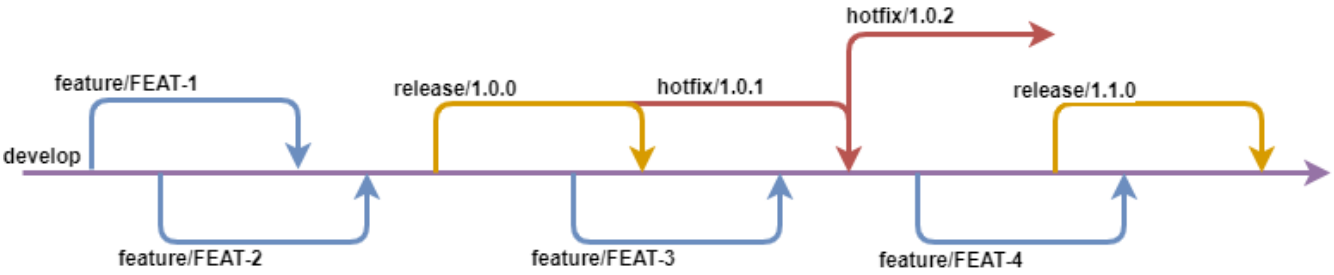
SwatFlow Workflow

Maintained in https://www.notion.so/build-one/Git-Workflow-be5694f897934a38b558d3b36c574a7e

Overview

SwatFlow Workflow is a Git workflow design that was published as an extension to the OneFlow branching model, an alternative to Gitflow.
SwatFlow's main focus is to make the development and deployments processes as flexible as possible.
In this idea, SwatFlow allows developers to work both on active development and hotfixes at the same time without interfering with each other.
The only limitation becomes the number of developers working on the project!

SwatFlow is just a blueprint of how the branches and tags should be maintained and handled during the development and deployment process.
It uses the traditional feature, release and hotfix branches, with slightly modified behavior, but also introduces new branch types for added functionality, such as long-term support. One major change is the role of the 'master' branch, which has been reduced to a symbolic branch.
The purposes of the branches and tags will be described in the following sections.



Develop and Master Branches

In SwatFlow, develop is the core of everything, most feature branches and all release branches will branch out of it.
By default, develop will contain the history of all previous releases and will be the base of any future ones, but it doesn't contain the history of the whole repository as SwatFlow allows the update of any older release, no matter how old.
While develop takes a central role in this workflow, master becomes less important and an optional branch by having the sole purpose of pointing to the latest stable release.
This might be desired as any new clone of a git repository will, by default, check out the master branch.
Both develop and, if present, master, are indefinite branches which will exist forever in the repository.

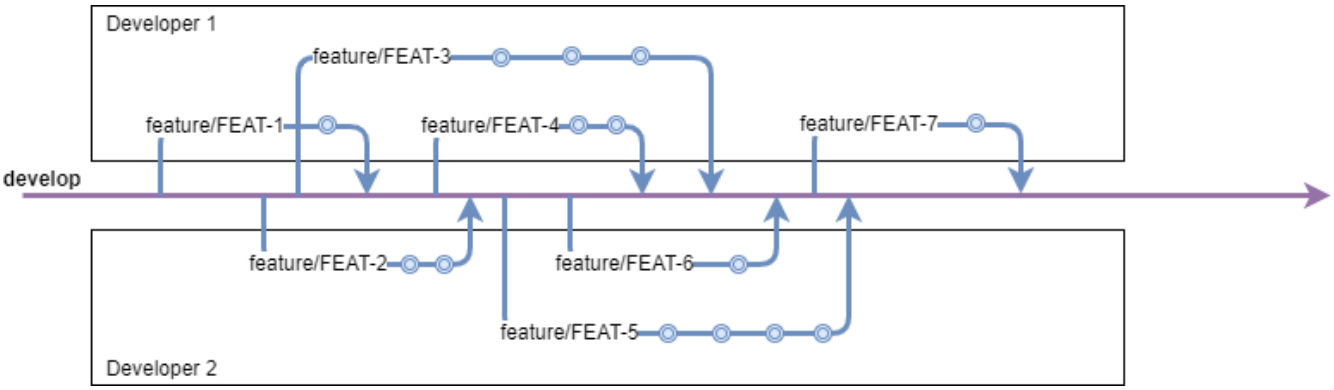
How to fast-forward master branch

git fetch
git checkout master
git pull
git merge --ff-only <RELEASE-TAG>
git push origin master

```
------------
<RELEASE-TAG> -
feature branch that will
be created (x.y.z)
```

