? We have said in rule that all development will always go into a staging, and the purpose of a staging is to have a chance for CICD/QA. If we follow git strategy 7, we will have a clear idea of the contents in a staging, thus whenever CICD passes a certain criteria, that particular commit can then be a release candidate. Usually the criteria is per user-acceptance-test (UAT), thus it represents **what the user considers done**.

At that point, the staging can be further *elevated* into a official release. Again, release is mostly dictated by time line or a managerial decision. Dev (including QA) is responsible for passing the UAT on staging, but they don't roll out a release whenever. When we release V5, we fold the code into develop and tag it.

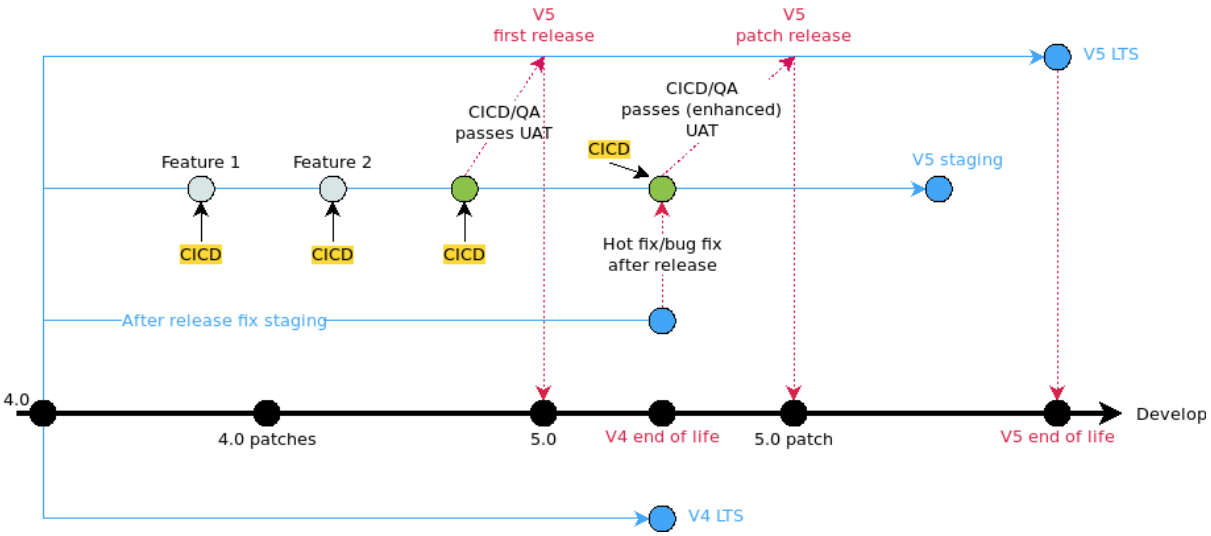

Figure 11: CICD/QA, UAT, and life cycle of LTS

If we released V5 and now have a hot fix, what do we do? We roll hot fix into v5-staging to test, then to a patched release if UAT passes. Note that UAT in this case may be an enhanced version of the original UAT because of the hot fixes. Patch release should also be foled to develop as an official release.

When LTS reaches its end of life, the v5-lts branch is folded into develop. At that point, v5-staging will be removed.

14 Conclusion

To sum these all up:

1. All three types of in-progress branches should root in the same commit, ideally a releas commit — feature, hot fixes, and release staging.
2. If you create multiple release stagings, have them use the same root, too.
3. CICD/QA monitors staging branch.
4. Merge a release commit to `develop` as soon as the release is official.
5. `develop` should not have commits other than releases.

Enjoy.

15 References