# Goals and choices regarding the migration to github

Software Call 2020-12-01

Sebastian Fiedlschuster
https://github.com/fiedl/icecube-git-migration
sebastian.fiedlschuster@fau.de

December 1, 2020

Document 2020-Wa6hohfe

Erlangen Centre for Astroparticle Physics

# Introduction

# Resources

Repository with proposal, tests and experiments:
`https://github.com/fiedl/icecube-git-migration`

Build and run combo on github actions:
`https://github.com/fiedl/icecube-combo-install`

Current combo on github:
`https://github.com/IceCube-SPNO/IceTrayCombo`

Combo will move to:
`https://github.com/icecube/combo`

Current combo on svn:
`http://code.icecube.wisc.edu/svn/meta-projects/combo/trunk/`

These slides as LaTeX:
`https://github.com/fiedl/icecube-git-migration-talk/releases/`

# Motivation and Scope

Care about the upcoming git migration & would like to help bring it forward.

# Motivation and Scope

Care about the upcoming git migration & would like to help bring it forward.

Reached out to github and asked for a username transfer of the empty `icecube` user.

⇒ now we have `https://github.com/icecube`

Plus: Github Education

1

# Motivation and Scope

Care about the upcoming git migration & would like to help bring it forward.

Working on a series of screencasts to
help new users getting started with
icecube software and the git workflow.

## Draft: YouTube Playlist

Todo list: `https://github.com/fiedl/icecube-git-migration/issues/17`

2

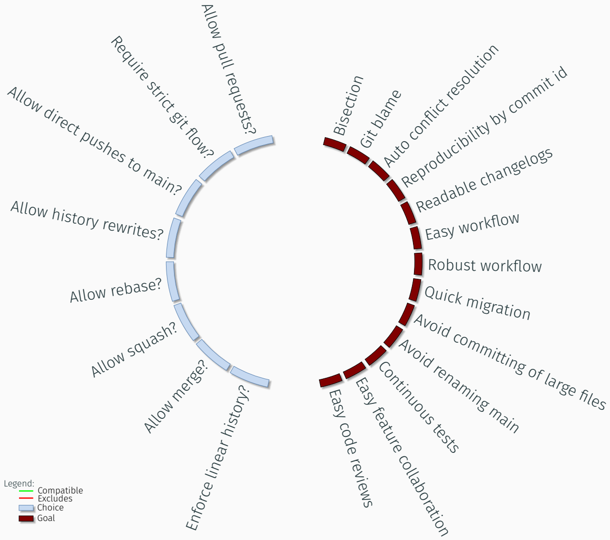
# Motivation and Scope

Care about the upcoming git migration & would like to help bring it forward.

Went through call notes & slack discussions and
summarized choices and goals and
look where they are connected (**this talk**)

3

# Introduction

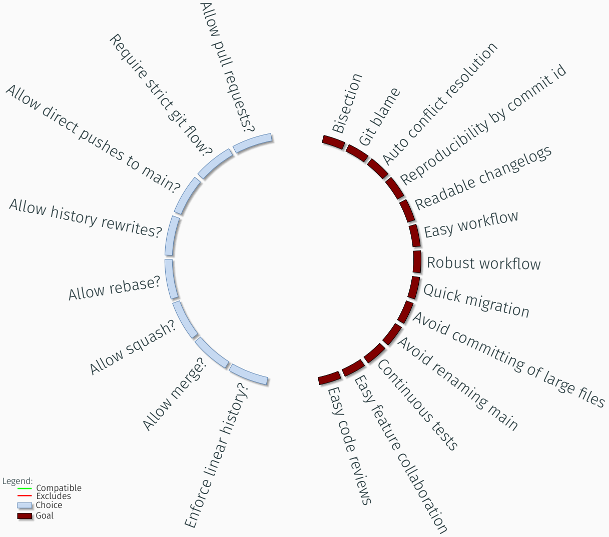


- We do have to make some 🟦 choices regarding the upcoming git migration.
- We have to agree on what 🟥 goals we want to pursue.
- Some of the goals help us to narrow down the options for our choices.
- In the end, we will probably need to make some compromises.

Go straight to the conclusions

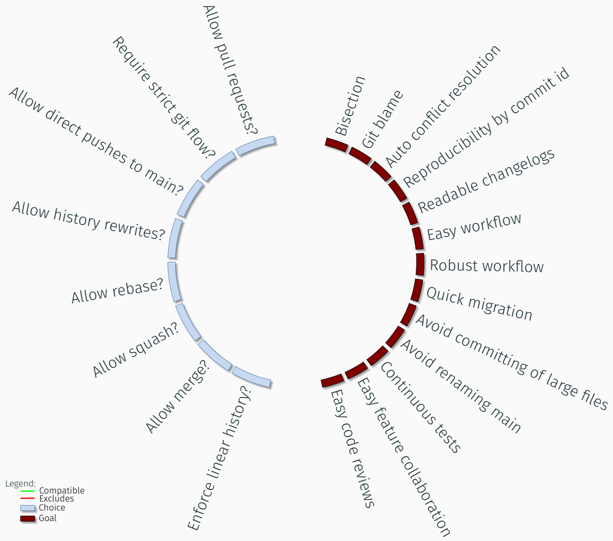
# Goals and Choices

# Choices



- From the software calls and the slack discussion, I gather that the major ▭ choices would be:
  - Allow rebasing commits when merging pull requests?
  - Allow squash commits when merging pull requests?
  - Allow merge commits when merging pull requests?
  - Allow direct pushes to `main` for all users?
  - Strictly require to follow *git flow*?
  - Allow history rewrites?
  - Allow github's pull-request workflow?
  - Enforce linear history?
- Anything missing?
- Lighter versions (*encourage/discourage*) also an option.

# Goals



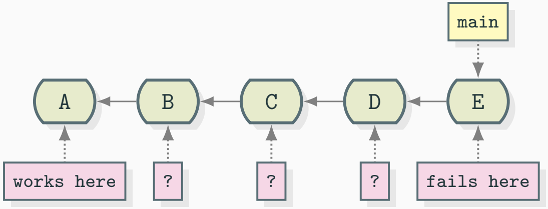
- Anything missing?
- Can weight those later.

- Enable `git bisect` on the `main` branch
- Use `git blame` to find the context of a changed line
- Utilize git's advanced auto conflict resolution
- Enable reproducing results (simulations, plots) by using the same code base identified by git's commit ids.
- Readable changelogs without commit clutter
- Easy workflow: Don't ask too much of our users
- Robust workflow: Don't break stuff. Don't create frustration. Don't produce a situation that needs manual clean up.
- Get our users quickly going after our migration from svn to git
- Avoid accidental commits of large files
- Avoid renaming the `main` branch
- Check code continuously with tests
- Enable easy collaboration on features
- Enable easy code reviews

# Connecting Goals and Choices

# Bisection

# Goals: Bisection

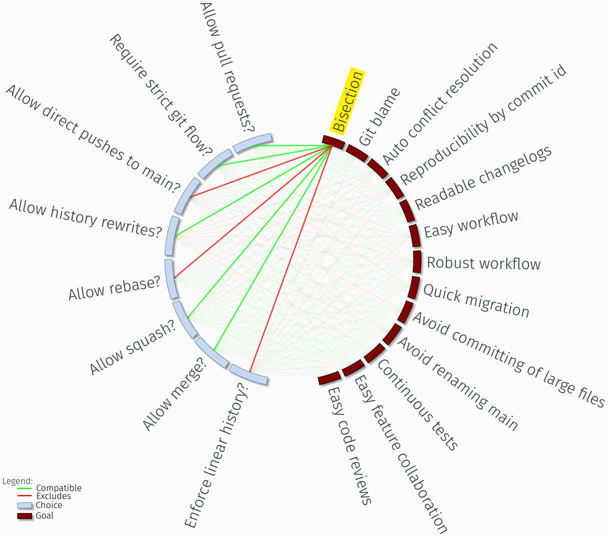


## What is bisection?

- Consider a feature that you know has worked in commit `A`, but fails in commit `E`.
- There are a lot of commits in between. Where did it break?
- Idea: Write a test script and run it with `git bisect` to efficiently find the commit that breaks the test script.