Feature Branches

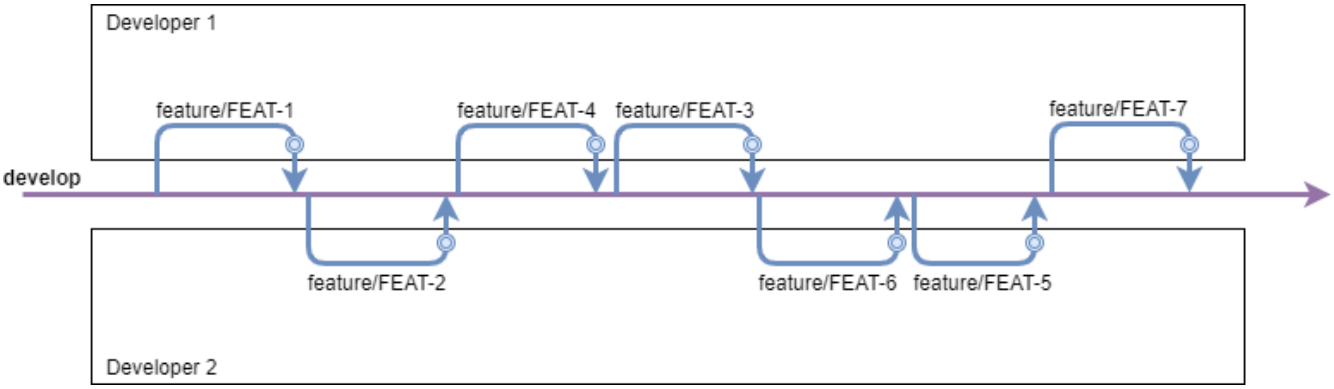
Similar to GitFlow and OneFlow, SwatFlow follows the Feature Branch Workflow, which entails doing any active development (new features, improvements, bugfixes, ...) on separate, independent 'feature/' branches.
Feature branches are created from develop and allow developers to work on their tasks, independent of one another.
Once the feature has been finished and has been approved by QA, a Pull-Request should be done to merge it into develop.
After the merge, the feature branch should be deleted as the branch is relevant during the development process, it is a short-lived branch.



To improve the model even further, the following actions are recommended:

- Before doing a Pull-Request, developers should rebase onto develop to ensure no merge conflicts appear during the PR process.
  This avoids pilling up merge issues when PR are accepted, providing a more swift development.
  This is a requirements due to the way developers are able to work independent of one another, which means that developers can make changes to the same files without being aware of this.
- Before doing a Pull-Request, once the developer has decided that the ticket has been finished, the commits on the feature branch should be squashed into a single one.
  The advantage of this, is that, once merged, a feature branch is represented as a single commit on develop.
  This provides a cleaner history and easier debugging.

With these improvements, the diagram above will look like this:



It can be seen that now, by rebasing and squashing, the history is considerably cleaner.

How to open a feature

```
git fetch --tags --force
git checkout -b <FEATURE>
<SOURCE>

git push origin <FEATURE>

------------
<FEATURE> -
feature branch that will
be created
                (feature
/...)

<SOURCE>  -        source
branch/tag of feature
branch
                (origin
/develop, origin/lts/x,
                origin/lts
/x.y, origin/release/x.y.
z,
                x.y)
```

