How did moving to git go?

Outline

- Our workflow
- The good
- The bad and the ugly
- Lessons learned

Our workflow: gitflow

- Development happens in feature/bug branches.
- Feature branches are merged into one or more of the master, release, and development branches.
- … so branching and merging happens all the time.

The bad and the ugly: branching

- People were confused by gitflow and didn't know which branch to edit.
- People got confused about which branches they were in, so they'd end up trying to push changes to the wrong branch.
- People got confused about the distinction between local branches and remote branches.
  - E.g. -

    ```
    git checkout my-branch
    ```

    makes a local copy of the remote branch called 'my-branch' and changes you to that branch. In contrast,

    ```
    git checkout origin/my-branch
    ```

    checks out a particular commit, but doesn't create a local branch.

The bad and the ugly: the git command line

- People were confused by the git command line. E.g. -
  - Why isn't `git branch` showing all the branches?
    - Answer: git branch only shows you local branches, not remote branches. Use `git branch -a`.
  - Why isn't `git commit` doing anything?
    - Answer - you have to add your changes before committing even if you've previously added the files to git. Use `git commit -a` to add all changed files before committing.

The bad and the ugly: merging

- People weren’t sure which branches should be merged where.
- People weren’t sure what to do about merge conflicts or how to resolve them.

The good: new tools

- Modern tools often do not support CVS.
- Moving to git allowed us to use these new tools, including in our case the Atlassian tool suite.

The good: less anxiety over commits

- In CVS, as soon as you commit, it shows up in the central repository and can cause other people headaches.
- In git, you can commit locally as much as you want without anyone else seeing it. Once you are done with a feature, you can push your changes so others can see them.

The good: code reviews

- When you have a branch for every feature, there's an obvious place to have a code review - namely, when that feature is merged into your main working branch.
- Hooks let you put code reviews in as requirements for merging.

The good: cheap new repos

- Repositories are cheap, easy, and require almost no resources: `git init`
- You can make as many repositories as you want and delete them if you no longer need them.

Other good stuff we haven't really used

- Forking:
  - Want to modify someone else's repository? Getting an editable copy is easy: `git clone`
  - If you've made a useful change, you can ask for commit/merge privileges into the parent repository.
  - Alternatively, you can keep your forked repository and periodically merge feature changes from the parent repository.
- Commit without network access:
  - You don't have to access the central repository every time you commit, only when you push or fetch.
  - So, you only need access to the machine you are actually working on.

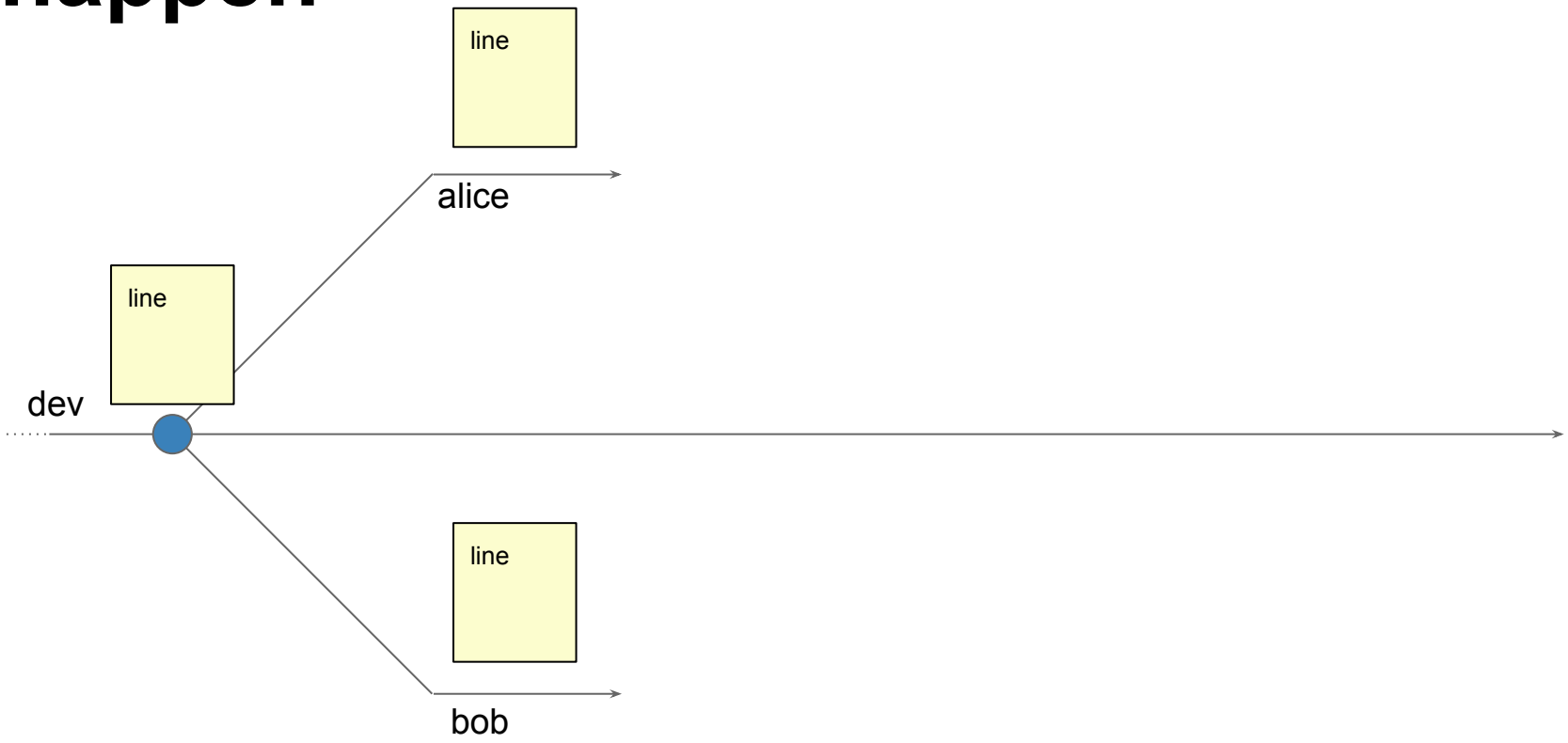
Lessons learned

- Repository controls are great. They stop people from accidentally doing the wrong thing.
- Learning the new system takes some time.
- Merge conflicts can be very confusing if you haven’t been doing a lot of branching. Taking a moment to walk people through why and how they happen is helpful.

