Remote collaborative work

A practical introduction to `git`

by Guillaume Desforges

Data & Software Engineer @ Tweag

Goals

- understand why it's important to use the right tools
- learn `git`'s model of collaboration
- master its most basic workflows

Why?

This year (and the next)

Work on text files

- code
- reports: TPs, projects, internships

What's the matter?

- text content evolves with time
- real work requires collaboration

Tracking changes: versioning

- cancel changes (rollback)
- work on different versions in parallel
- understand when and why something was changed

Working together: collaboration

- make some modifications on one's own computer
- share their modification
- integrate others' modifications
- store all those modifications in one central place

How to do all that?

- rollback to a previous version
- work on different versions in parallel
- share/get modifications with/from others

The solution?

`git` versioning model

"How to version with `git`"

Reminder: goals

- rollback to a previous **version**
- work on different **versions** in parallel
- share/get **modifications** with/from others

What's a version?

A version represents evolving content

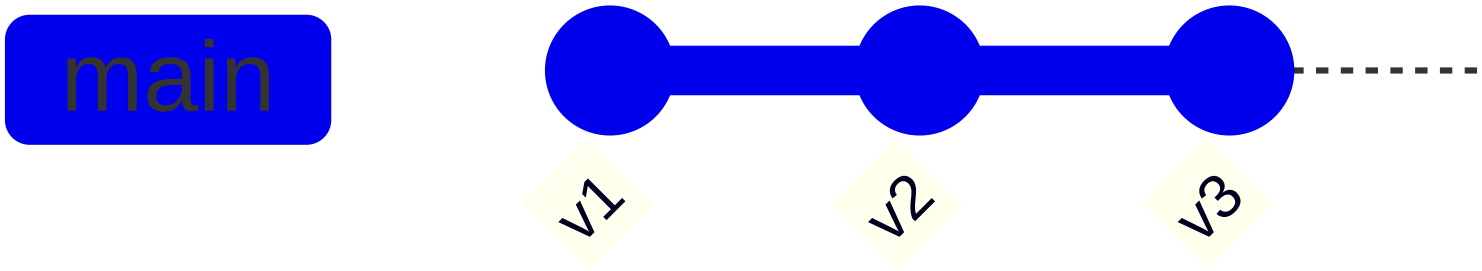
Version `v1`

```
Hello world.
```

Version `v2`

```
Hello, world!
```

Visualizing versions



Version `v1` is followed by `v2`.
Version `v2` is followed by `v3`.

A new version is "modifications from past version"

```
- Hello world.
+ Hello, world!
```

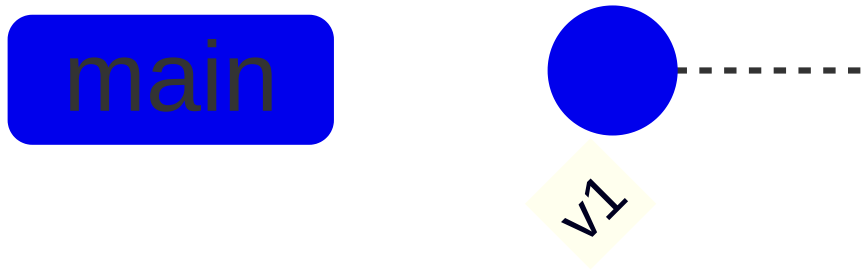
We call it a **diff**.

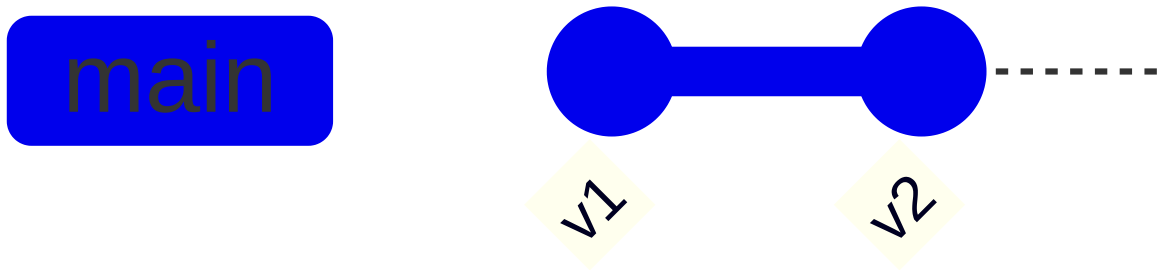
What's a version?

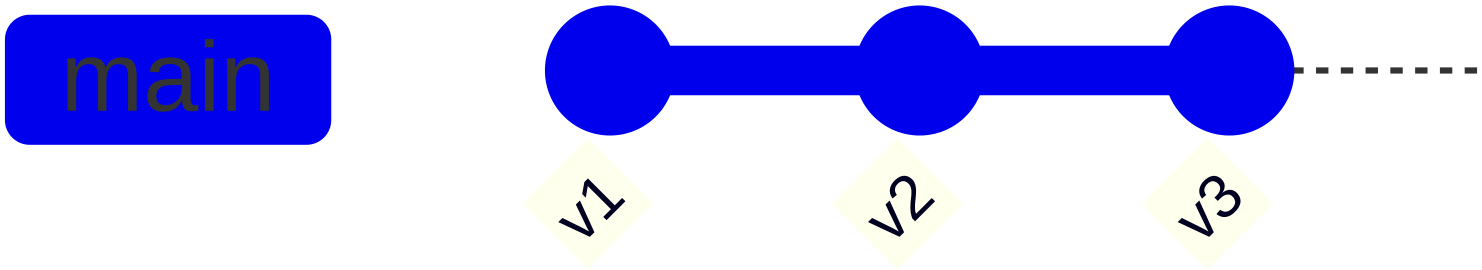
- diff
- id of parent
- id (hash)
- additional data (date, author, description)

This is a "commit"

commit		
diff		
hash		
parent		
date	author	description


main
v1


main
v1
v2


main
v1
v2
v3

How to create a commit

- using `git`
  - it's a command-line interface (CLI)

    ```
    git <command>
    ```

  - some visual tools exist
- before anything: `git init`

How to compute the diff?

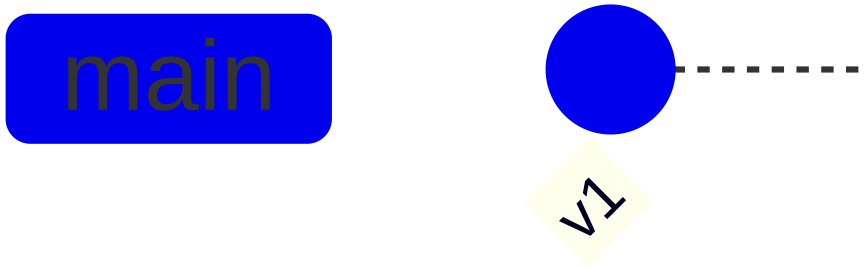
`git diff`

output:

```
diff --git a/song.txt b/hello.txt
index 18249f3..af5626b 100644
--- a/song.txt
+++ b/song.txt
@@ -1 +1 @@
-Hello world.
+Hello, world!
```