Source: https://git-scm.com/docs/git-bisect

# Goals: Bisection

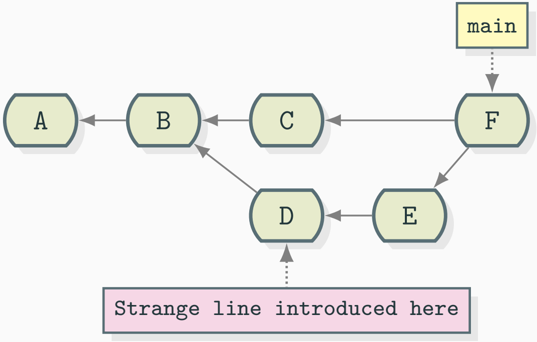


- ✗ Bisection → ¬ Allow rebase merges for pull requests because this adds feature-branch commits to the `main` branch, which might include breaking commits, introducing a flapping behavior, which slows down bisection significantly.
- ✗ This also means that we then can't use rebases to enforce linear history.
- ✗ Bisection → ¬ Allow direct pushes to `main` because direct pushes to `main` would enable people to rename the `main` branch. This pollutes the `main` branch with feature-branch commits or even removes commits from the `main`, which prevents bisection on `main`.
- ✓ Bisection does work with Merge commits because one can skip the commits of the feature branches from bisection using the `--first-parent` parameter.

Source: https://github.com/fiedl/icecube-git-migration/issues/11, https://github.com/fiedl/bisect-only-main-branch-test/issues/2, https://blog.quantic.edu/2015/02/03/git-bisect-debugging-with-feature-branches/

# Git blame

# Goals: Git blame

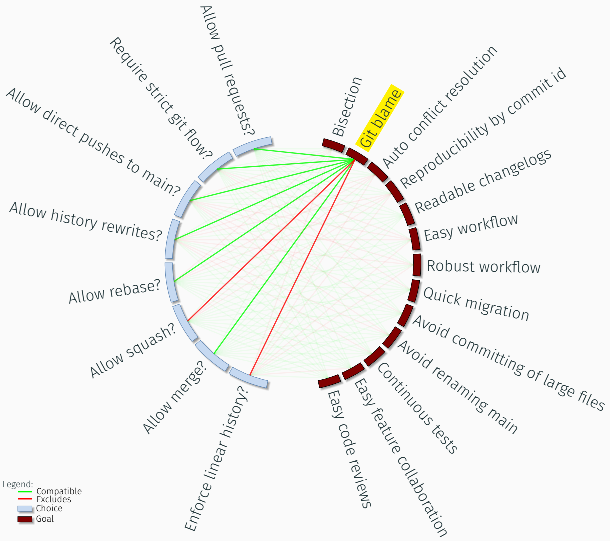


### What is `git blame`?

- Suppose, you are on `main` and find a strange code line in `code.py` you can't make sense of.
- Idea: To get an idea of the context the line was introduced in, run `git blame code.py`, which helps you to identify the commit `D` that has introduced the line.
- Now look at the commit message of commit `D` and the other changes made in this commit.

Source: `https://git-scm.com/docs/git-blame`

# Goals: Git blame

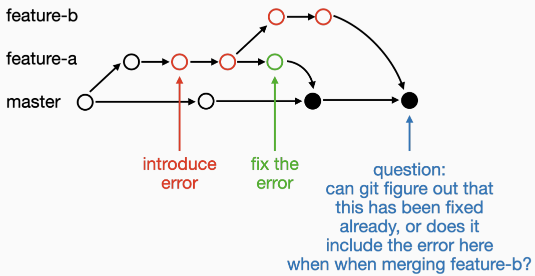


- ✗ When using `git blame` with squash commits, the summary commit ( `F` in the example) is the commit returned by `git blame` . If there are many changes, this context might not be specific enough to make sense of the line in question.
- ✗ This also means that we lose this detailed `git blame` if we use squash commits to enforce linear history.
- ✓ When using `git blame` with merge commits, the specific commit introducing the line in question can be found.
- ✓ The `git blame` behavior of squash commits, i.e. returning the summary commit, can be emulated when using merge commits by using `git blame --first-parent` .

Source: hhttps://www.felixmoessbauer.com/blog-reader/why-git-squash-merges-are-bad.html, https://git-scm.com/docs/git-blame#Documentation/git-blame.txt---first-parent

# Auto conflict resolution

# Goals: Auto conflict resolution


feature-b
feature-a
master
introduce error
fix the error
question: can git figure out that this has been fixed already, or does it include the error here when when merging feature-b?

Real-world example: You open a pull request and need to base other work on that feature branch while waiting for pull-request approval.

Source: `https://git-scm.com/docs/merge-strategies`

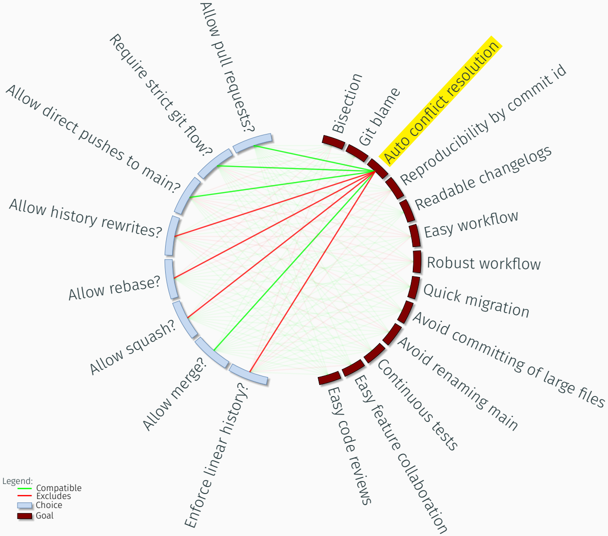
## What is automatic conflict resolution?

- Git uses a directed acyclic graph (DAG) to store which commits a new commit is based on. This makes git very powerful when it comes to resolving merge conflicts automatically.
- Suppose, an error is introduced in the `feature-a` branch. The `feature-b` branch is based on the `feature-a` branch and does also include this error. The error is later fixed in `feature-a` and the fix is merged into `main`.

✗ Without the DAG information, git would re-introduce the already fixed error into `main` when merging `feature-b` because it could only compare the two code states of `main` and `feature-b` (*2-way merge*).

✓ With the DAG information, the fix takes precedence when merging `feature-b` because git compares the changes to the common ancestor commit, which can be determined from the DAG (*3-way merge*).

# Goals: Auto conflict resolution



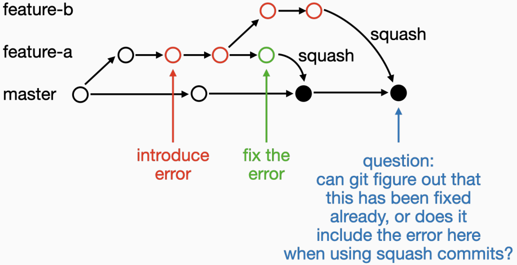
✗ Auto conflict resolution → ¬ Allow rebase merges

✗ Auto conflict resolution → ¬ Allow squash merges

✗ Auto conflict resolution → ¬ Enforce linear history

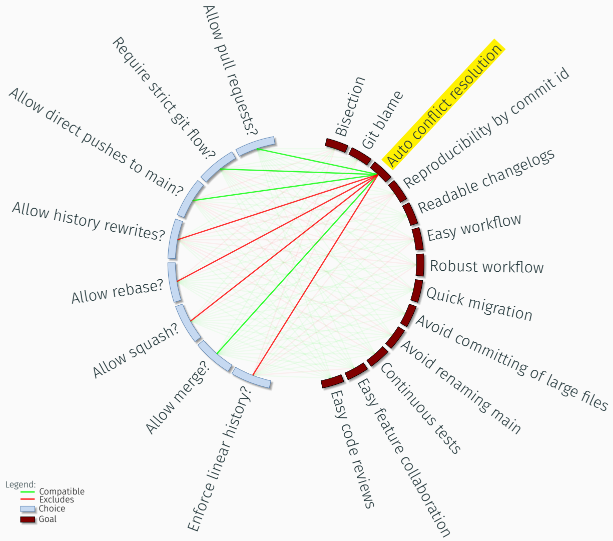
because rebase commits and squash commits do not keep the graph dependency information to the commits in the feature branch. If one would base work onto commits from the feature branch, conflicts could not be resolved automatically, later.