How to close a feature

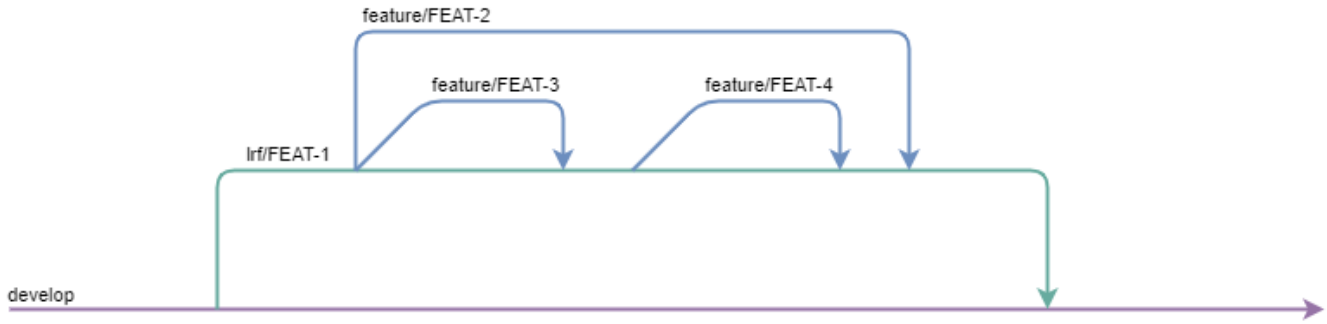
```
git fetch
git checkout <TARGET>

git pull
git merge --squash
<FEATURE>
git push origin <TARGET>

------------
<FEATURE> -
feature branch that will
be closed
                    (origin
/feature/...)
<TARGET>  -        source
of feature branch
(develop,
                     lts/x,
lts/x.y, release/x.y.z,
hotfix/x.y.z)
```

LRF Long Running Feature Branches

It is recommended that any development to be broken down into smaller steps so that a team can take full advantage of the parallel development capabilities of SwatFlow.
There are exceptional cases, where this might not be possible and intermediary stages between the smaller steps can break existing functionalities.
To be able to facilitate the development of such functionalities, the "LRF - Long Running Feature" branches have been created.
Their role is to separate the development of the overall functionality from develop, until it is ready to be merged into develop.

The LRF branches are created exactly like standard feature branches, out of develop, but allow the creation of other feature branches out of it.

A very important distinction is the way the LRF branches are updated and closed.
Usually, whenever a feature branch needs to be updated with the latest changes from develop, it is rebase onto develop.
As there might be feature branches branching out of the LRF branch and referencing its commit ids, it is not possible to do a rebase.
Because of this, the merge operation is required for LRF branches. Instead of rebasing a LRF branch onto develop, develop should be merged into the LRF branch.
The disadvantage of doing this is that the commits on the LRF branch will be scattered among the commits which were added during the merged.
To work around this issue, before the LRF branch is merged into develop, all commits corresponding to it and its subbranches should be squashed into a single commit.



Release Branches

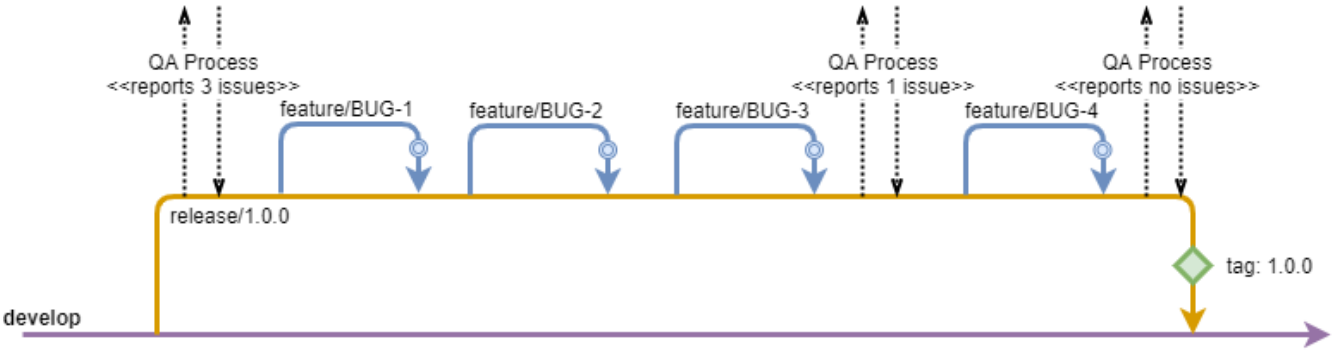
Once develop has accumulated an established amount of features, a new version should be released.
For this, a 'release/' branch will be created, which has the role of a snapshot of develop.
Once the release branch is created, the context of the release has been established and features can be continued to be merged into develop.

During the release process, the main focus will be QA.
Once the release is started, testing should be immediately started.
In the unfortunate case where QA detects an issue, developers will need to provide fixes.
For this, a feature branch should be branched from the release branch for each reported issue.
Similar to normal development, this allows for multiple developers to work independent of each other on the fixes.
As a result, all reported issues are resolved more efficiently, productivity is increased and release time can be considerably shortened.