How to create a commit

From version `v1`

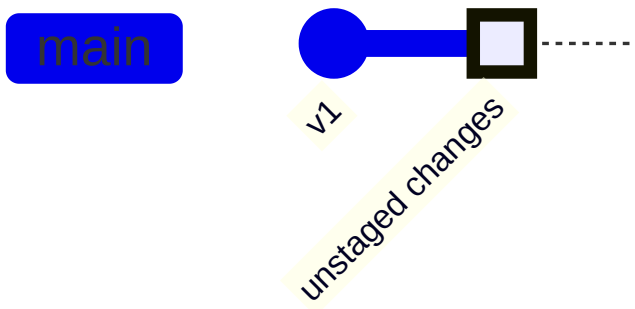


`song.txt`:

```
Hello world.
```

How to create a commit

Edit files



`song.txt`:

```
Hello, world!
```

How to create a commit

Check current status with `git status`

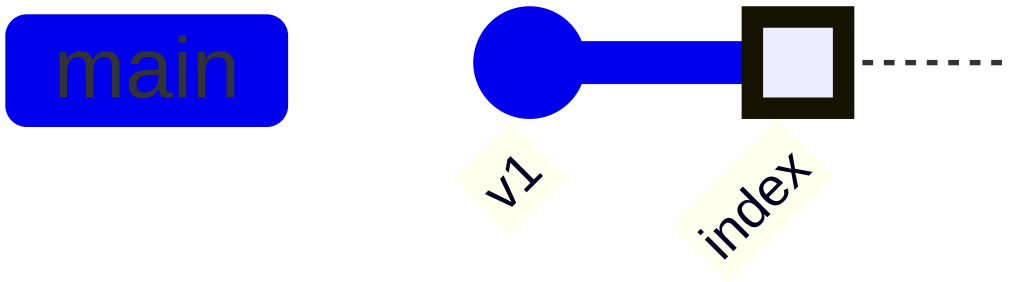
output:

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   song.txt

no changes added to commit (use "git add" and/or "git commit -a")```
```

How to create a commit

Add to index: `git add song.txt`



How to create a commit

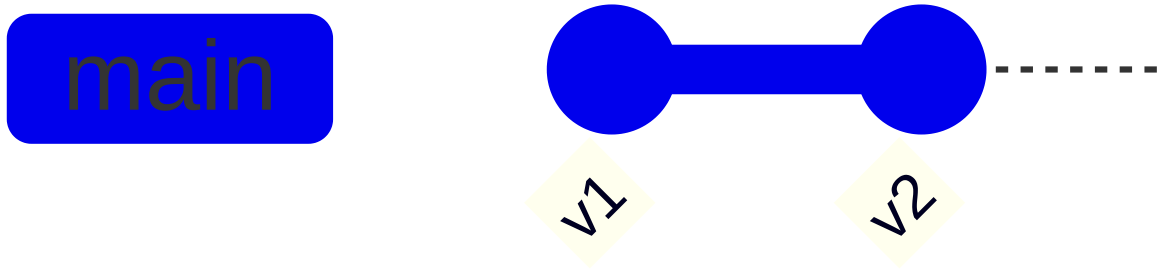
Check current status with `git status`

output:

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   song.txt
```

How to create a commit

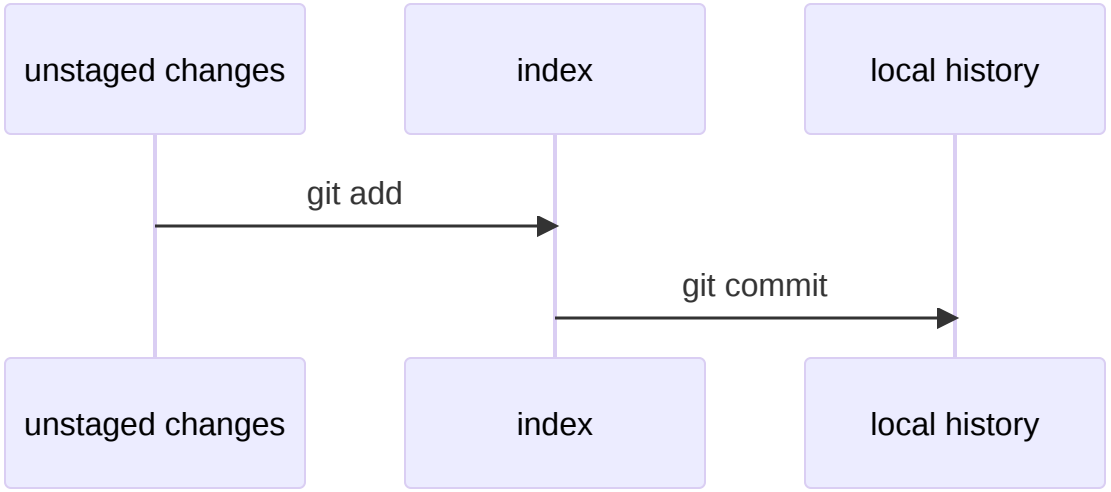
Commit to a new version: `git commit -m "v2"`



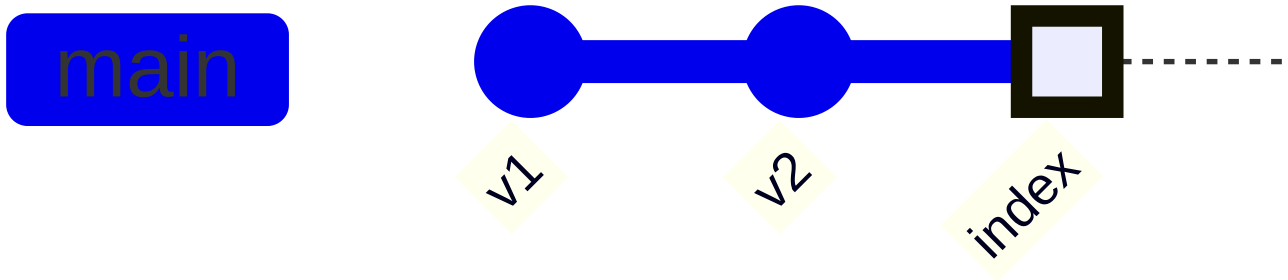
How to create a commit

- `git add song.txt`
- `git commit -m "v2"`
- tip: use `git status` often

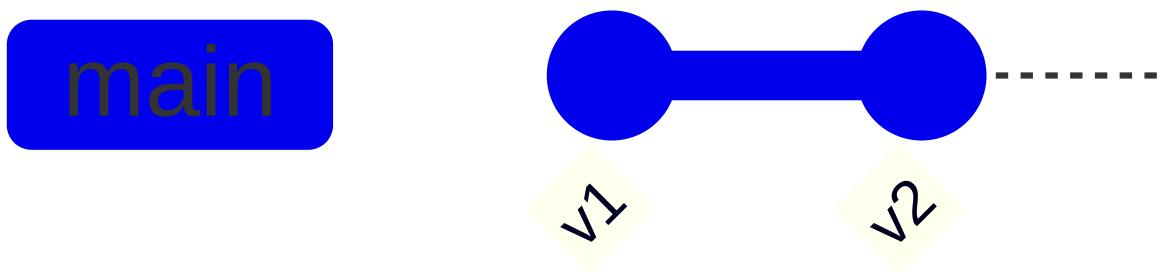
What happened?