Experiment: https://github.com/fiedl/icecube-git-migration/issues/15



feature-b
feature-a
master
squash
squash
introduce error
fix the error
question: can git figure out that this has been fixed already, or does it include the error here when using squash commits?

# Goals: Auto conflict resolution



- Auto conflict resolution → ¬ Allow history rewrites

  because this destroys the dependency information to any own work in cloned repositories.

  We would need to make sure that *everyone* re-clones the repo and re-commits his own work before creating a pull request. Otherwise, git can't resolve the dependencies and merge the pull request.

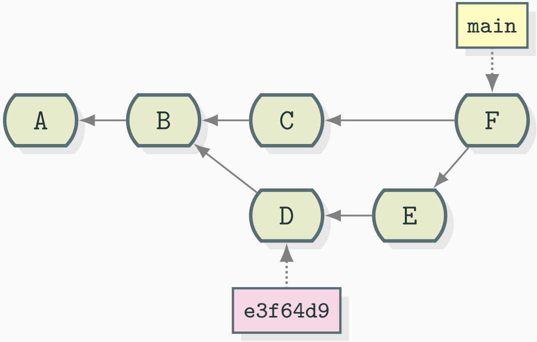
  One of the golden git rules is:

  > *Do not rebase commits that exist outside your repository and that people may have based work on. … If you don't, people will hate you, and you'll be scorned by friends and family.*

  https://git-scm.com/book/en/v2/Git-Branching-Rebasing#_rebase_peril

# Reproducibility by commit id

# Goals: Reproducibility by commit id

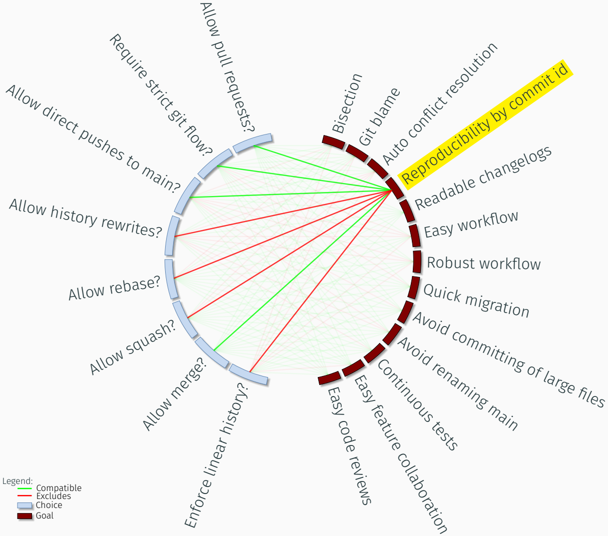


## What is reproducibility by commit id?

- Git uses unique commit ids, e.g. `e3f64d9` to reference repository state.
- If one does know the commit id, one can reproduce the repository state regardless of the branch the commit has been made on.
- Idea: Have icetray output and log the commit id of the combo repository along datetime when executing. Encourage people to commit before executing noteworthy runs.
- This way, the run results can be reproduced using the respective commit ids of combo and the script's repo.

Source: https://git-scm.com/book/en/v2/Git-Tools-Revision-Selection

# Goals: Reproducibility by commit id



- ✗ Reproducibility → ¬ Allow rebase merges for pull requests
- ✗ Reproducibility → ¬ Allow squash merges for pull requests
- ✗ Reproducibility → ¬ Enforce linear history
- ✗ Reproducibility → ¬ History rewrites

because the original commits can become dangling (*unreachable*) and than be removed by garbage collection. Then, they the original commits cannot be accessed anymore.

See also: `https://github.com/fiedl/icecube-git-migration/issues/18`

Source: `https://git-scm.com/docs/git-gc`

# Readable changelogs

# Goals: Readable changelogs

## Bad changelog

```
e3f64d9e  fixing issue
0a61924e  Merge branch 'fix' into 'main'
a91acb11  Working on cable shadows...
361ad89d  Fix steamshovel
```

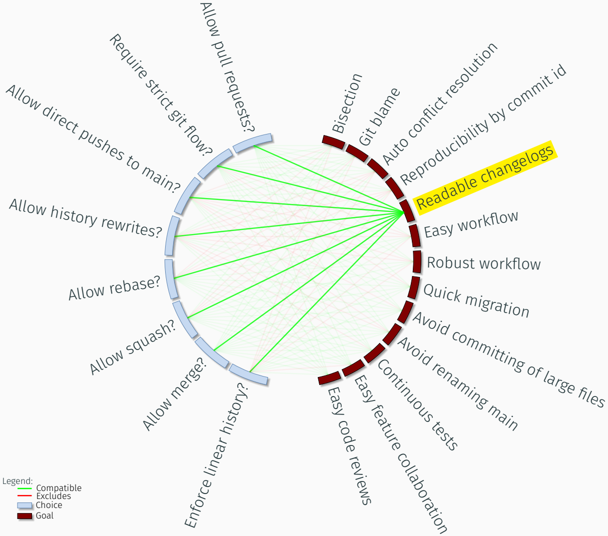
## Better changelog

```
a1e5e5c7  [clsim] Cable shadows: Make the cable radius configurable.

          When using direct cable simulations, add the cable radius
          using the 'cable_width' parameter of the 'I3CLSimFoo'
          module.

          Example:

              tray.AddModule("I3CLSimFoo",
                direct_cable_simulation = True,
                cable_width = 5 * I3Units.centimetres
              )

          Defaults to 3cm.

          See also: https://wiki.icecube.wisc.edu/index.php/Cables

0a61924e  [steamshovel] Fixing syntax error that has prevented

          steamshovel from displaying any photon tracks.

          Fixes https://github.com/icecube-spno/IceTrayCombo/issues/12345
```

Even better would be adding links, code highlighting, etc.

Source: https://git-scm.com/docs/git-log

# Goals: Readable changelogs



✓ Readable changelogs go well with merge commits, even if the commit messages of the feature branches are less readable.

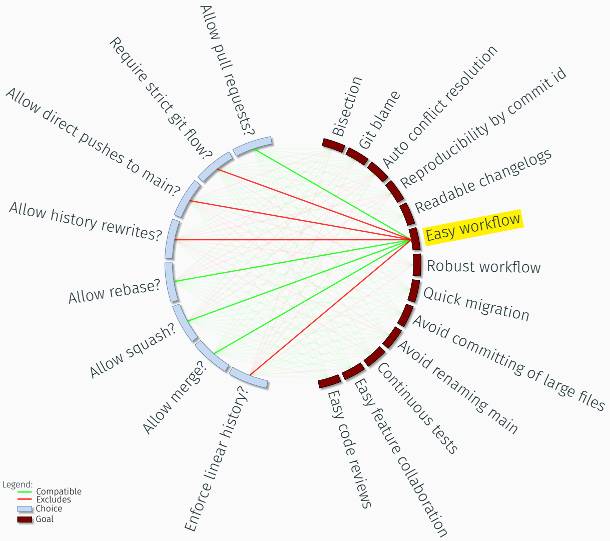
In the terminal, one can show only the commits on the `main` branch and hide the commits of the feature branches using `git log --first-parent`.

We could also generate `CHANGELOG.md` files for the combo repo and its project directories by tying a changelog generator into a *github action*.

Source: https://github.com/fiedl/bisect-only-main-branch-test/issues/2, https://github.com/fiedl/icecube-git-migration/issues/11, https://github.com/topics/changelog-generator

# Easy workflow

# Goals: Easy workflow



- An easy workflow would be one that is easy and fast to learn by people new to icecube or git.