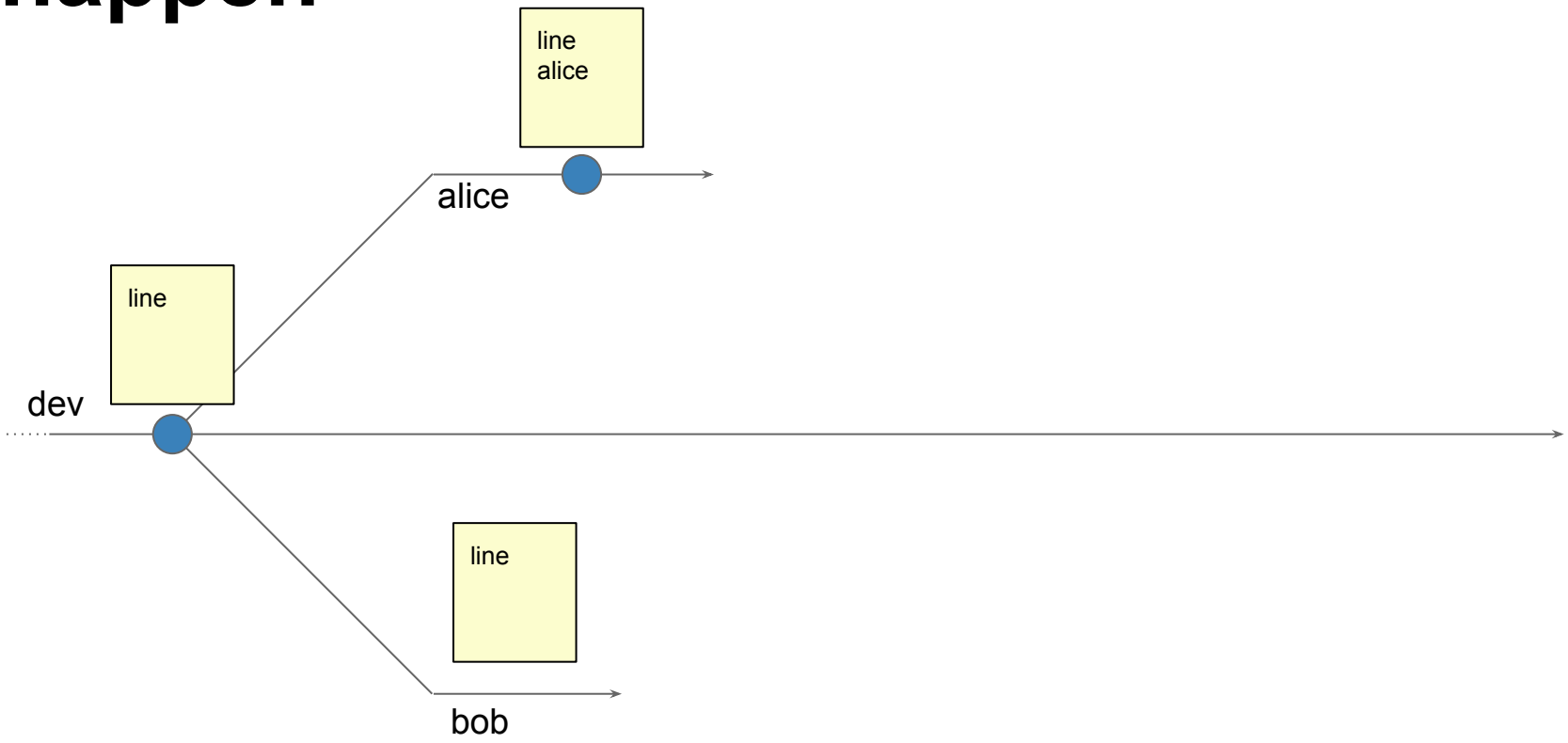
On that note...