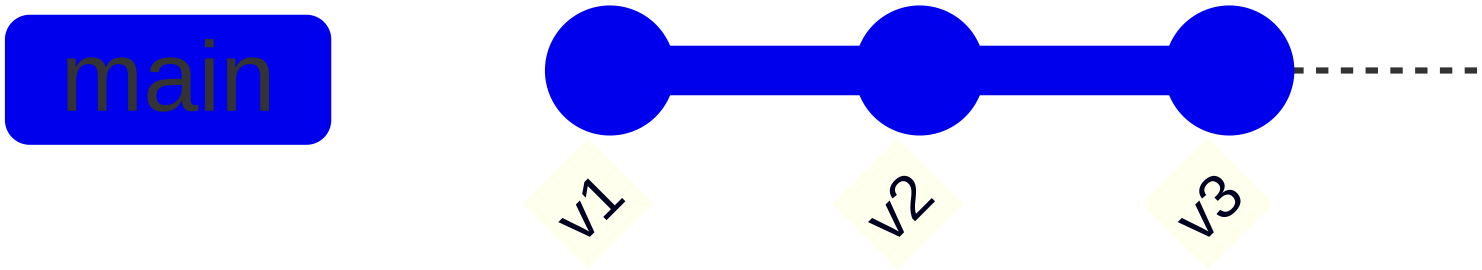
Cancel current changes



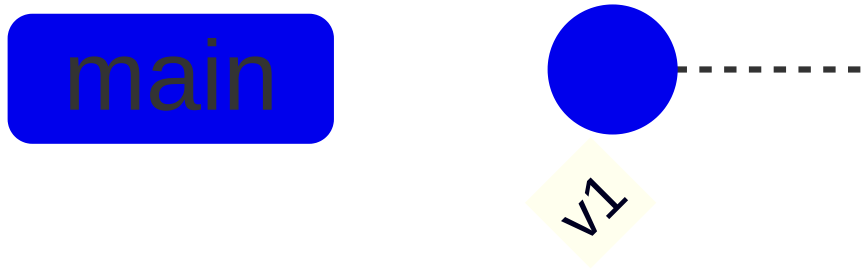

```
git reset
```



Reset to another version

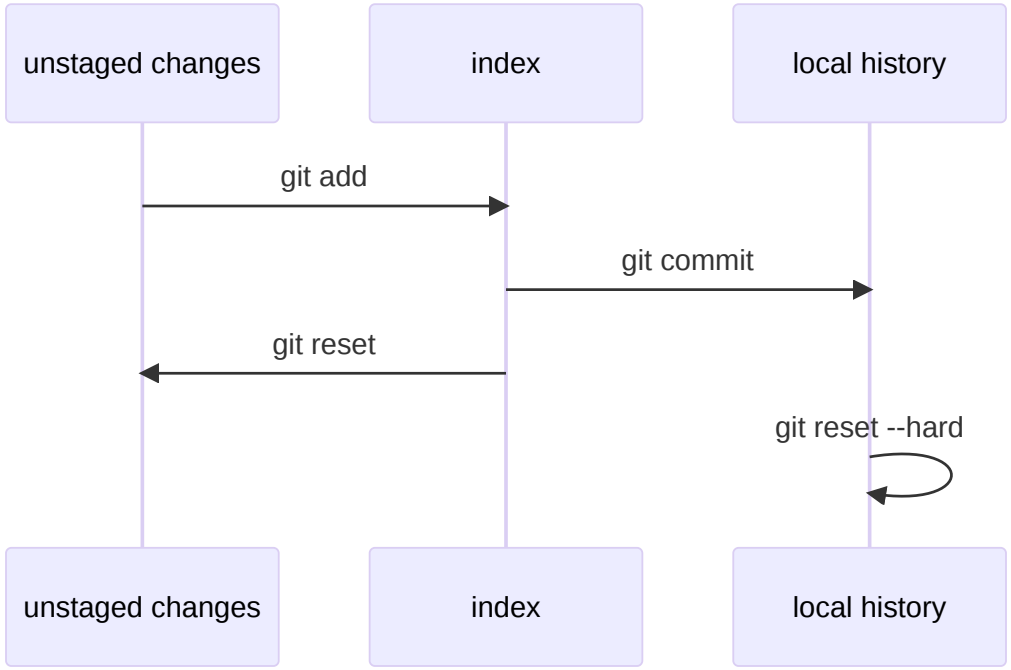



```
git reset --hard v1
```

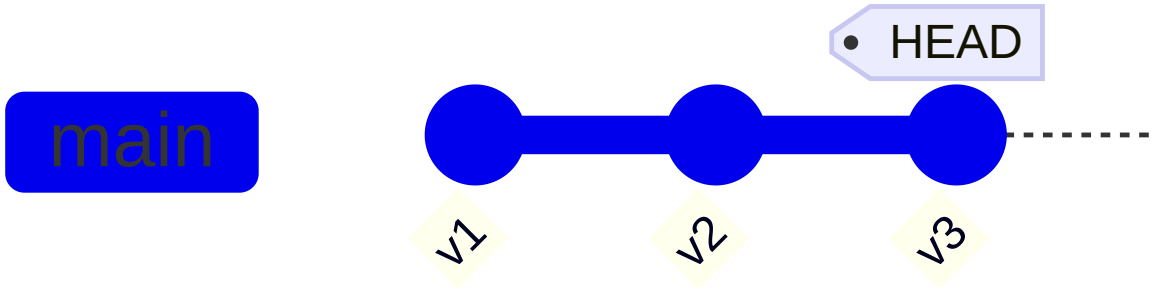


It's like *deleting* commits.