At the end of the release process, once QA confirms that there are no more issues detected, a tag is created to denote the corresponding version and the release branch is merged into develop and deleted.
The release is merged into develop to make sure that all the changes on the release branch are also included on develop, and, inherently, in the next release.
Similar to the feature branches, release branches are short-lived and exist only during the release process itself.
Once the tag has been created and the branch is deleted, the release is finalized with no further changes allowed.
Any additional changes to the release will be done through hotfixes.

There can always only be, at any given time, one release branch relative to develop and each long term stability 'lts/' branch.
This means you can have multiple release branches, but they cannot have the same source branch (develop or 'lts/').

When a release is finished, either the major or minor versions can be increased.



How to open a release

```
git fetch

git checkout -b <RELEASE>
<SOURCE>

git push origin <RELEASE>

------------
<RELEASE> -
release branch that will
be created
                (release
/x.y.z)

<SOURCE>  -        source
branch for the release
                (origin
/develop, origin/lts/x,
                origin/lts
/x.y)
```

How to close a release

```
# get latest version of
the release

git fetch
git checkout <RELEASE>
git pull

# create version tag and
push it to remote

git tag <VERSION>

git push origin <VERSION>

# create maintenance tag
and push it to remote

git tag --force
<MAINTENANCE>

git push origin
<MAINTENANCE>

# merge release branch
into develop/lts branch

git checkout <TARGET>
git pull

git merge <RELEASE>
git push origin <TARGET>

------------
<RELEASE>     -
release branch that will
be closed

(release/x.y.z)
<VERSION>     -
version of the release,
also included in
                        the
release branch name (x.y.
z)
<MAINTENANCE> - partial
version of the release (x.
y)
<TARGET>      -
target branch for the
release (develop,
                        lts
/x, lts/x.y)
```

Hotfix Branches

Hotfix branches are similar in behavior to release branches, where QA is the main focus. Whenever an issue is reported by QA, feature branches are created to resolve them.
Where hotfixes differ to releases is in how they are created, finished and in their purpose.
A hotfix has the function of providing urgent fixes to existing releases, hence it is created directly from the release tag and, conditionally, merged into develop.
When the hotfix has been finished and approved by QA, a tag will be created, to mark the version. If the hotfix is done on the most recent release, then the hotfix also needs to be merged into develop to ensure that the fix is also included in the next release.
In case of a fix on an older release, the branch is simply deleted without any merge as the fix might not be relevant at all to more recent releases, for whom, hotfixes can also be created, if required.

When a hotfix is finished only the patch value can be increased.

How to open a hotfix

```
git fetch --tags --force

git checkout -b <HOTFIX>
<SOURCE>

git push origin <HOTFIX>

------------
<HOTFIX> -        hotfix
branch that will be
created


(hotfix/x.y.z)
<SOURCE> -        source
branch for the hotfix (x.
y)
```

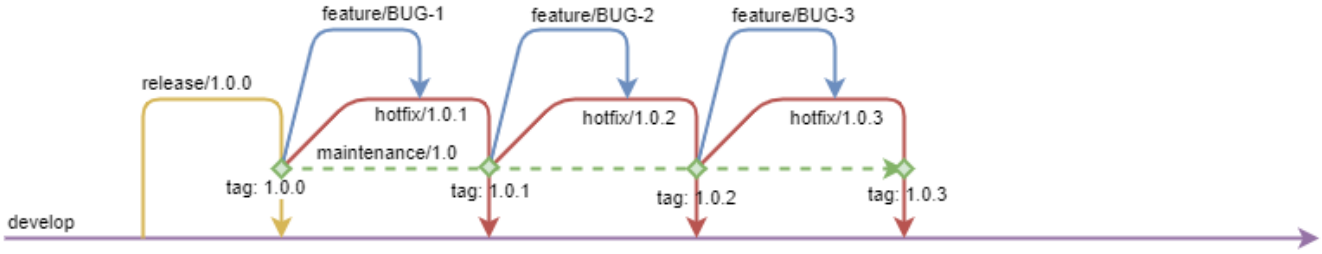
How to close a hotfix

```
# get latest version of
the release

git fetch
git checkout <HOTFIX>
git pull
# create version tag and
push it to remote

git tag <VERSION>

git push origin <VERSION>

# update maintenance tag
and push it to remote
git tag --force
<MAINTENANCE>
git push --force origin
<MAINTENANCE>

# if it is a hotfix to
the latest release,
# then merge into
development branch

git checkout <TARGET>
git pull

git merge <HOTFIX>

git push origin <TARGET>

------------
<HOTFIX>      -
hotfix branch that will
be closed

(hotfix/x.y.z)
<VERSION>     -
version of the hotfix,
also included in
                    the
hotfix branch name (x.y.z)

<MAINTENANCE> - partial
version of the hotfix (x.
y)
<TARGET>      -
target branch for the
release (develop,
                    lts
/x, lts/x.y)
```