How and why merge conflicts happen

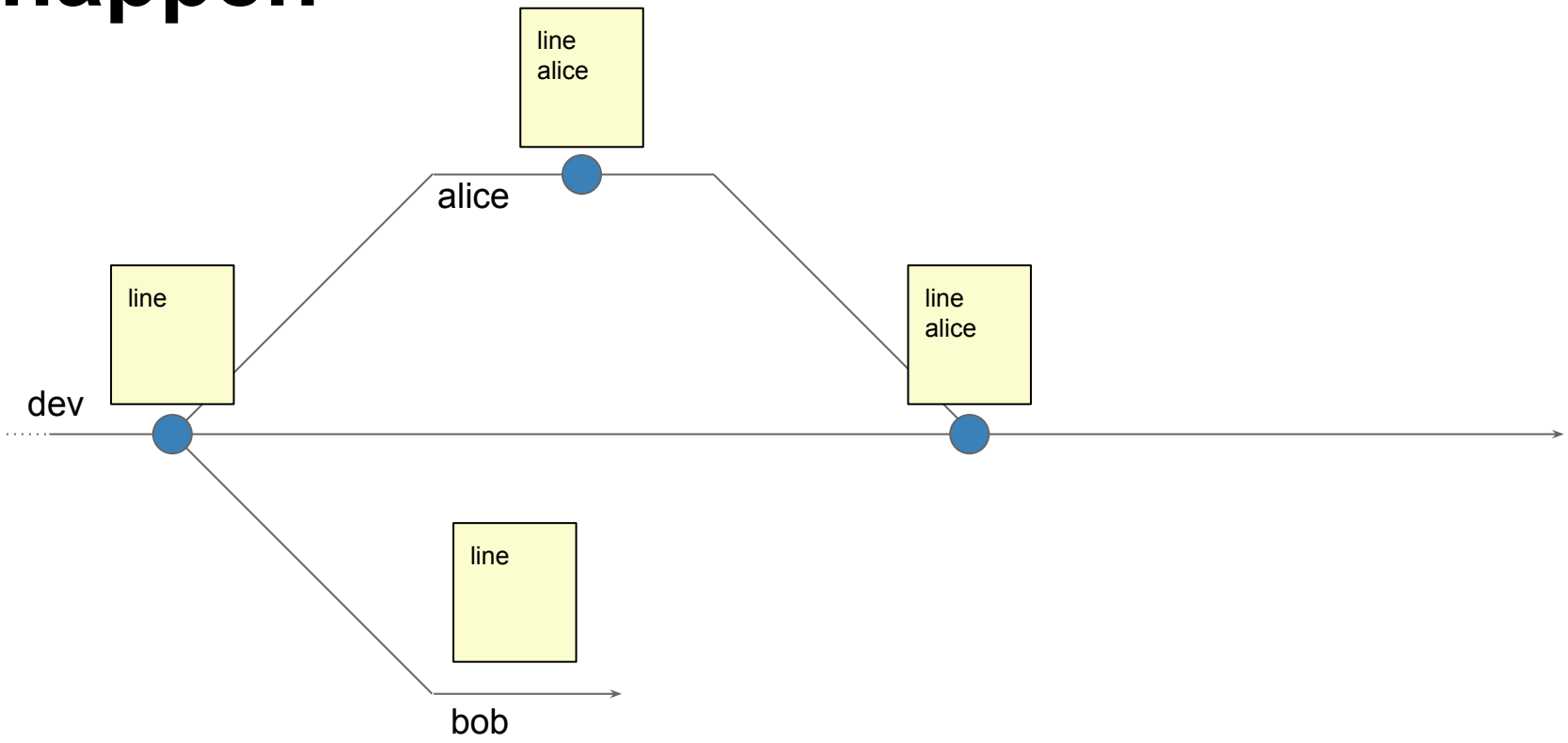
How and why merge conflicts happen



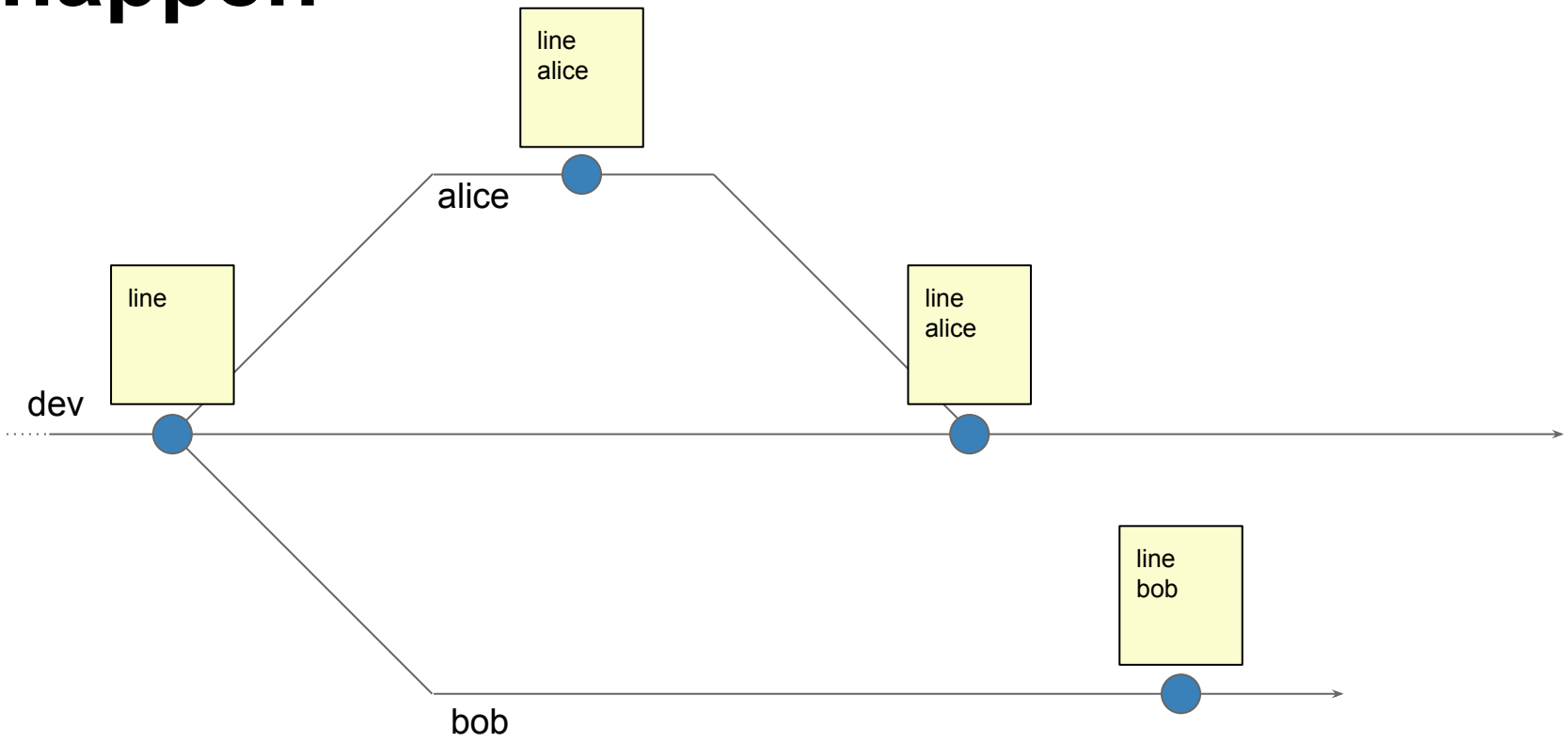
How and why merge conflicts happen