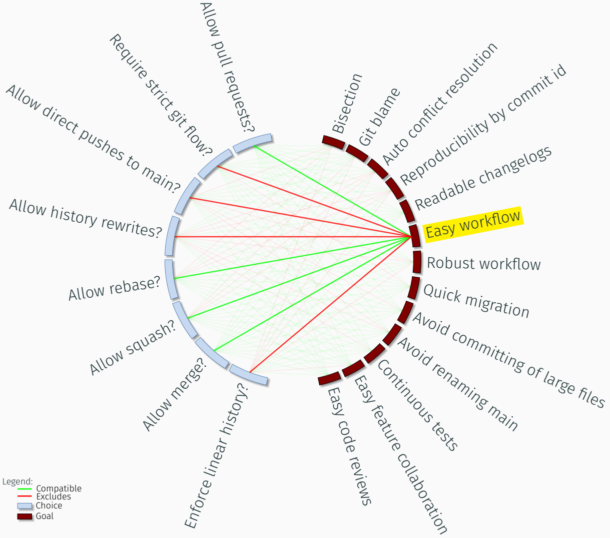
✗ Easy workflow → ¬ Require strict git flow because newcomers would need to lookup many conventions and would be required to think about these and other questions before committing any code:

- Is my change a hot fix?
- Is my change a feature?
- On which branch do I need to base my work on?
- Do I need to create this pull request against `main` / `dev` / `whatever` ?
- Do I need to cherry-pick this patch to some other branch as well?

✓ While git flow is very structured and considered robust, a simple github pull-request workflow would be more approachable for new users.

Source: `https://nvie.com/posts/a-successful-git-branching-model/`, `https://guides.github.com/introduction/flow/`

# Goals: Easy workflow



Source: https://nvie.com/posts/a-successful-git-branching-model/, https://guides.github.com/introduction/flow/

✗ Easy workflow → ¬ Allow direct pushes to main for all users

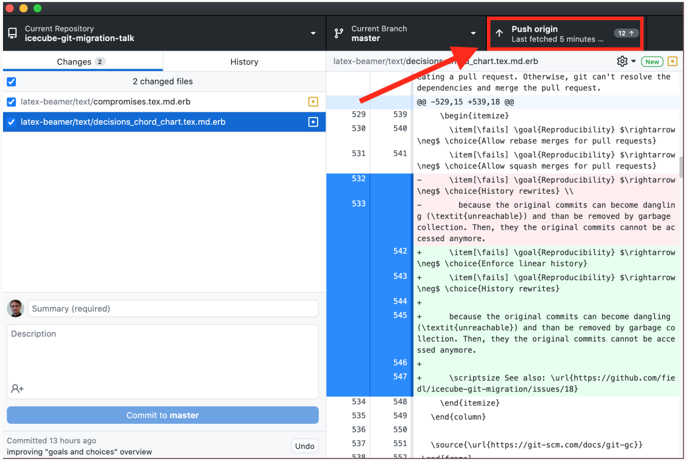
- because all users would need to be careful not to rename the `main` branch by accident.
- When pushes to `main` are rejected due to commits on `main` created by other users in the meantime, this will create frustration for new users. They will want to publish their changes with minimum effort and may even avoid pushing their changes if they find it hard.

✓ With a standard github pull-request workflow, these problems do not arise.

→ Nathan: Allow commits to `main`, but encourage to use pull requests (*"if you don't know what you are doing and why, do this"*).

→ Clear guides, screencasts

# Goals: Easy workflow



Current Repository
icecube-git-migration-talk

Current Branch
master

☑ 2 changed files

☑ latex-beamer/text/compromises.tex.md.erb

☑ latex-beamer/text/decisions_chord_chart.tex.md.erb

Committed 13 hours ago
improving "goals and choices" overview

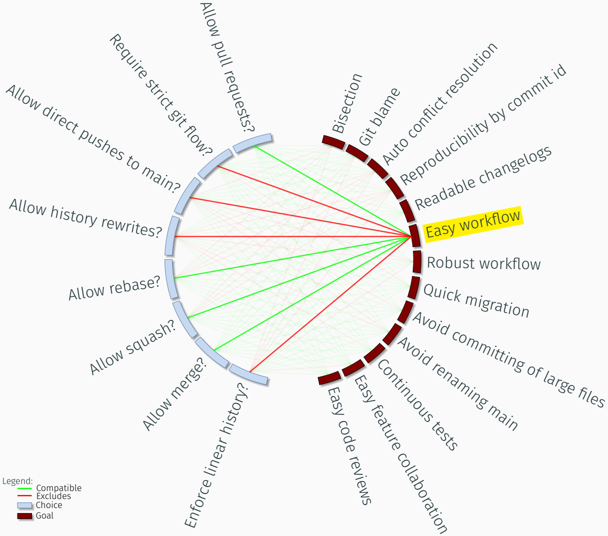
latex-beamer/text/decisions … d_chart.tex.md.erb

```
eating a pull request. Otherwise, git can't resolve the
dependencies and merge the pull request.
@@ -529,15 +539,18 @@
     \begin{itemize}
       \item[\fails] \goal{Reproducibility} $\rightarrow
\neg$ \choice{Allow rebase merges for pull requests}
       \item[\fails] \goal{Reproducibility} $\rightarrow
\neg$ \choice{Allow squash merges for pull requests}
-      \item[\fails] \goal{Reproducibility} $\rightarrow
\neg$ \choice{History rewrites} \\
-        because the original commits can become danglin
g (\textit{unreachable}) and than be removed by garbage
collection. Then, they the original commits cannot be ac
cessed anymore.
+      \item[\fails] \goal{Reproducibility} $\rightarrow
\neg$ \choice{Enforce linear history}
+      \item[\fails] \goal{Reproducibility} $\rightarrow
\neg$ \choice{History rewrites}
+
+      because the original commits can become dangling
(\textit{unreachable}) and than be removed by garbage co
llection. Then, they the original commits cannot be acce
ssed anymore.
+
+      \scriptsize See also: \url{https://github.com/fie
dl/icecube-git-migration/issues/18}
     \end{itemize}
   \end{column}

   \source{\url{https://git-scm.com/docs/git-gc}}
```

- Note: Clicking the *Sync* button on the github application will push directly to `main` if not protected and not switched to another branch.

→ Need to watch out for problems

# Goals: Easy workflow



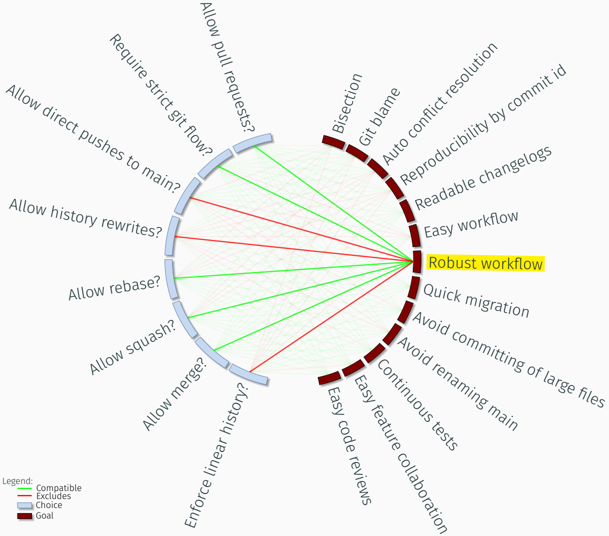
✗ Easy workflow → ¬ History rewrites

- because rewrites would require everyone to re-clone the repo and transfer his/her own work to the re-cloned repo before pushing anything.
- `git pull` will be rejected without obvious reason.
- Transferring existing commits to the new history works, but requires manual management of *branches* or *remotes* (not easy).

Source: https://nvie.com/posts/a-successful-git-branching-model/, https://guides.github.com/introduction/flow/

# Robust workflow

# Goals: Robust workflow



- A robust workflow would be one that prevents regular users from causing damage that would require admins to perform fixes or clean-ups.