Maintenance Branches

Maintenance branches are marker branches, their role is to point to the latest release/hotfix of a release family (ex. 1.0.0, 1.0.1, 1.0.2 - release family 1.0) and are more relevant during the development process.
Their role is to provide a quick and easy way for a developer to start working on fixes for a release.
Withing a release family, there always is a "head" tag, which points to the most recent release.
Whenever providing fixes, the starting point is always the "head" tag as older releases are immutable.
To avoid the identification of the "head" tag, the maintenance branches were created with the sole role of pointing to the most recent release.
This way, a developer no longer needs to have knowledge of the "head" tag, they only need to create a feature branch directly from the corresponding release family maintenance branch.

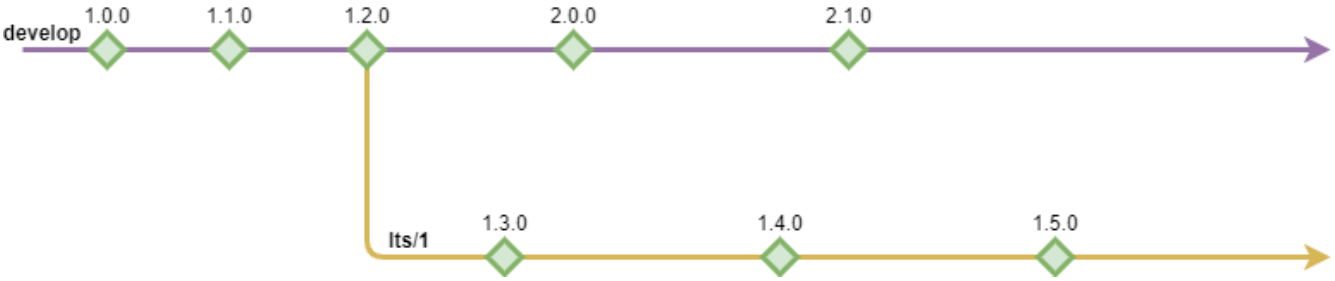
Maintenance branches are created together with a release, when a new release family starts and it is updated during the hotfix process.
It can never contain intermediary changes and will always point to a release/hotfix, "jumping" from one tag to another.
Feature branches should be created out of maintenance branches, but they should NEVER be merged into a maintenance branch.



LTS Long Term Support Branches

There have been use-cases where the developed product got split into different version due to breaking changes, but both versions would need to be actively maintained. (ex. the Angular framework with AngularJS and Angular 2)
In such a case, there are 2 approaches, you can either fork the project into a second one and maintain the two versions as two completely separated projects, or you can maintain them in the same project.
For the second option, SwatFlow supports 'lts/' long term support branches, which allow active development on other branches than develop.
Long term support branches should only be maintained if actual development is planned, for maintaining a release and providing fixes, hotfixes should be able to satisfy any requirement.

Although the lts branches are part of the same repository, the lts versions of a project should be considered as completely separate products.
The reasoning behind this is that, of course, close to the time of separation, the lts branch will be almost identical to develop, but in time the code will become so different that common implementations will not be possible.



Once the lts branch separation occurs, there should not be any more contact with develop or other lts branches.

Deployment and Maintenance of Releases

One of the greatest advantages of SwatFlow is the possibility to maintain all the previously released version and provide fixes to the oldest ones as easy as to the newest ones.
The main idea is that each release allows the creation of a hotfix branch from its tag, providing an isolated snapshot of the state of the project at the time of the actual release.
This allows for easy investigation and implementation of a fix, without being hindered by the SwatFlow in any way.

But despite its full flexibilty, SwatFlow doesn't allow the update of older release relative to a major.minor release group.
For instance, if the following versions have been released: 1.0.0, 1.0.1, 1.0.2 and the customer is on version 1.0.1 and reports an issue, the fix must be implemented starting from version 1.0.2 and will be release as part of version 1.0.3 which also includes the changes in release 1.0.2.
This means that the customer will also receive the 1.0.2 changes, besides the requested changes from 1.0.3.