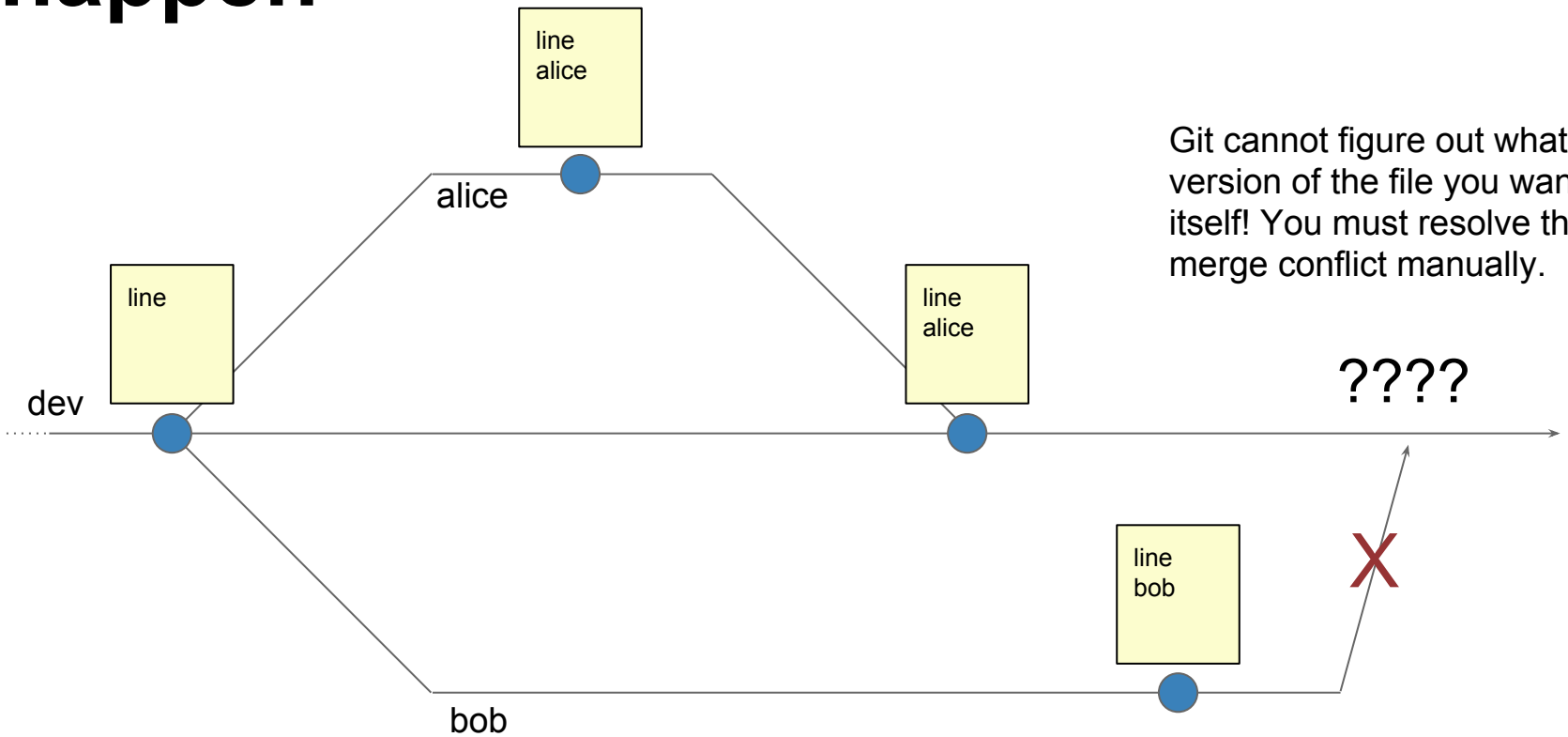
How and why merge conflicts happen



How and why merge conflicts happen

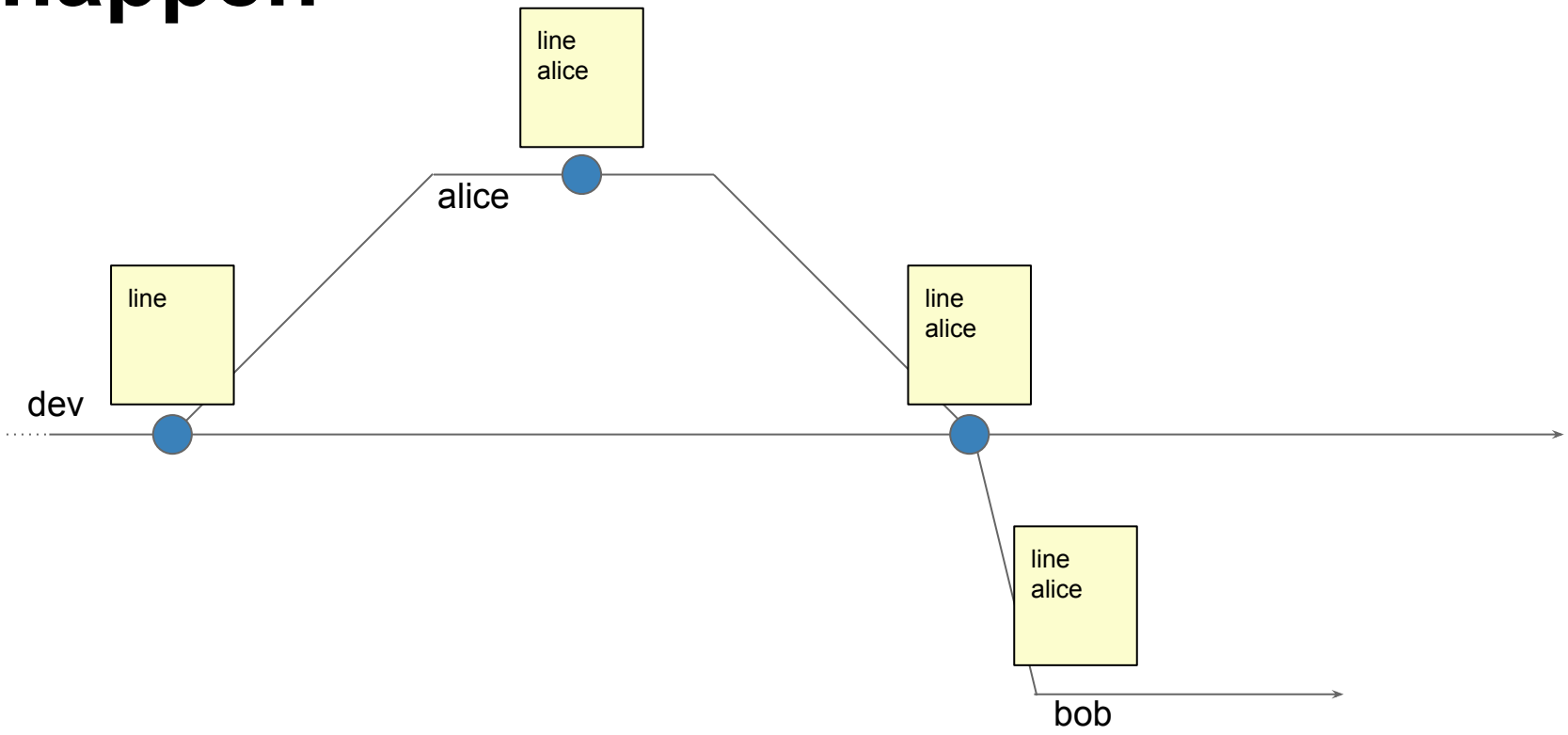