✗ Robust workflow → ¬ Allow direct pushes to main for all users

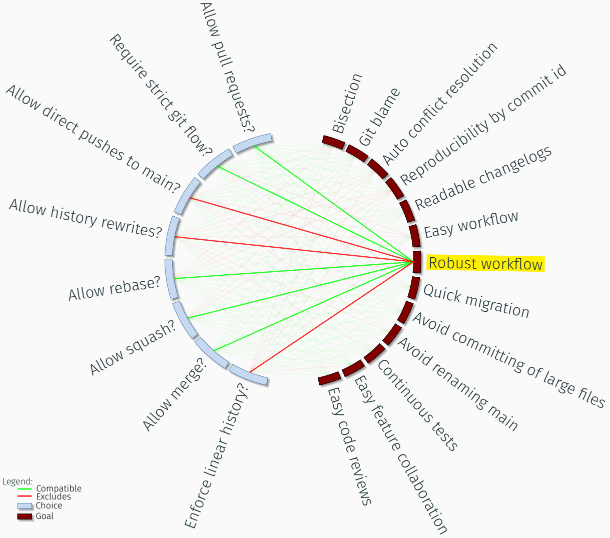
  ✗ because it would allow anyone to rename the `main` branch, which would require clean-up, history rewrites, or break bisection and readable changelogs.

  ✗ Pushes to main circumvent automated checks.

  ✗ Pushes to main are not revertable without history rewrites.

- But: Preventing pushes to `main` could also cause friction → need good guides

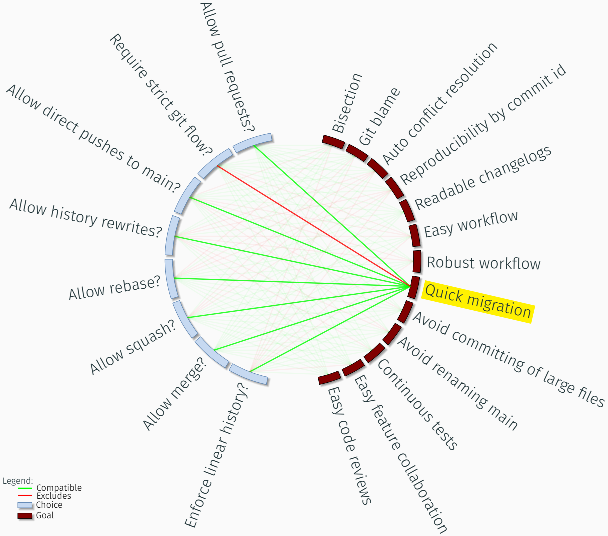
# Goals: Robust workflow



- ✗ In the same sense, enforcing linear histories also causes friction. Worst case: Parallel, long-lived forks and long-term fragmentation.
- ✗ Robust workflow → ¬ Allow history rewrites
  - ✗ because creating pull requests based on the old history won't work due to *unrelated history* error.
  - ✗ Pushing commits based on the old history will re-upload the old history resulting in having both, original and duplicate commits in the same git graph.

# Quick migration

# Goals: Quick migration



- A quick migration would be one that enables everyone in icecube to use the github repository, contribute code, issues, reviews, etc. soon without risks.

✗ When requiring strict git flow, everyone would need to learn the git-flow conventions before contributing.

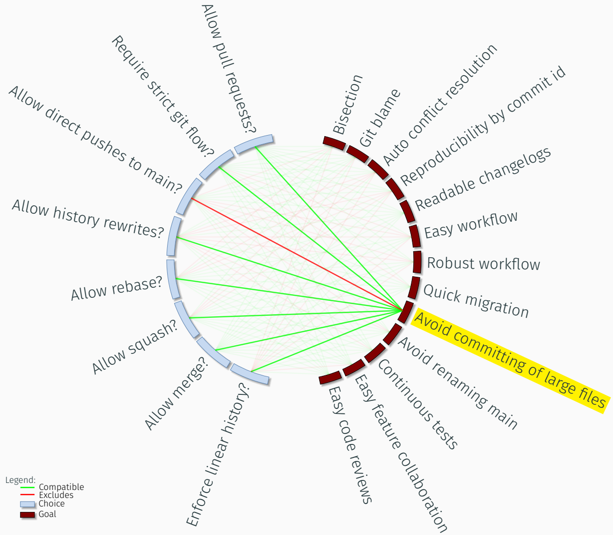
✓ In contrast, people might already know the pull-request based github standard workflow.

  ✓ Github has worked hard to create a workflow that works well for large projects as well for new contributors.

  ✓ Don’t work against the tools (as this would create additional friction and additional work for our admins).

# Avoid accidental commits of large files

# Goals: Avoid accidental commits of large files



- We want to avoid accidental commits of large files, e.g. `*.i3` files, that do not gain anything from version control, because they would make the repository larger and slower.

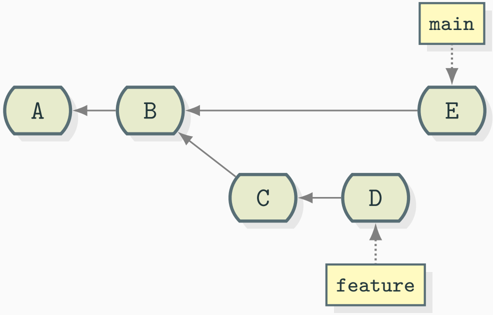
✓ Providing a suitable `.gitignore` file can help.

✓ We can check for large files in *automated checks* when using a pull-request workflow.

✗ Allowing direct pushes to main circumvents automated checks.

# Avoid renaming the `main` branch

# Goals: Avoid renaming the `main` branch



## What is renaming the `main` branch?

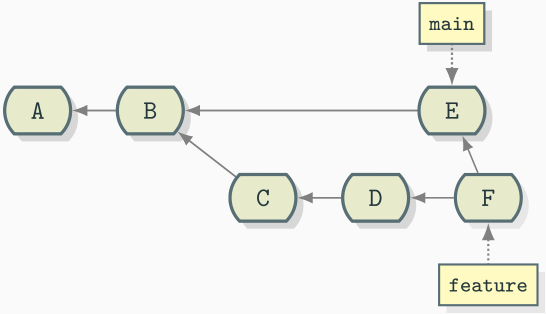
- Consider a `main` branch and a `feature` branch.
- The author of the `feature` branch pulls in the `main` branch to get the changes that have been committed to the `main` branch in the meantime:

```
git checkout feature
git merge main
```

- The author of the `feature` branch merges his feature branch in to `main`, which would be a fast-forward merge at this point.

```
git checkout main
git merge feature
```

# Goals: Avoid renaming the `main` branch



## What is renaming the `main` branch?

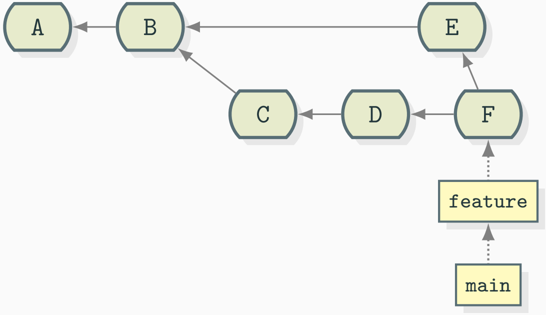
- Consider a `main` branch and a `feature` branch.
- The author of the `feature` branch pulls in the `main` branch to get the changes that have been committed to the `main` branch in the meantime:

```
git checkout feature
git merge main
```

- The author of the `feature` branch merges his feature branch in to `main`, which would be a fast-forward merge at this point.

```
git checkout main
git merge feature
```

# Goals: Avoid renaming the `main` branch



## What is renaming the `main` branch?

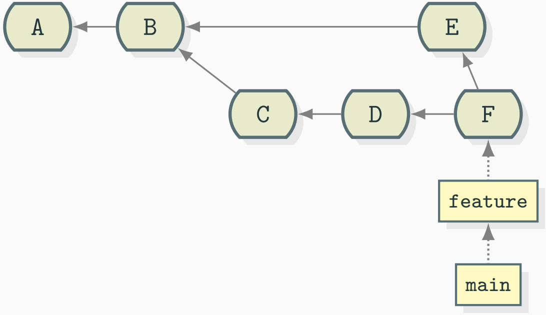
- Consider a `main` branch and a `feature` branch.
- The author of the `feature` branch pulls in the `main` branch to get the changes that have been committed to the `main` branch in the meantime:

```
git checkout feature
git merge main
```

- The author of the `feature` branch merges his feature branch in to `main`, which would be a fast-forward merge at this point.

```
git checkout main
git merge feature
```

# Goals: Avoid renaming the `main` branch



The `main` branch now contains the commits `A B C D F` rather than `A B E`.