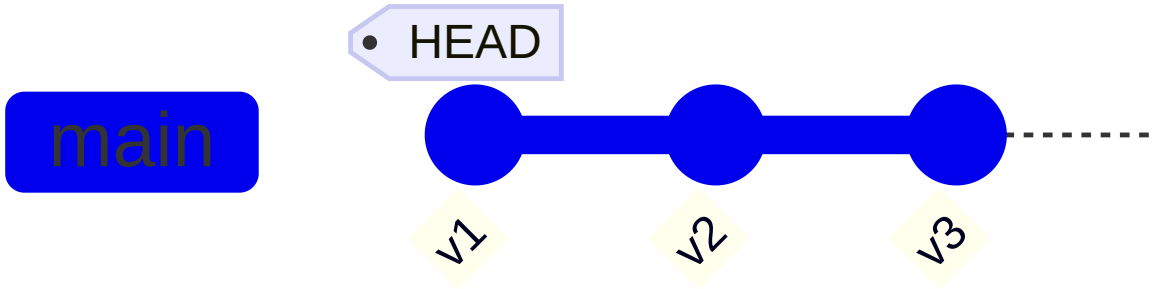
Reset changes



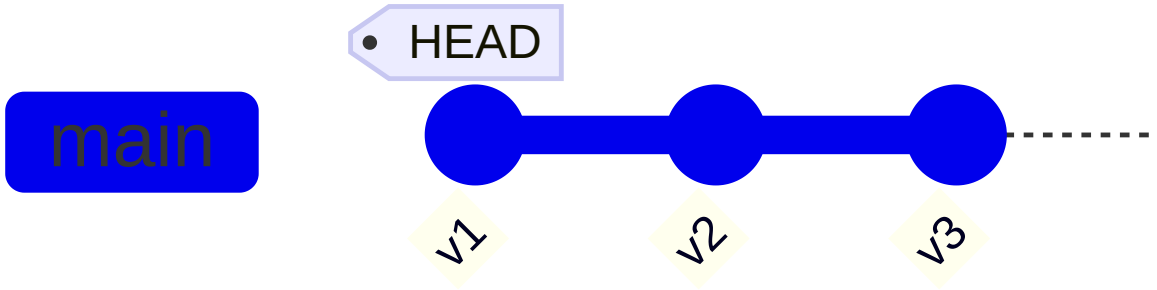
Travel in the history



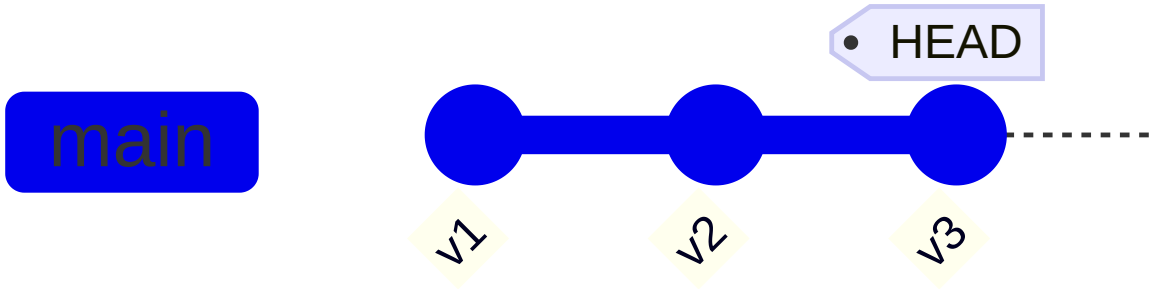

```
git checkout v1
```



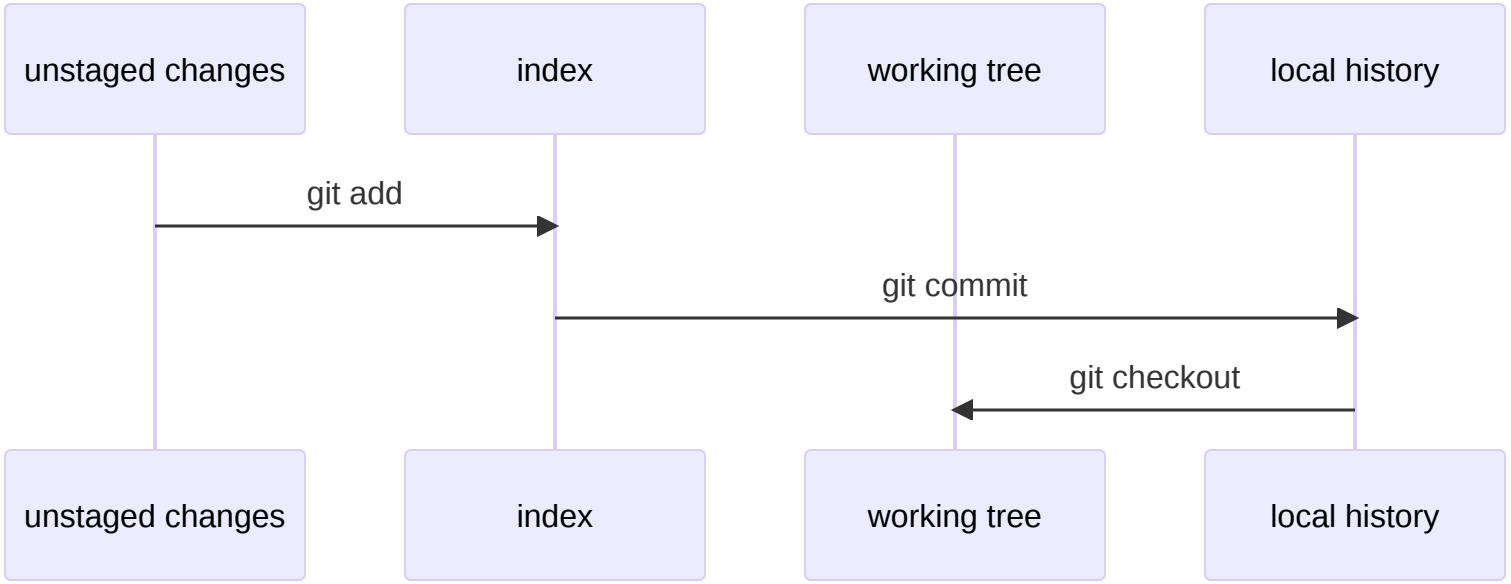
Travel in the history



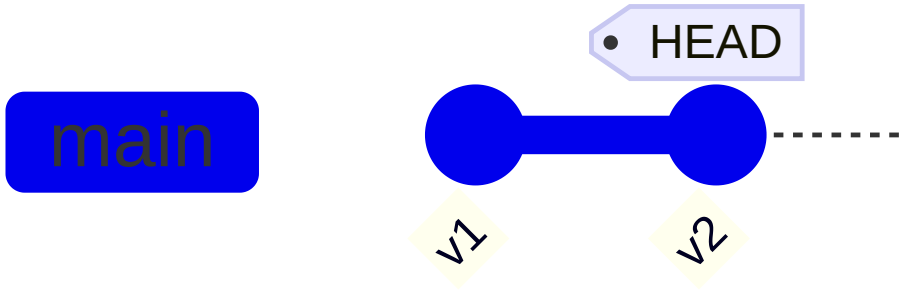

```
git checkout main
```



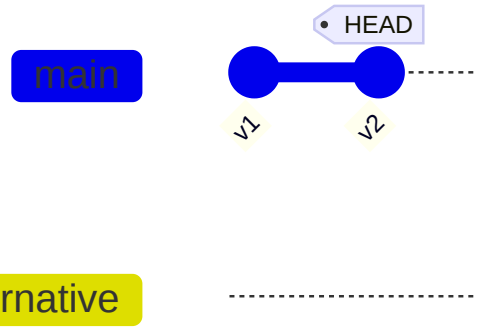
Travel in the history



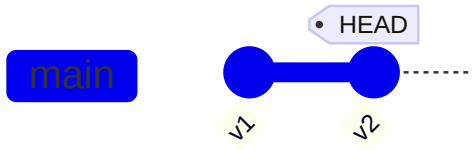
Working in parallel




```
git branch alternative
```