How and why merge conflicts happen

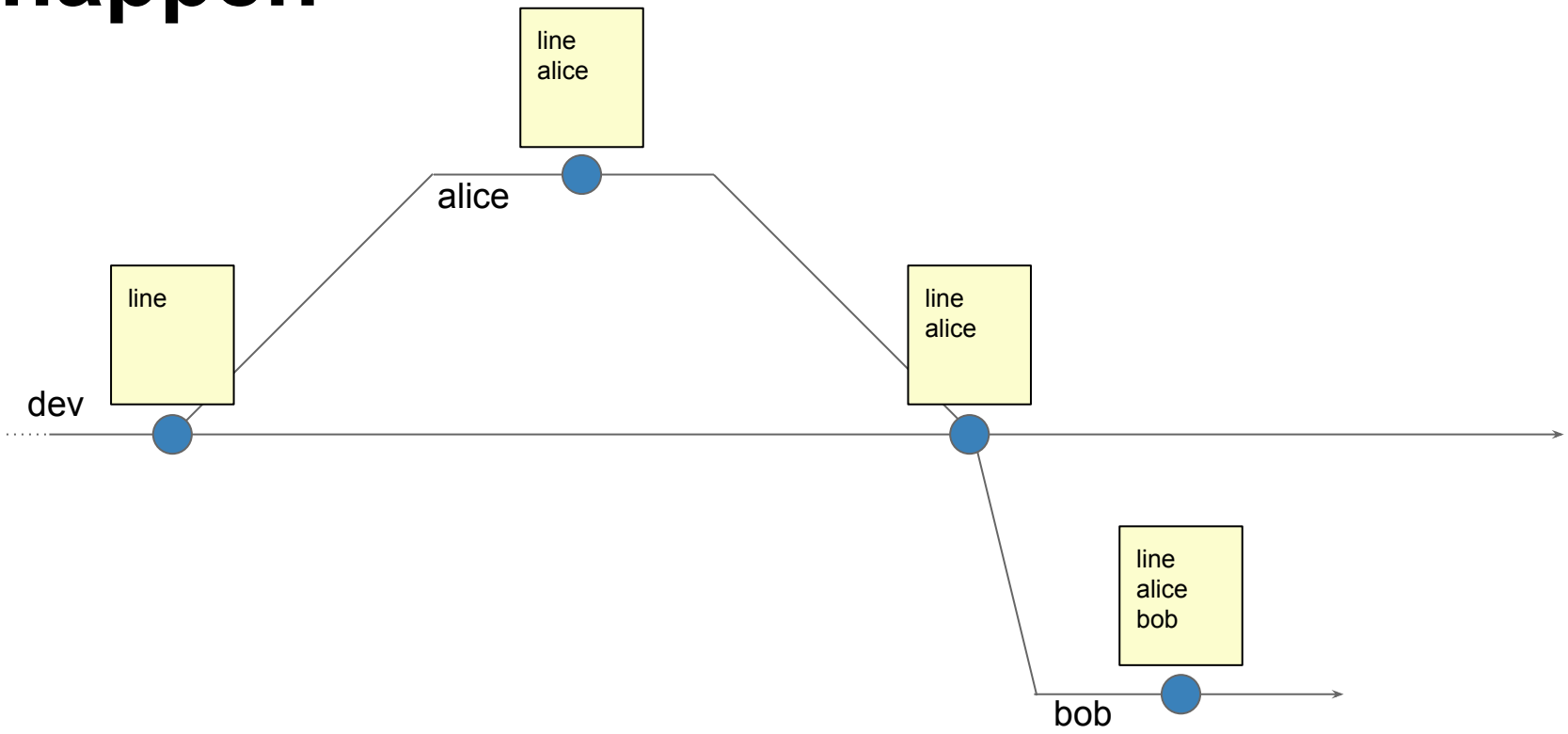


Git cannot figure out what version of the file you want by itself! You must resolve the merge conflict manually.