This breaks bisection and readable changelogs.

## What is renaming the `main` branch?

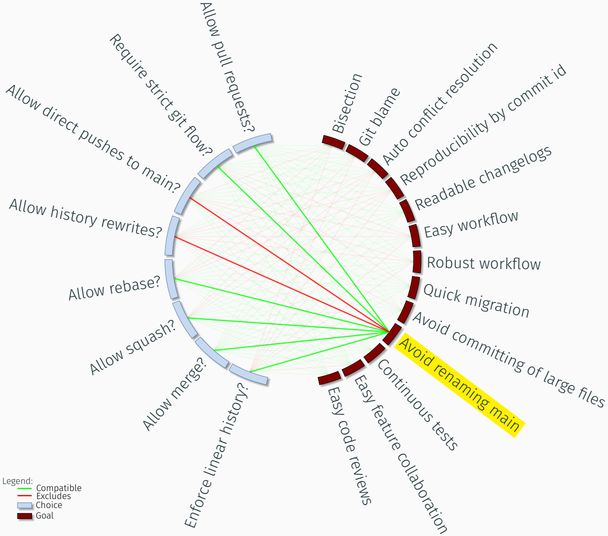
- Consider a `main` branch and a `feature` branch.
- The author of the `feature` branch pulls in the `main` branch to get the changes that have been committed to the `main` branch in the meantime:

```
git checkout feature
git merge main
```

- The author of the `feature` branch merges his feature branch in to `main`, which would be a fast-forward merge at this point.

```
git checkout main
git merge feature
```

# Goals: Avoid renaming the `main` branch



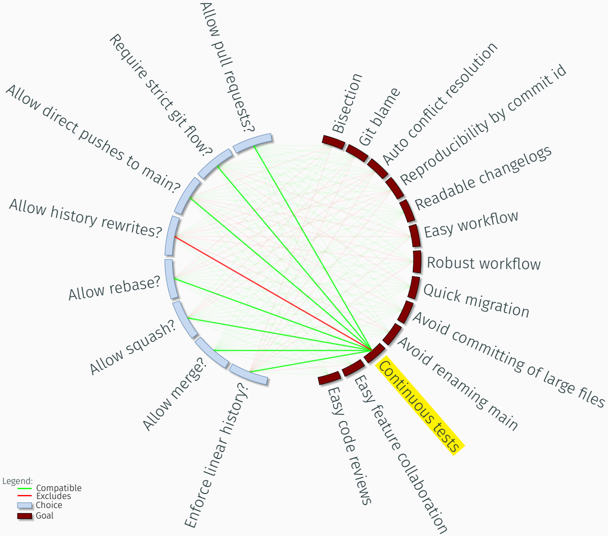
✓ Using a standard pull-request workflow does not allow to rename the `main` branch.

✗ Allowing direct pushes to main allows to rename the `main` branch.

✗ Rewriting history, strictly speaking, also renames the `main` branch because it moves the `main` branch label to another commit.

# Continuous tests

# Goals: Use continuous tests

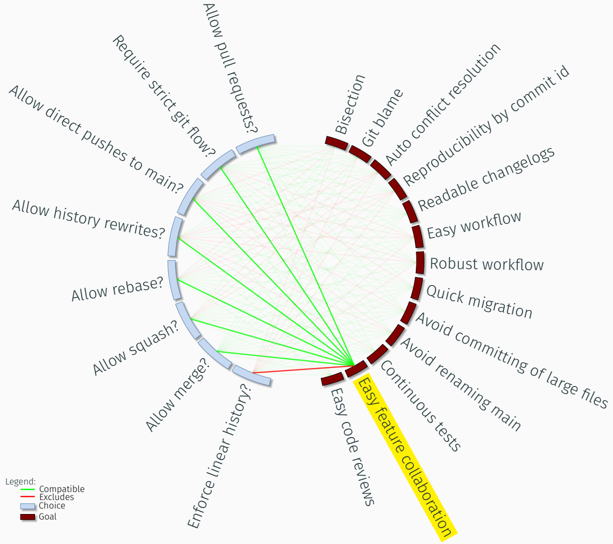


- Using continuous tests that run on each commit and on each pull request before merging help to avoid or resolve issues before landing in the `main` branch.
- Test results are archived and can be accessed from the commits or pull requests.

✗ Rewriting history destroys the references between commits and the continuous test runs. So we would lose the information on which commits the tests succeeded. Or we would have to re-run the tests on all commits of the new history.

# Easy feature collaboration

# Goals: Easy feature collaboration



✓ Collaborating on feature branches works well with git flow as well with the github pull-request workflow.

✗ Allowing rebase commits and squash commits in general are no deal-beaker for feature collaboration, but should be avoided in this context.

One of the golden git rules is:

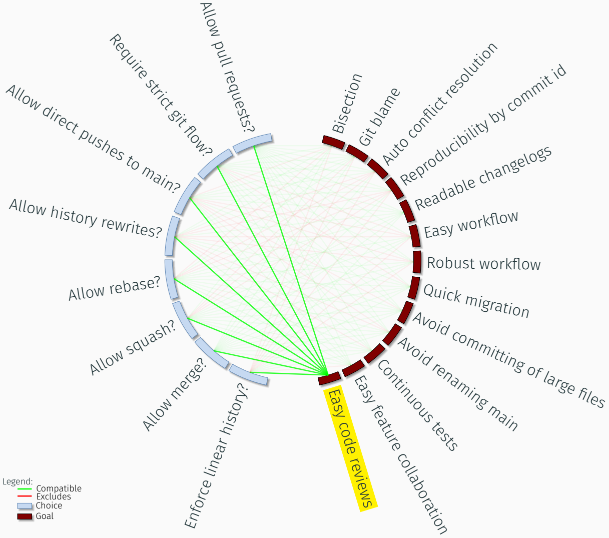
> *Do not rebase commits that exist outside your repository and that people may have based work on. … If you don't, people will hate you, and you'll be scorned by friends and family.*

https://git-scm.com/book/en/v2/Git-Branching-Rebasing#_rebase_peril

✗ This is also a scenario where enforcing linear history would do more harm than good because this would break the dependency graph → need to be much more careful, not easy.

# Easy code reviews

# Goals: Easy code reviews

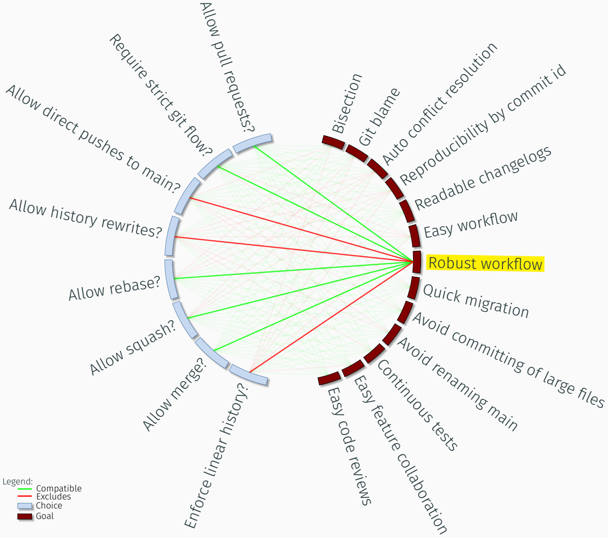


✓ Code reviews are baked into the standard github pull-request workflow.

✗ Code reviews don’t work for pushing code directly to the `main` branch.