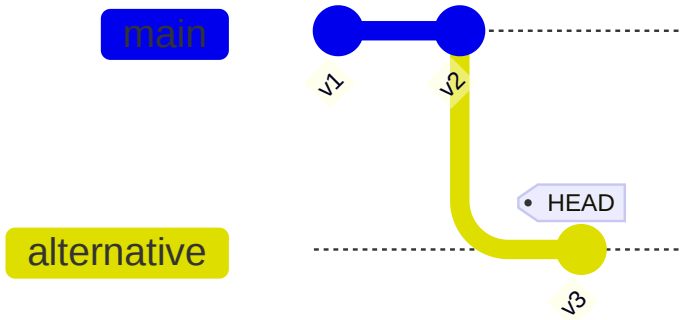


Working in parallel

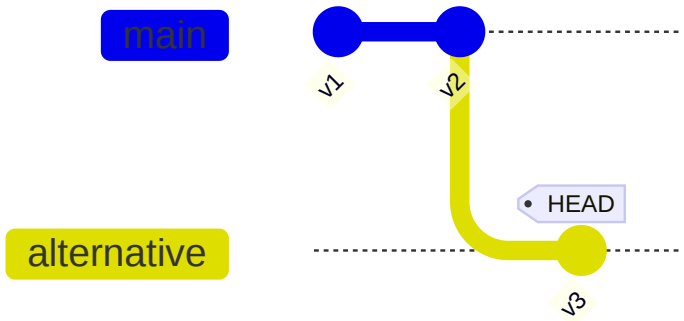



```
git checkout alternative
```

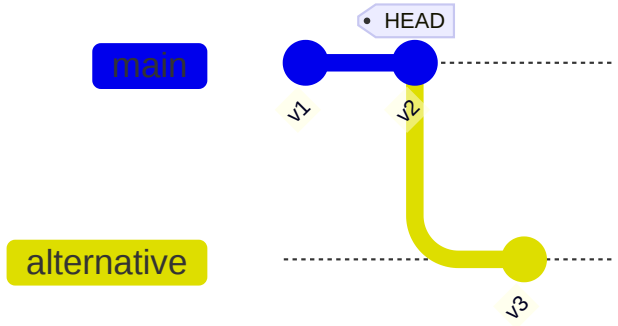
```
git commit -m "v3"
```



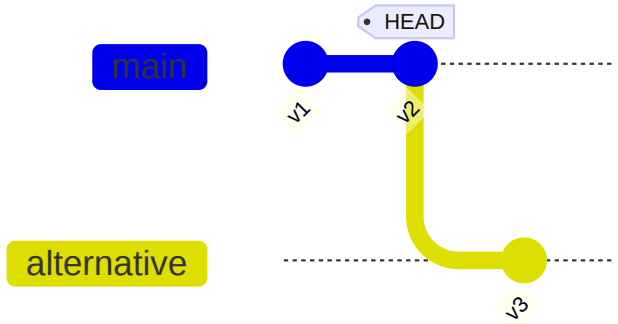
Working in parallel



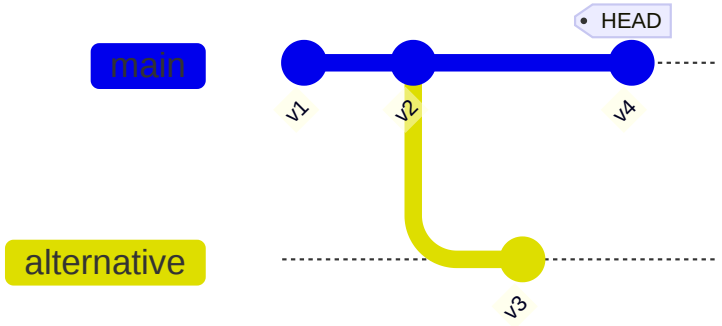

```
git checkout main
```



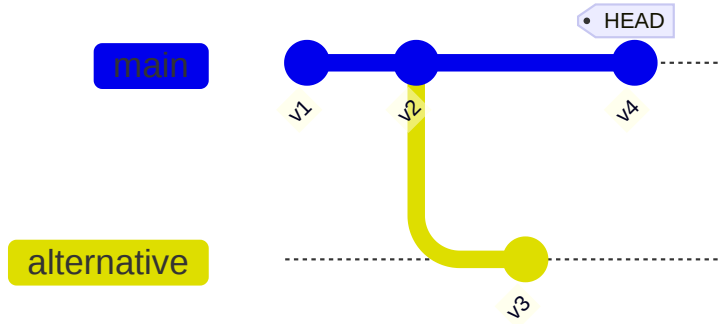
Working in parallel




```
git commit -m "v4"
```

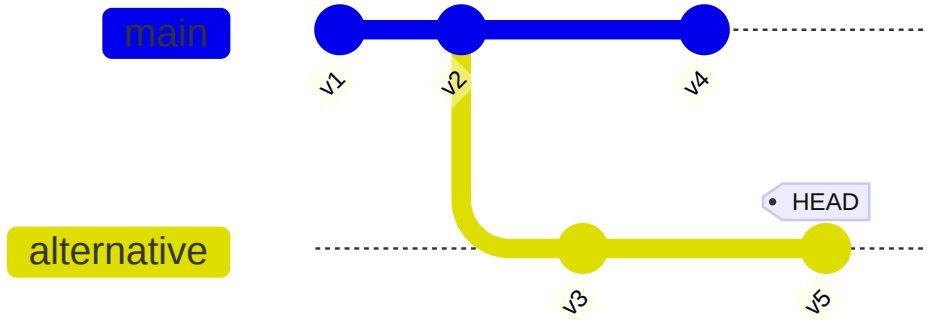


Working in parallel

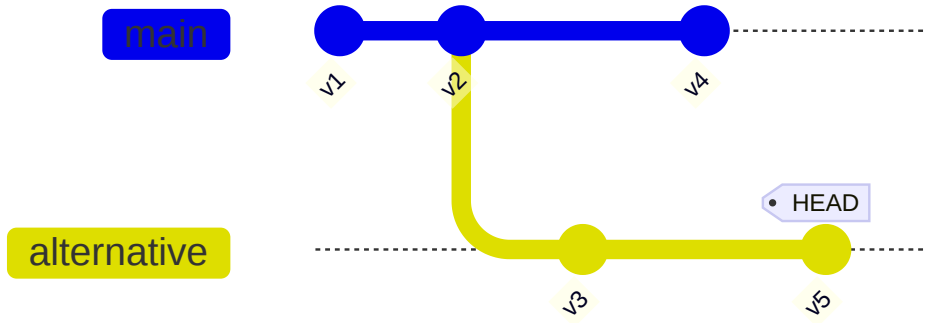



```
git checkout alternative
```

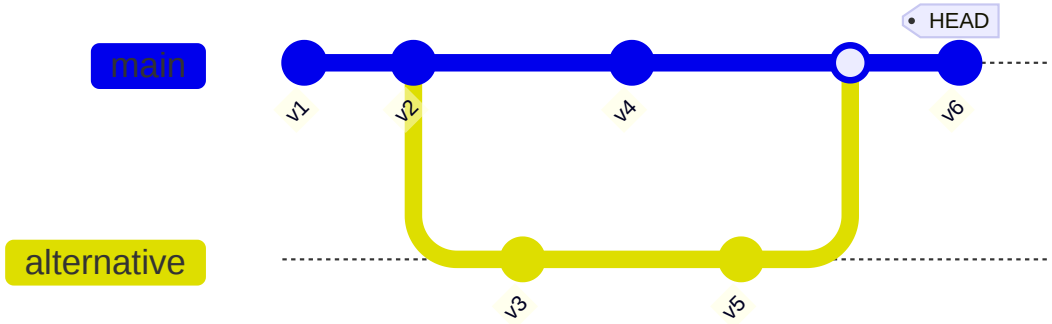
```
git commit -m "v5"
```



Merging branches




```
git checkout main
git merge alternative
```



What if they are not compatible?

From `v1` to `v4`

```
- Hello world.
+ Never gonna give you up
```

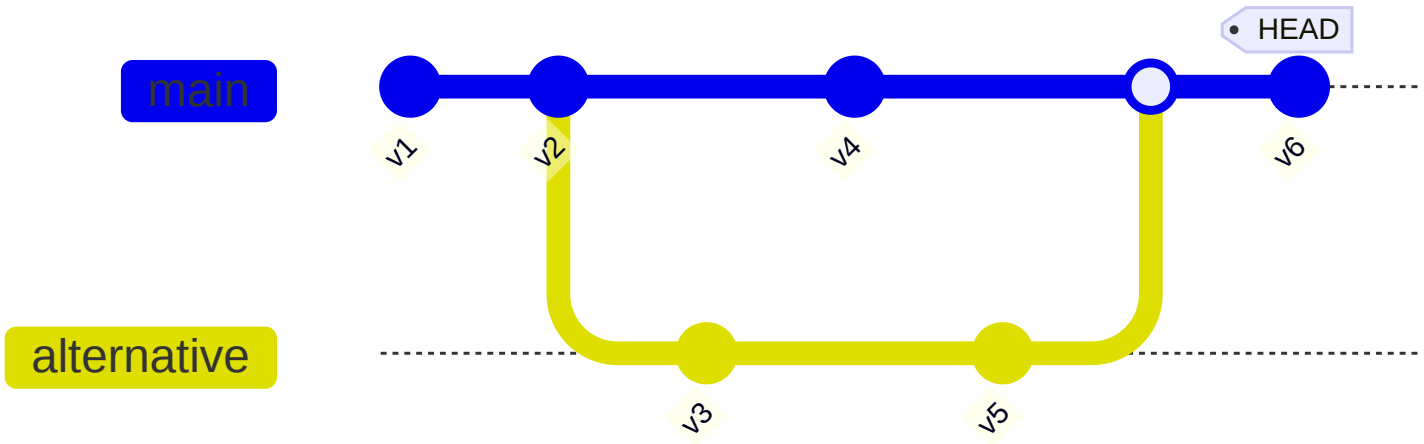
From `v1` to `v5`

```
- Hello world.
+ Never gonna let you down
```

From `v1` to `v6` (merged)

```
- Hello world.
+ Never gonna give you up nor let you down
```

- It is called a **conflict**



What if they are not compatible?

- It is called a **conflict**
- Practical session will cover how to handle it
- `git` was *made* to handle conflicts properly, it's its strength: don't dodge it!

What if they are not compatible?