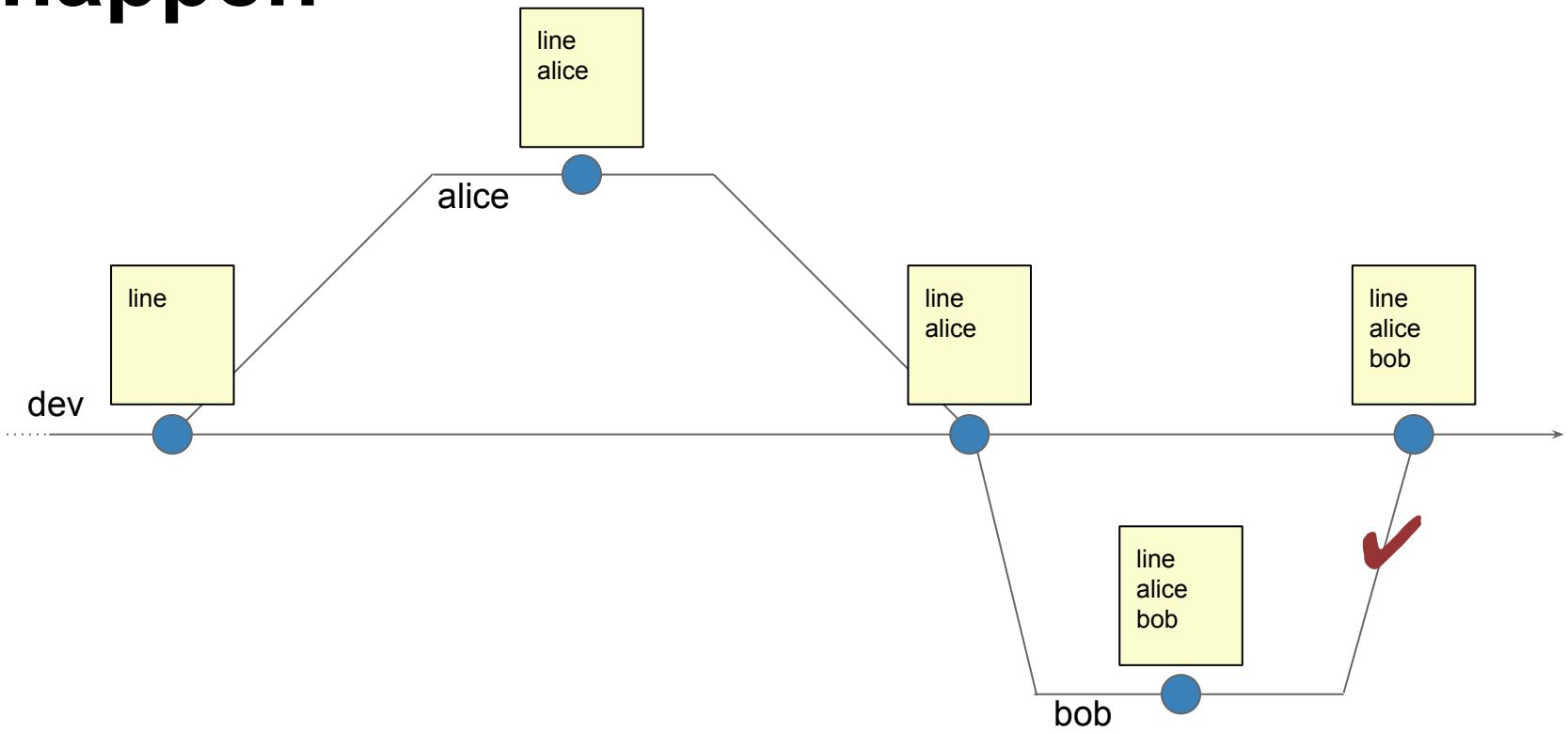
How and why merge conflicts happen



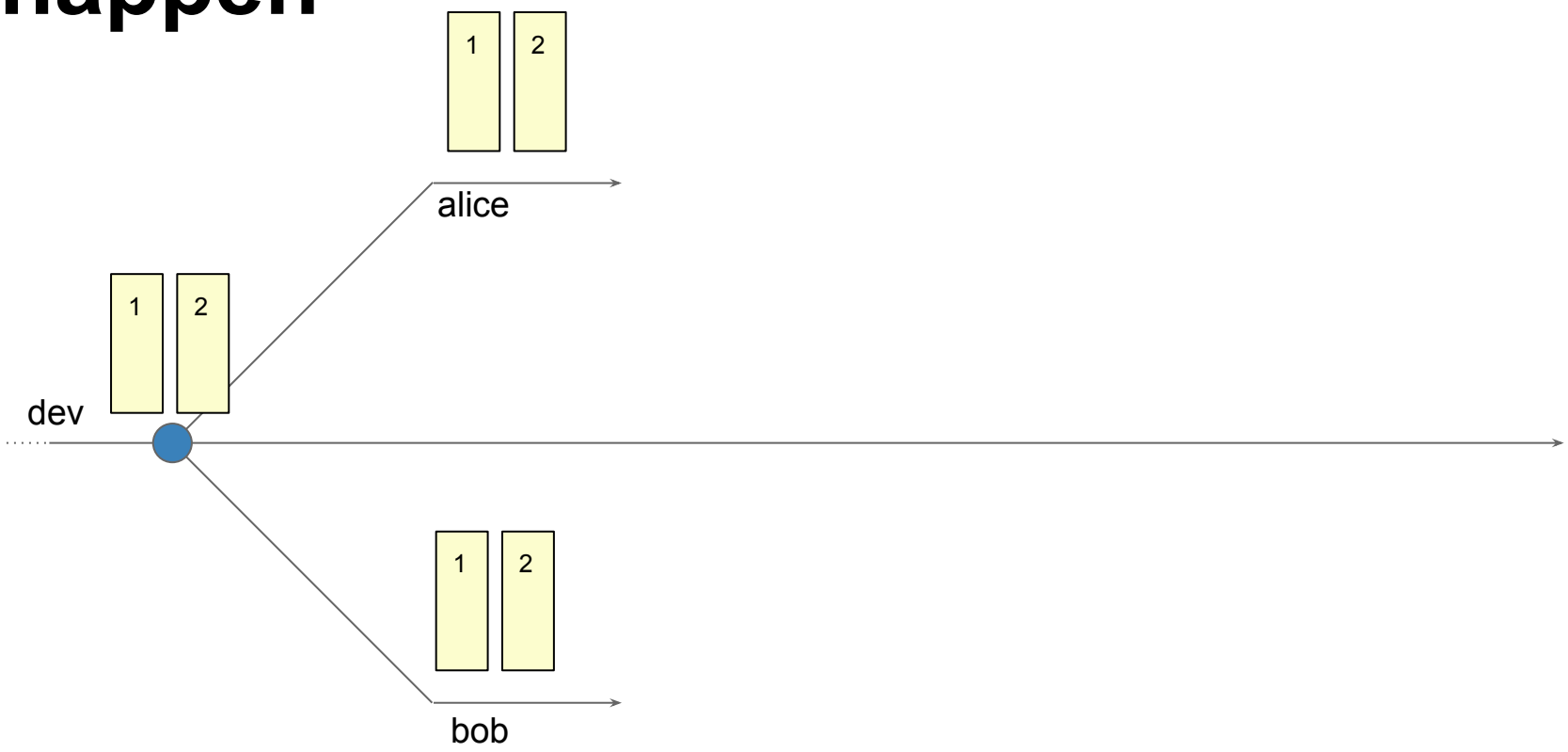
How and why merge conflicts happen



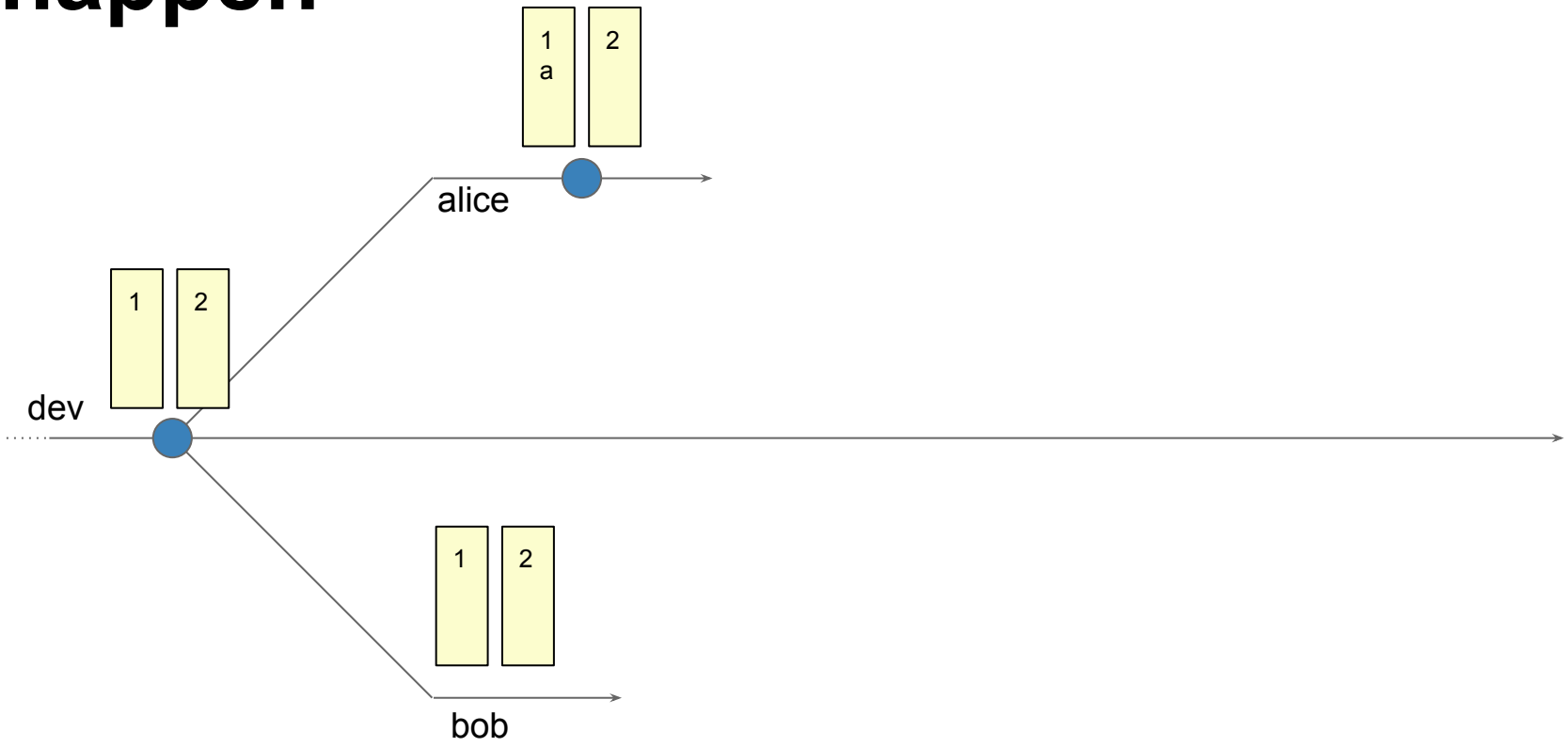