# Conclusions

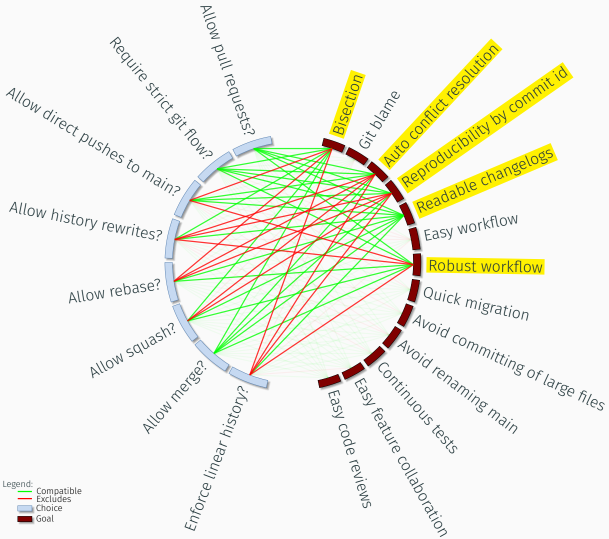
# Conclusions



## To which choices do these goals lead us?

- Aiming for a robust workflow, we need to forbid history rewrites and forbid, restrict, or at least discourage direct pushes to the `main` branch. (Slide 20)
- Aiming for bisection, auto conflict resolution and reproducibility requires us to deactivating rebase pull requests and preferring merge-commit pull requests over squash-commit pull requests. (Slides 7, 11, 13)
- Aiming for a quick migration and an easy-to-learn workflow, we should probably go with github's standard pull-request workflow rather than a strict git-flow workflow. (Slides 19, 21)
- The other goals considered on these slides are compatible with these choices.

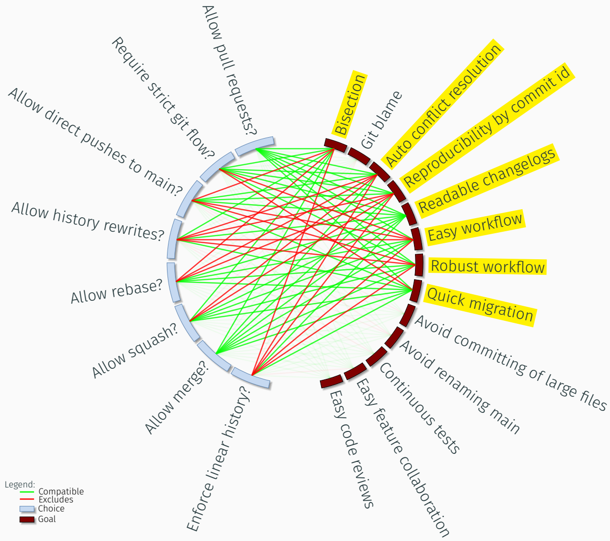
# Conclusions



## To which choices do these goals lead us?

- Aiming for a robust workflow, we need to forbid history rewrites and forbid, restrict, or at least discourage direct pushes to the `main` branch. (Slide 20)
- Aiming for bisection, auto conflict resolution and reproducibility requires us to deactivating rebase pull requests and preferring merge-commit pull requests over squash-commit pull requests. (Slides 7, 11, 13)
- Aiming for a quick migration and an easy-to-learn workflow, we should probably go with github's standard pull-request workflow rather than a strict git-flow workflow. (Slides 19, 21)
- The other goals considered on these slides are compatible with these choices.

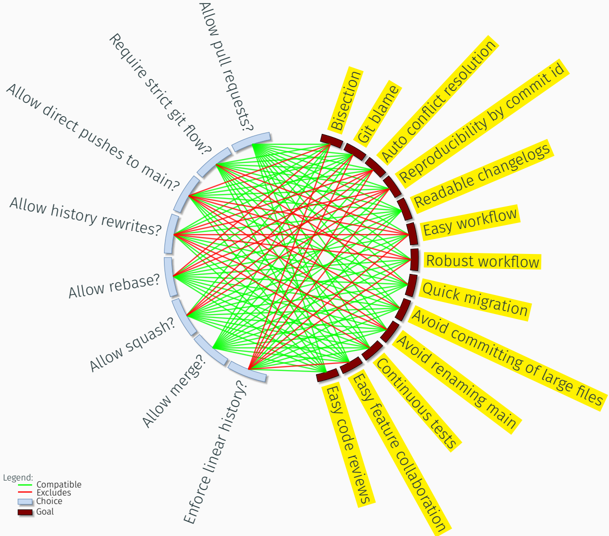
# Conclusions



## To which choices do these goals lead us?

- Aiming for a robust workflow, we need to forbid history rewrites and forbid, restrict, or at least discourage direct pushes to the `main` branch. (Slide 20)
- Aiming for bisection, auto conflict resolution and reproducibility requires us to deactivating rebase pull requests and preferring merge-commit pull requests over squash-commit pull requests. (Slides 7, 11, 13)
- Aiming for a quick migration and an easy-to-learn workflow, we should probably go with github's standard pull-request workflow rather than a strict git-flow workflow. (Slides 19, 21)
- The other goals considered on these slides are compatible with these choices.

# Conclusions



## To which choices do these goals lead us?

- Aiming for a robust workflow, we need to forbid history rewrites and forbid, restrict, or at least discourage direct pushes to the `main` branch. (Slide 20)

- Aiming for bisection, auto conflict resolution and reproducibility requires us to deactivating rebase pull requests and preferring merge-commit pull requests over squash-commit pull requests. (Slides 7, 11, 13)

- Aiming for a quick migration and an easy-to-learn workflow, we should probably go with github's standard pull-request workflow rather than a strict git-flow workflow. (Slides 19, 21)

- The other goals considered on these slides are compatible with these choices.

## Time to compromise

- Want to go *somewhere forward*
- Not everyone would support the same goals/choices
  or does care to use the same git features/tools.
- I've talked to Nathan and other people that I thought would prefer other choices than myself.

→ Compromise

## Time to compromise

- In principle, allow much. Otherwise, there will be friction, and there will be forks with other settings. But give clear encouragements in easy-to-digest guides.
  - Allow direct pushes to `main`. But encourage pull-request workflow & provide good guides.
  - Don’t enforce but encourage linear histories. There are valid cases when using non-linear histories is useful. But guide users how to keep the changelog clean.
  - Also up to the maintainer or admin whether to squash/rebase/merge a particular pull request. There are valid cases for each. Inexperienced contributors may need more guidance here.
- Accept that this model is not bullet-proof for now and allow iterations of this workflow (Erik).
- Don’t forget to prompt the commit id rather than the svn revision in icetray.
- Never rewrite history when people had the chance to base work on it.

## Thanks for your attention!

# What do you think?

Any input you might have is welcome:

`https://github.com/fiedl/icecube-git-migration/issues`

`sebastian.fiedlschuster@fau.de`

Slack: `@fiedl`, `#git_migration`

# Backup slides

# Wishlist Nathan Whitehorn i

See also: Slides | #9 | #5

- Commit authors need to be identifiable (have meaningful emails)
  - We can ensure this in pull requests on github with checks
  - We can run the same tests after pushes to master, and then send warnings, or even reset master, and move the changes into a feature branch
- Answer questions like “what was the repository state on May 25th?”
  - We can achieve this by logging date and commit id into a log file when executing `env-shell.sh` and icetray.
- Is access control for sub-folders an issue?
  - We cannot restrict write access to sub folder when allowing pushes to master (this would require gitlab or github enterprise)
  - What would be the benefit of having project-wise access control? Would it be worth the effort?
  - We can run access checks after pushes to master, and then send warnings, or even reset master

# Wishlist Nathan Whitehorn ii

- How do we prevent big-file commits?
  - We can do checks for pull requests, but not reject pushes to master based on these checks
  - We can run the same tests after pushes to master, and then send warnings, or even reset master, and move the changes into a feature branch
- Git migration of combo drops svn commits before Oct 2018
  - We can still include other histories (which I am in favor for)
  - But from the git graph, it is not obvious then that the code has been merged in 2018
- Migrate trac issues to github tickets? How could we keep the existing links intact?
  - We can open github issues in any case
  - We can move open trac issues to github if we want, and link the new github issues in the trac thread (but we could also keep track of currently open issues in trac, i.e. live with both systems in parallel for now)

# Wishlist Nathan Whitehorn iii

- What to do with metaprojects?
  - They can still live in svn and still use svn externals to include the code of projects living in combo.
- What to do with sandboxes?
  - Erik suggests to have them in (possibly private) repos in the users' namespace on github.
- What to do with papers?
  - Erik suggests solutions (slide 7)
- How do we move pre-highlander history to git (important information)?
  - We can import those histories as well
  - But from the git graph it will not be clear when the merges actually have happened