How it looks like

```
$ git merge alternative
Auto-merging song.txt
CONFLICT (content): Merge conflict in song.txt
Automatic merge failed; fix conflicts and then commit the result.

$ cat song.txt
<<<<<<< HEAD
Never gonna give you up
=======
Never gonna let you down
>>>>>>> alternative
```

- a conflict is purely "textual"
- there is no conflict if diffs are the same on both sides

Recap

- *modifications* of content is a **diff**
- create **commits** to keep track of *versions* progressively
- use **branches** to work on *alternative histories*
- **merge** two branches to *merge both histories* together
- conflicts may happen: just solve them and move forward

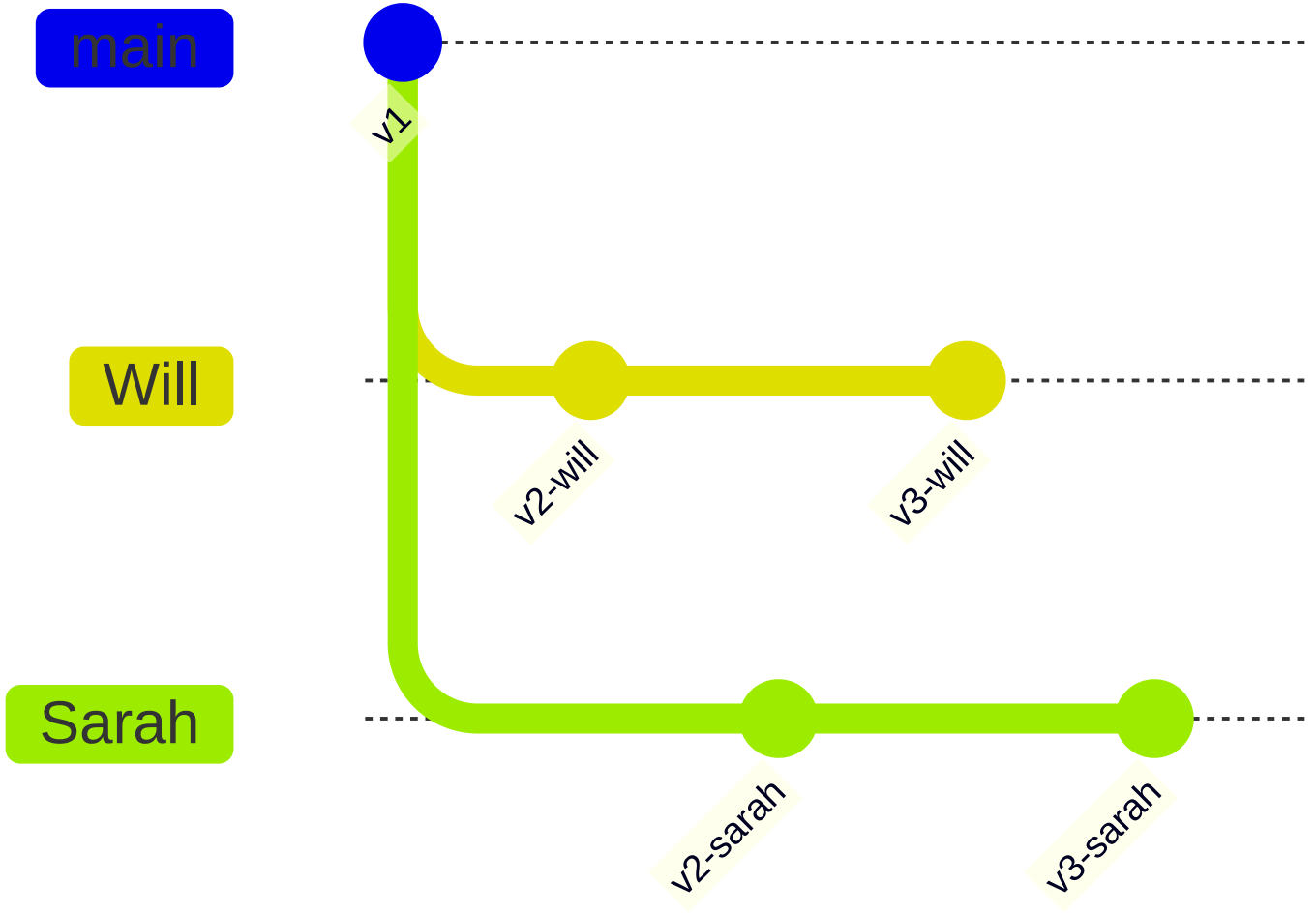
`git` collaboration workflow

"How to collaborate using `git`"

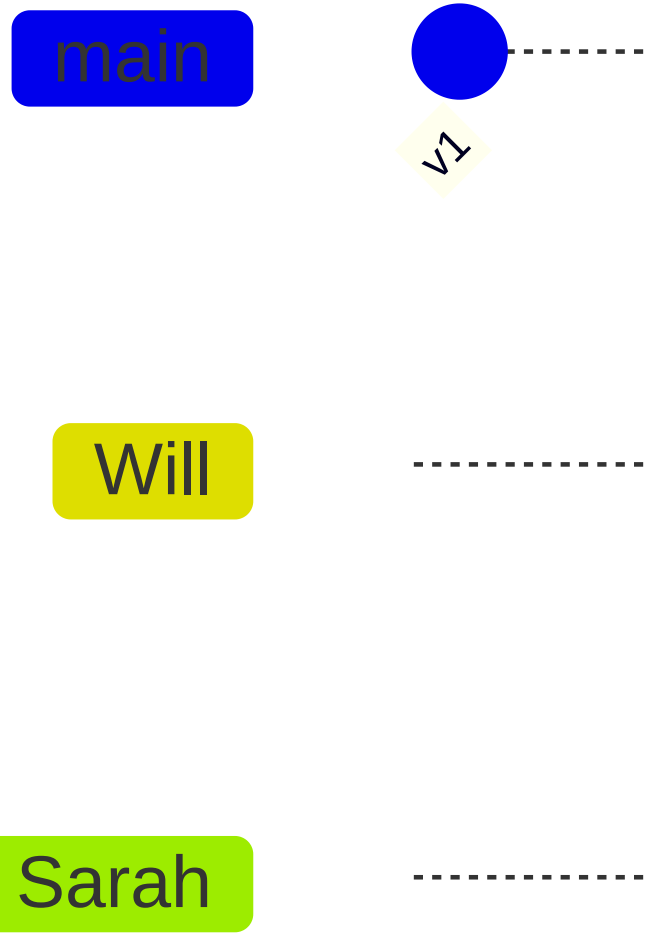
Collaboration

- Working on the same text
- Working from different computers
- Keeping a common shared history to make progress together