How and why merge conflicts happen



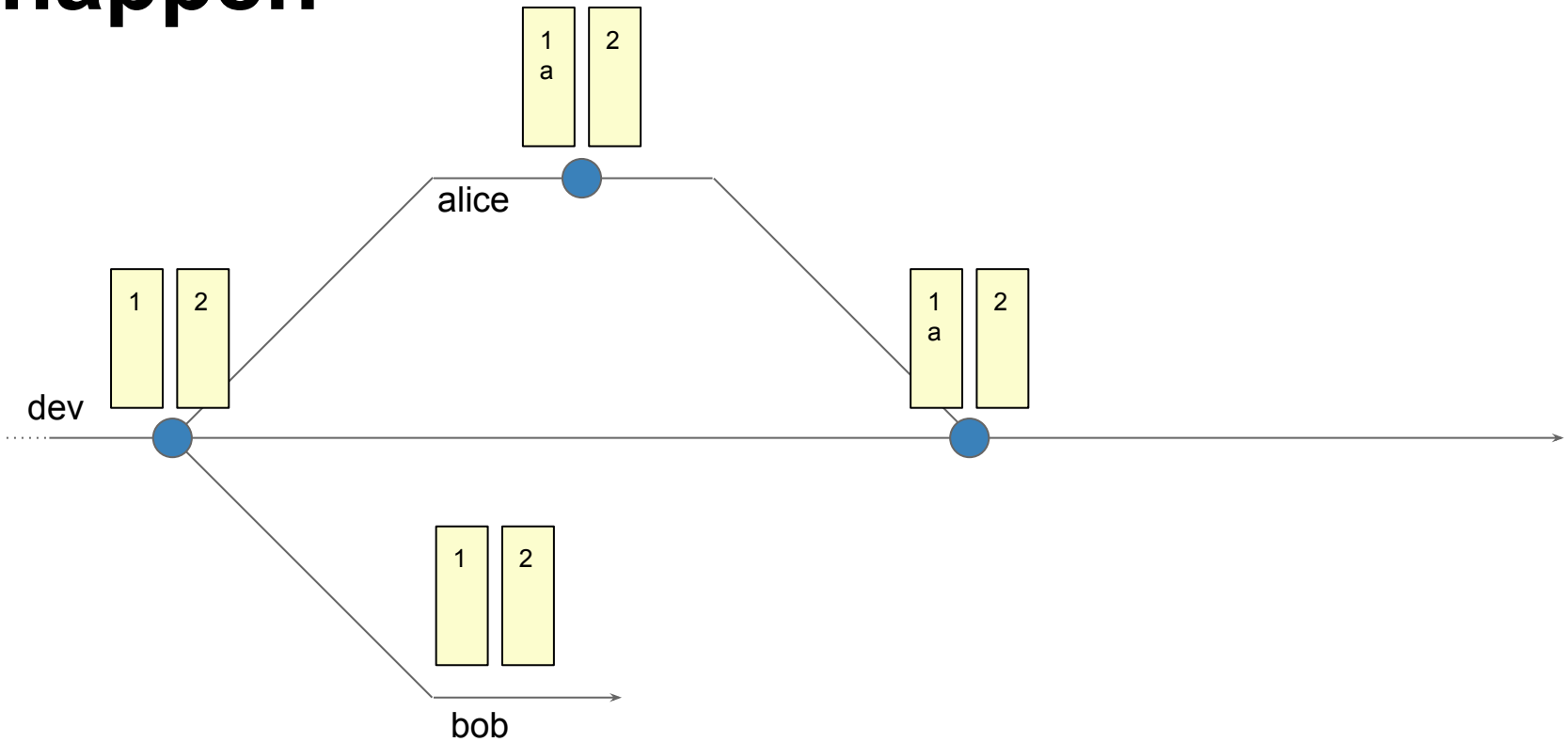
How and why merge conflicts happen



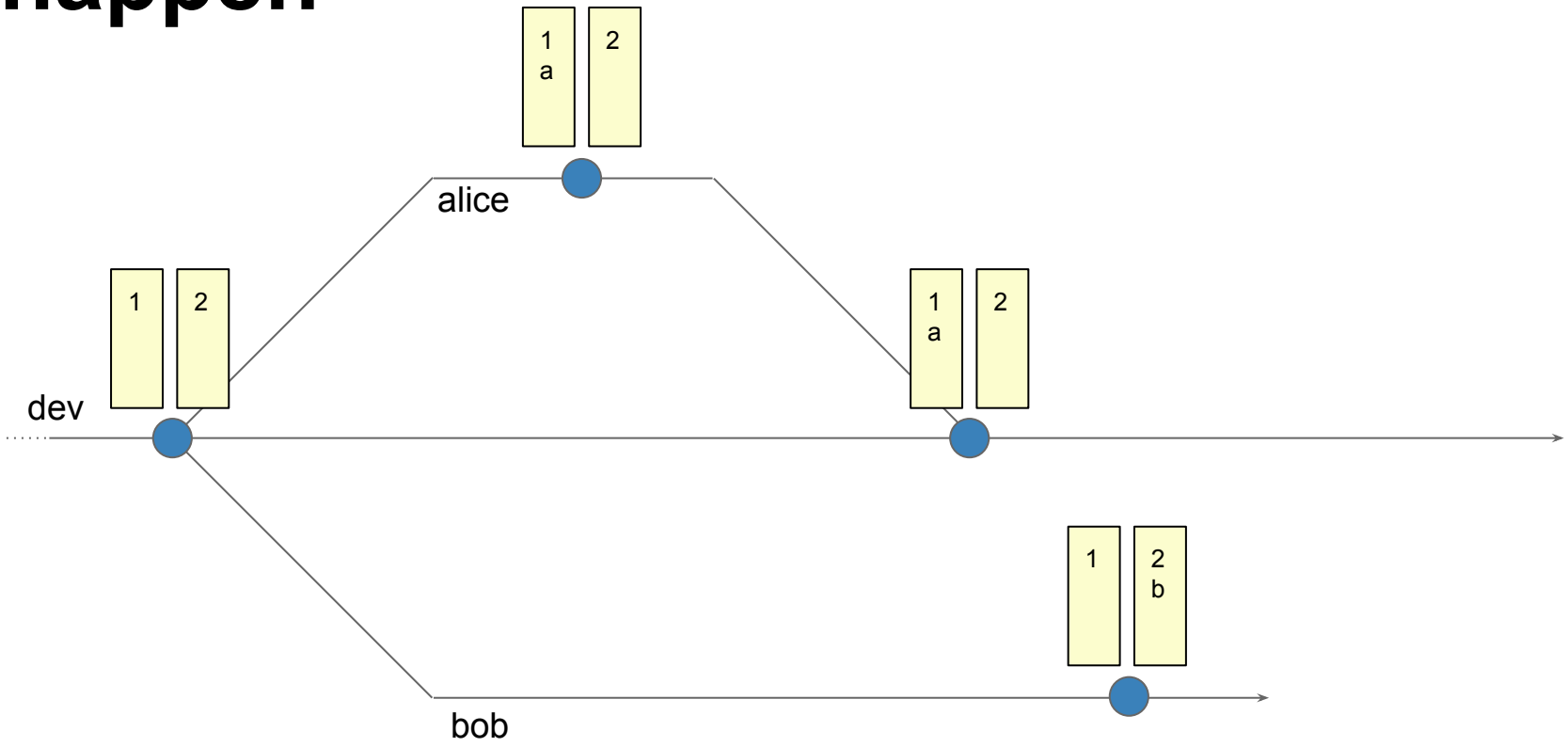