Because of this limitation, SwatFlow requires that any changes done in a hotfix must be absolutely necessary.
They should provide fixes to existing issues and not introduce new features.
If this principle is respected, then regardless if a customer receives more hotfixes than requested, they will only receive fixes to existing issues reported by them or other customer.

Tags

As it can be seen, tags have an important role in the SwatFlow Workflow and are essential to having short-lived branches.
In SwatFlow there only is one category of tags:

- Version Tags: are created at the end of a release or hotfix branch and denote a specific release with proper major.minor.path values (ex. 1.1.4)

Version Tags

Version tags have the purpose of denoting a proper release and are used to uniquely identify a release/hotfix.
It is a static tag, which should never be changed.
Once a release has been tagged, it cannot be modified in any way.
The only option is to create an additional tag through the hotfix process.

.

Versioning

SwatFlow is fully compatible with GitVersion and is, actually, the recommended versioning system.

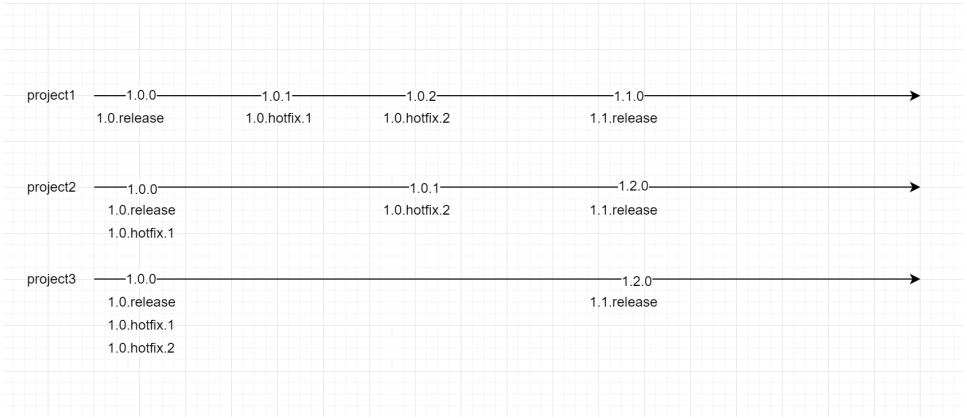
Maintaining the version of a single-project product

As seen above, whenever we finish a release or a hotfix, we tag the commit with a version (ex. 1.1.0, 1.3.2, ...).
Each tag represents a production-ready deployable version of the project.
This works fine when your product is maintained in a single git projects, but becomes more complex when it's spread among multiple ones.

Maintaining the version of a multi-project product

Regardless of having a single-project or multi-project product, each project should have an internal version.
In a single-project product, that internal project version can also be used as the product version, but this is not possible when multiple project are involved.
In this case, we need to have a separate product version. Similar to the project version, the product version can also be maintained through tags.

Below is an example for a product maintained in 3 different git projects:



The versions on the lines represent the corresponding project's internal release version, which is accesible through the corresponding tag.

All the projects are tagged with the initial 1.0.0 version, which will be released as version 1.0.release of the product.
For this we add an additional tag to the 1.0.0 project tags to signal that those versions are used in the 1.0. release.

A hotfix needs to be provided to 1.0. release, but the only required changes are in project1.
Because of this, project1 will need to be hotfixed with version 1.0.1.
We now need to specify which project versions will be included in the 1.0.hotfix.1 release of the product.
For this we add the 1.0.hotfix.1 tag to the following releases:

- project1: 1.0.1
- project2: 1.0.0
- project3: 1.0.0

An additional hotfix is now required, so the next product version will be 1.0.hotfix.2.
This time, both project1 and project2 need to be modified.
Project1 will be hotfixed to version 1.0.2 and project2 will be hotfixed to 1.0.1.
To release the second hotfix, we will need to add the 1.0.hotfix.2 tag to the following releases:

- project1: 1.0.2
- project2: 1.0.1
- project3: 1.0.0

It is decided that version 1.1.0 will be released.
For this, all projects will go through the release process and will have the 1.1.0 tag.
To symbolize the product 1.1. release, we will add the 1.1.release tag to all the projects on the 1.1.0 commit.