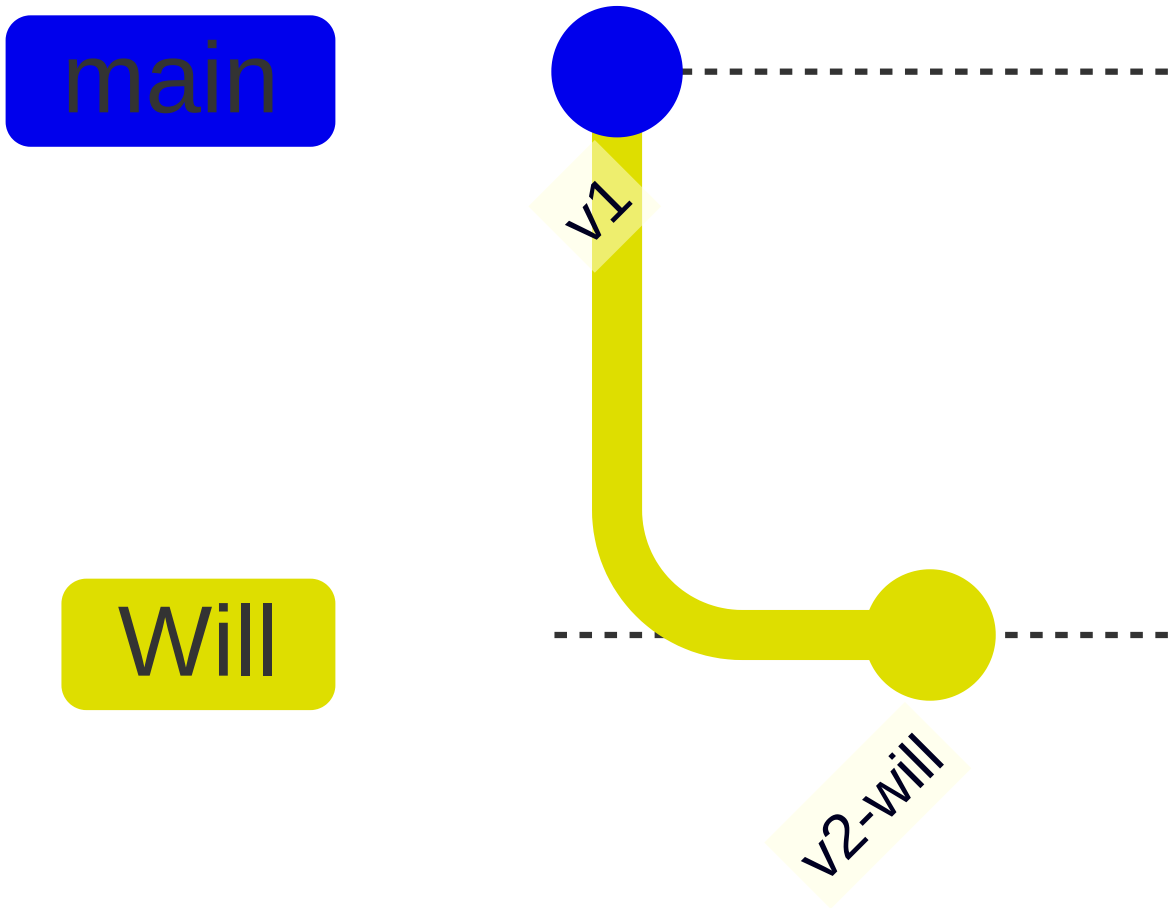
What we *don't* want



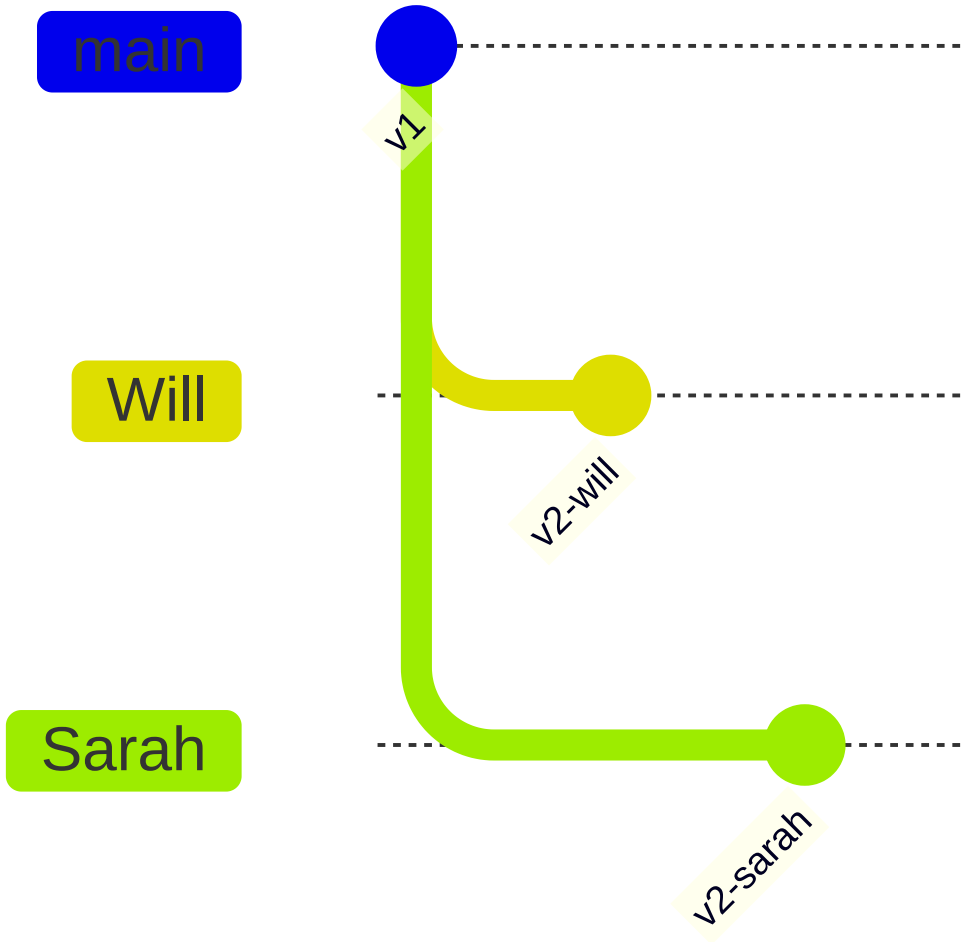
What we want



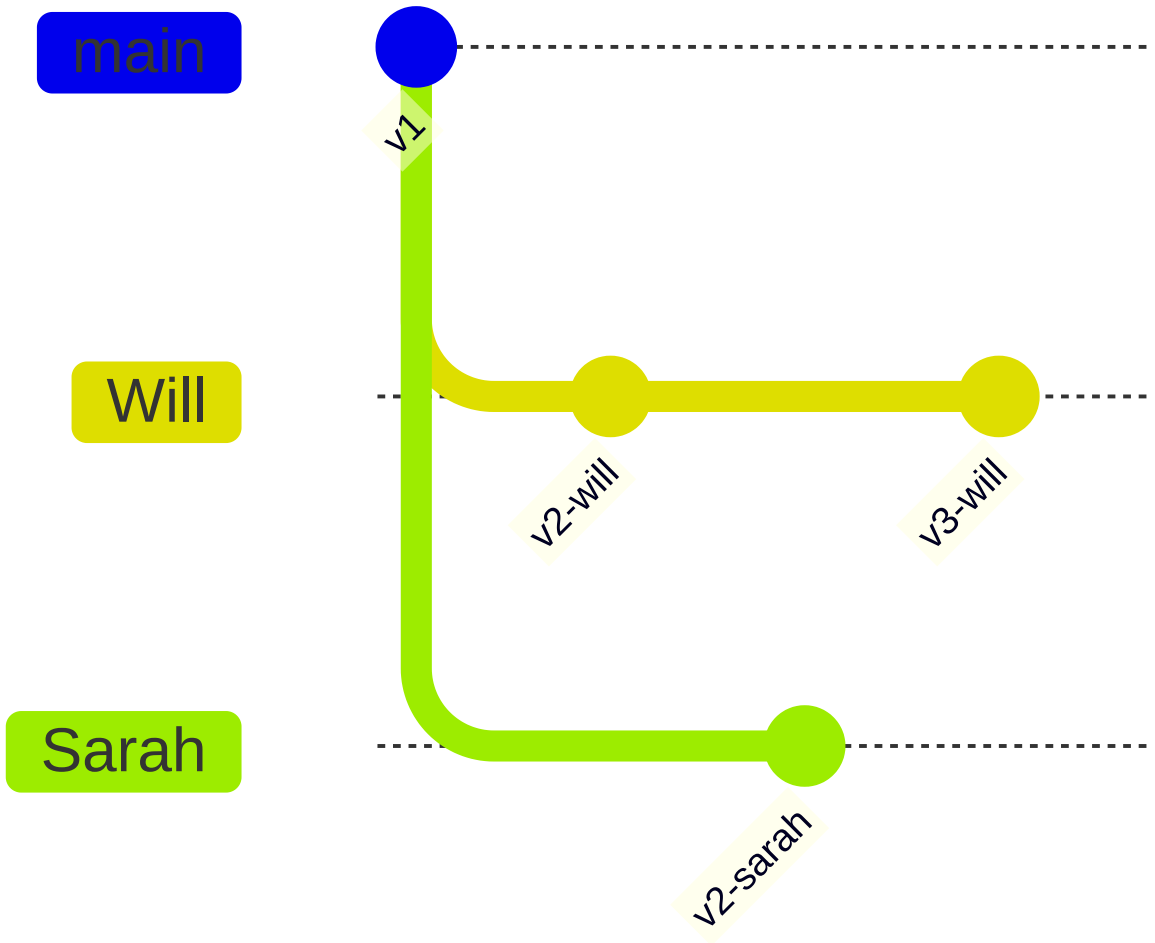