How and why merge conflicts happen



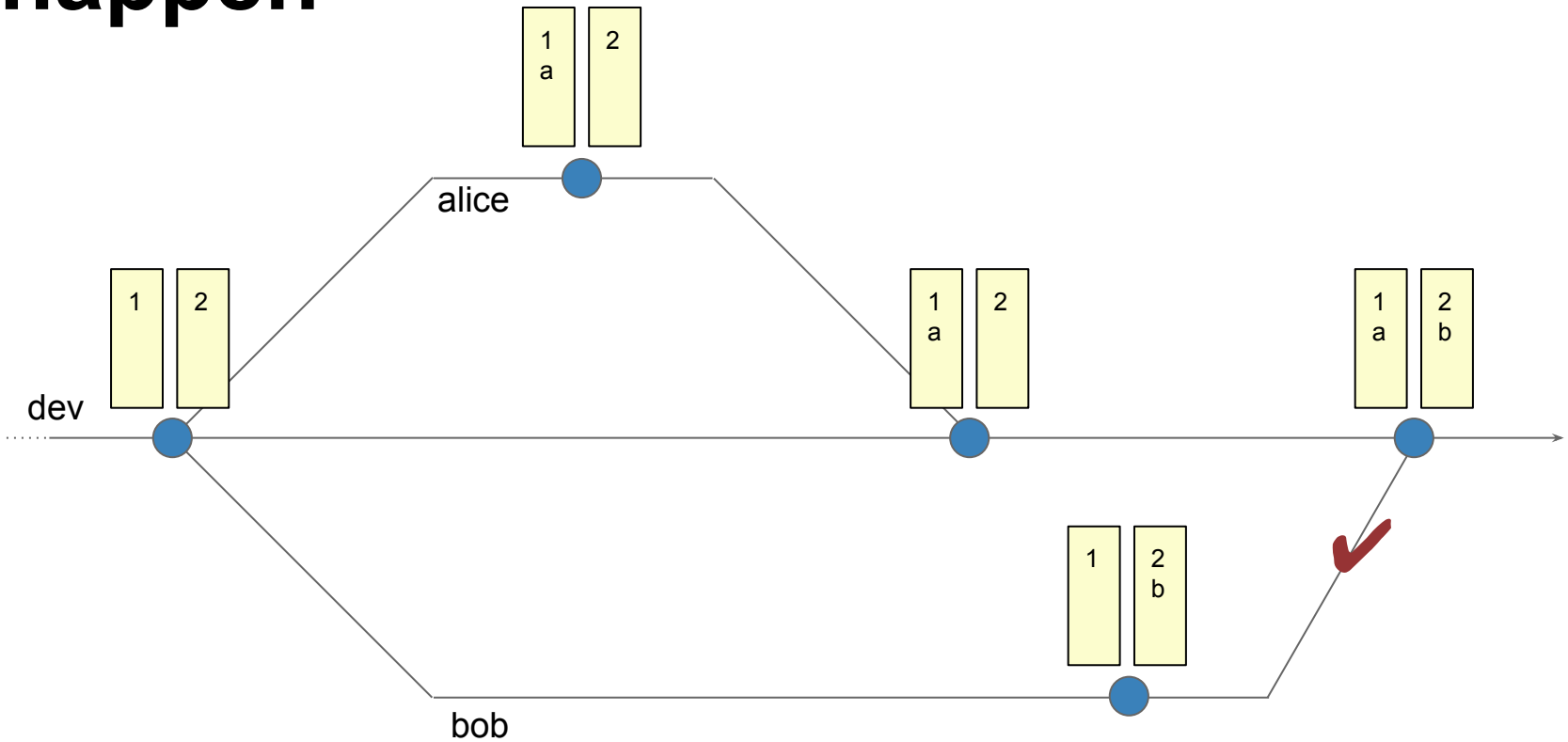
How and why merge conflicts happen



How and why merge conflicts happen



How and why merge conflicts happen



Questions?