See also: Slides | #9 | #5

# Wishlist Erik Blaufuss and Don La Dieu

- Sandboxes: Use private github account; also svn sandboxes will not disappear.
- Also plan for analysis code and papers (slide 7)
- Suggests shadow projects to allow release-based imports into meta projects
  - We can have svn meta projects without shadow projects using svn externals and github's svn bridge, or by using a manager like gitman
  - See also: Discussion: Single combo repo vs. separate project repos
- Svn is not going away
  - We can keep our svn repo readonly
  - Maybe we can use github's svn bridge
  - But we should *not* allow a separate svn repo for write access to save us from the work of sorting and cross-syncing

See also: Slides | #9 | #7

# Wishlist Alex Olivas i

See also: #9 | #8

- the main sticking point is figuring how to make the git workflow work for us at scale. Or really coming up with some dev model (gitflow variant) that works.
    - encourage pull requests and code reviews
    - but allow commits to master
    - add people as collaborators to the combo repo, such that they don't need to fork, but can push feature branches
- to answer questions like "what was the state of combo at 23 October" (because "it has worked then, why doesn't it now") is an issue, but a minor one. this is a natural problem with git that we'll just have to get used to.
    - See discussion: The code was running at 23 October
    - We should log the commit id and the current date time to a log file when executing the env-shell.sh and icetray. This way, we could ask people that need support to grep that log file (`grep 2019-10-23 /var/log/icetray.log`) and give us the commit id from the output.

# Wishlist Alex Olivas ii

- If we go with vanilla gitflow, where everyone forks and submits a PR, what's the process for accepting PRs that's sustainable? We have 100 projects, almost 40 developers on paper, and up to 100 collaborators committing in a year. Through the upgrade when we're expecting more activity, if we go with the Linux kernel model, then I'm probably the Linus and I approve all PRs going into the master branch. Or do we go with something closer to what we have now where people can commit directly to anything, with little oversight.
    - a good start would be a couple of automated requirements like checking for over-sized files, an email address being present, and CI tests
    - reviews, while not being mandatory, could increase the confidence in a pull request, and increase the speed it is going to be merged
    - if we deny pushes to master, the frustration might be higher, because something people were used to do does not work anymore. But the migration should feel good, instead.
    - What we can do: Run checks also after pushes to master and have the bot send feedback to people. (Your code has been moved into this feature branch, because …; check results; this is how you can add more commits, …) – People can still work as usual, but get constructive feedback.
    - Later on, we could still decide to deny pushes to master.

# Wishlist Alex Olivas iii

- I'll actively discourage collaborators from clinging to svn.
  - We can use Github's svn bridge for svn externals of meta projects
  - If people really want to still use svn, they'll probably figure out that the svn bridge is there (`https://lmgtfy.com/?q=github+svn`)
  - But we should tell people about the advantages of using git
- Should we allow people to push to master? Or should we require pull requests? Or some hybrid?
  - I vote for allowing pushes to master, at least in the beginning, in order to avoid mega frustration
  - it's just one setting in the github repo
- Should combo be one repo, or git submodules, or some kind of package manager? The main concern are separate meta projects (e.g. ehe or pnf) that require individual projects from combo.
  - I vote for one big repo, see discussion *Single combo repo vs. separate project repos*
  - We can use github's svn bridge to add combo projects to svn meta projects
  - Later, we can use gitman to manage dependencies

# Wishlist Alex Olivas iv

  - The other meta projects should not live in combo, because more maintenance would then be needed in preparation of a release, because we would add to the dependencies. For a combo release, the whole combo code needs to be in a good state.

- Some projects want to stay with svn
  - Meta projects and other projects can stay in svn, as long as these projects are not in combo
  - For combo projects, it would be easiest to use the svn bridge if this is really necessary
- really we'll want to provide people with the option to compose meta-projects based on combo. This can be done with svn externals. The only downside is that it's not a clone, so people can't push back and not great for development. But we can live with that.
- Before we start making decisions though we want to make sure we're choosing the optimal solution and have gamed out as many scenarios as we can.
  - Please challenge this proposal by posting scenarios, either in the `##git_migration` slack channel, or in this repo's issues.

# Wishlist Alex Olivas v

- We’ve already run into coordination issues with PROPOSAL, where we missed a critical (to some) bug fix. A bug was discovered about a week ago in PROPOSAL. Turns out the fix was committed to PROPOSAL in GitHub last October. We’ve since had two releases of combo where the bug fix wasn’t synced to combo in svn. I would worry about doing something similar with all of combo if we can avoid it.
    - We move combo to github
    - Meta projects use github’s svn bridge – no redundancy there
    - People could use github’s svn bridge to commit code – no redundancy there
    - For all pull requests, one can see if they are already merged to master
    - With a one liner, one can check whether a certain commit is included in a release, branch, etc.
- Add CI pipeline
    - We can use github actions to test the build, run the tests, and any other arbitrary code
    - But considering our build time, github actions will cost money if the repo is private
    - This can be circumvented by moving the actions into a public repo, but this will drop some niceties like having tests for pull requests

# Discussion: Single combo repo vs. separate project repos

- A single repo is easier to grasp for newcomers. One single repo, one central README as entry point. Get all necessary code with one single git clone.
- The github ui makes it easy to filter by project. Projects get their own rendered READMEs, their own histories, etc.
- Meta projects can pull releases of individual projects from single-project repos as well as from a ”big” combo repo alike
- We can merge single-project git repos later into combo, preserving both histories
- During the 2019-10-31 software call, it was argued that the optimal solution would be to manage dependencies using a package-management tool. Gitman is a simple python tool freeze dependencies based on release tags or commit ids. It supports both git and svn. But this cannot be used to manage inter-project dependencies if each project has its own repo (and its own gitman definition).

## Discussion: The code was running at 23 October

- This discussion arose in the 2019-10-31 software call from the scenario: ”I checked out the code on 23 October. Then it worked. Now it doesn’t. Why?”
- In git, by design, there is no answer to the question: What was the code on 23 October. Because several people can have their own branches at this time.
- Several people were hoping that enforcing a linear history would enable us to answer the question. But if I create commits locally, push them together next week, nobody has pushed in the meantime, i.e. fast-forward, then I will have changed the answer to the question: What was the code on 2 Nov.
- I’ve never seen something like `git log --show-push-date`, because I think, this is not well defined in git, because git is distributed, i.e. there could be several truths to the answer. I could push the code to several remotes at different times.
- One solution would be to rewrite history on push such that I have the commit date be the push date. But I really don’t like history rewrites, because they change my commit ids.
- We should log the commit id and the current date time to a log file when executing the `env-shell.sh` and icetray. This way, we could ask people that need support to grep that log file (`grep 2019-10-23 /var/log/icetray.log`) and give us the commit id from the output.

# Discussion: Changing history

- During the 2019-10-31 software call, it has been suggested that history rewrites might provide solutions to some of the open issues.
- Personally, I log my commit id in my lab journals. This way, I can reproduce results later.
- Also, when I work with somebody else on a feature branch, we need to agree on commit ids. When one of us changes commit ids, then our shared history is broken.
- When changing commit ids of commits already pushed, someone else could have fetched those (outdated) commits and based his own work on these. When merging his code later, this would re-introduce the outdated commits (or force to do a rebase, which then requires to change commit ids, again)
- Knowing a commit id of some point in development, gives us the information what commits the developer already during the development. When rebasing, this information is lost.
- I would vote for: Never change commit ids of commits that have already been pushed.
- This requires us to allow (non-linear) graph histories.
- Non-linear histories are harder to read
- Non-linear histories reflect the truth of what actually happened during development

# Discussion: Running checks for pushes to master

- Github allows checks for pull requests (author email, big files, ...)
- Github only allows checks for pull requests. For rejecting pushes to master based on these checks, we would need gitlab or github enterprise.
- We still can run these checks after pushes with github actions.
- We can use this to send warnings via email.
- We can even reset the master and move the commits into a feature branch using github actions. But there is a time delay of several seconds.