What we want



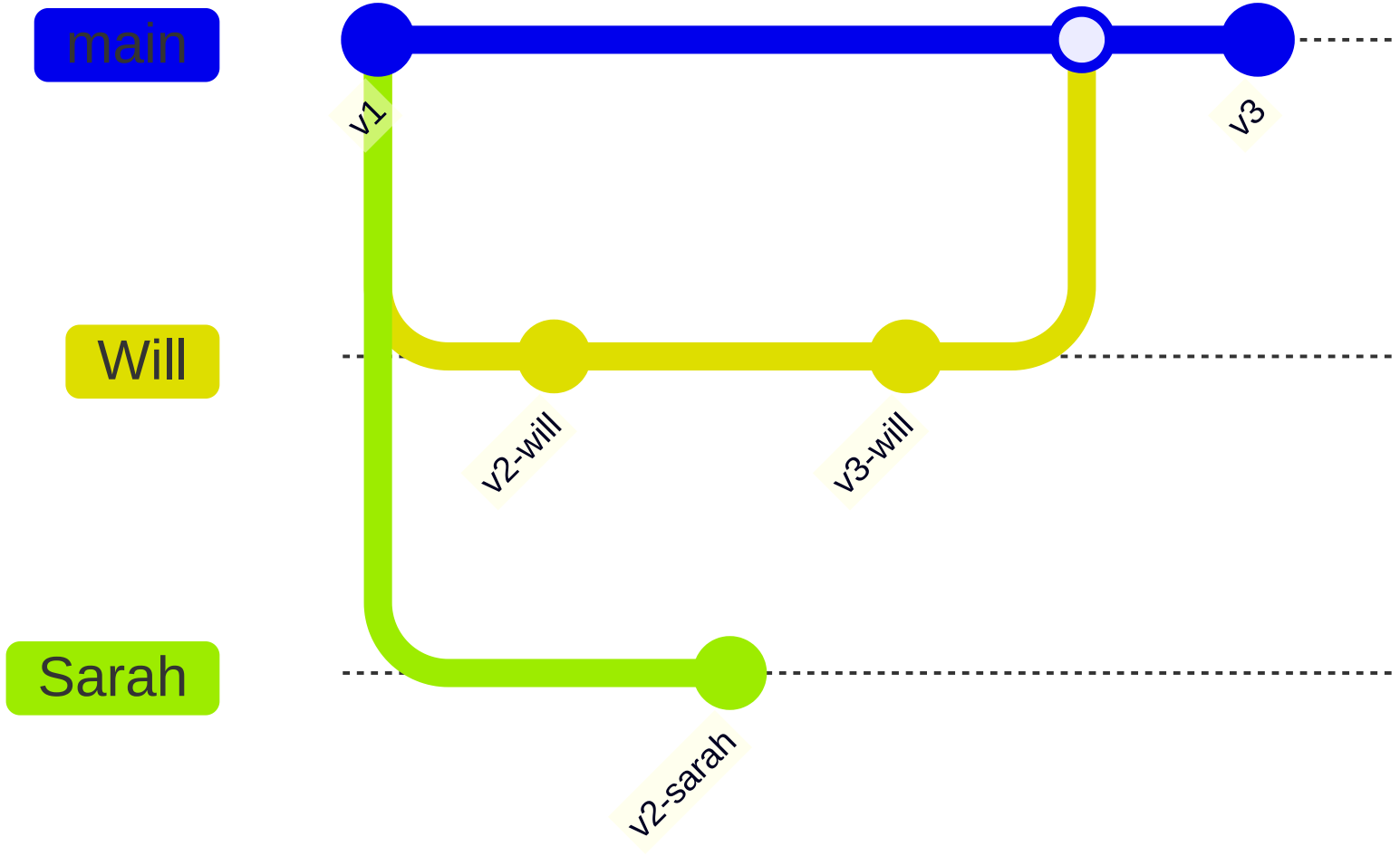
What we want



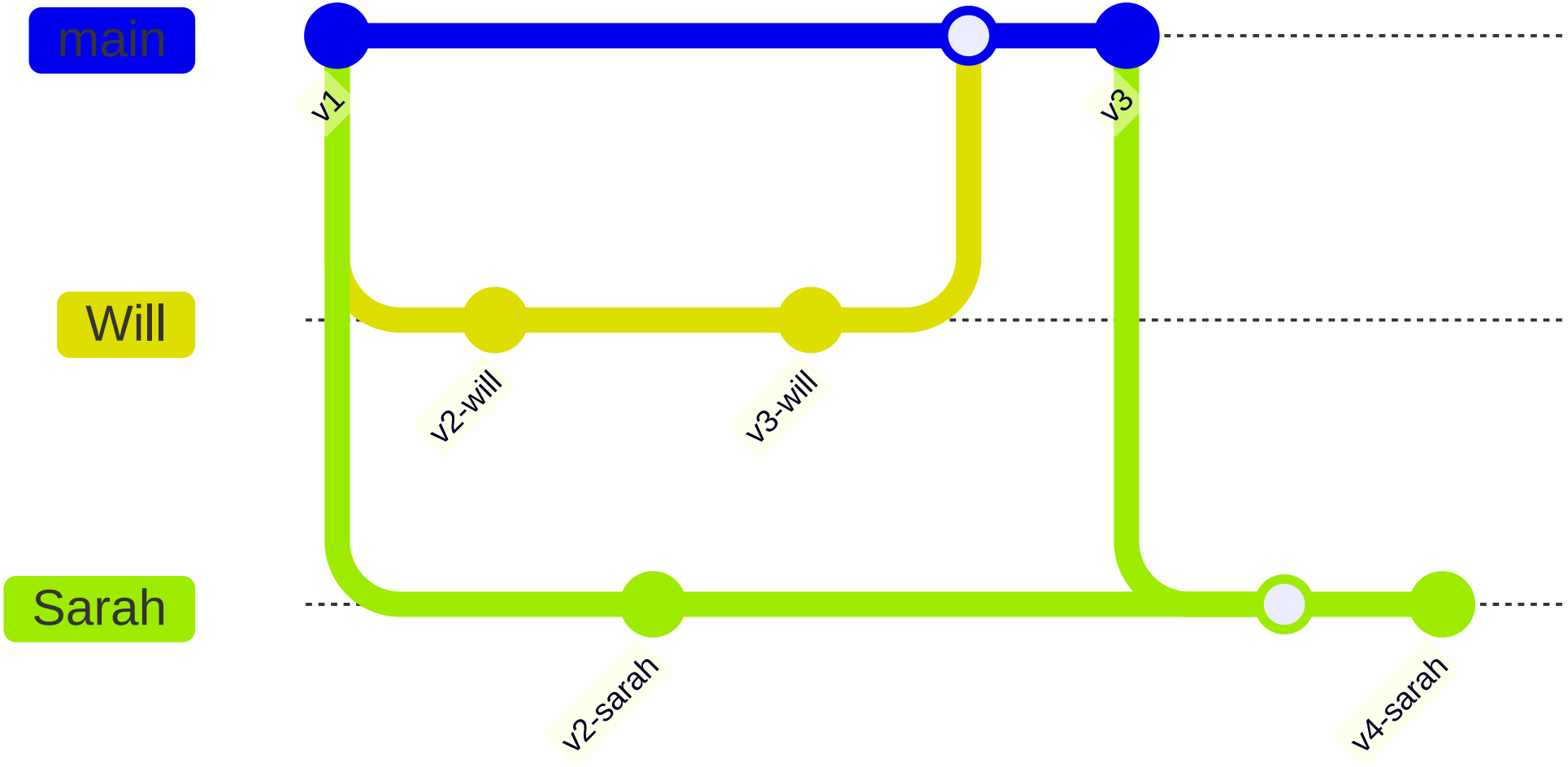
What we want



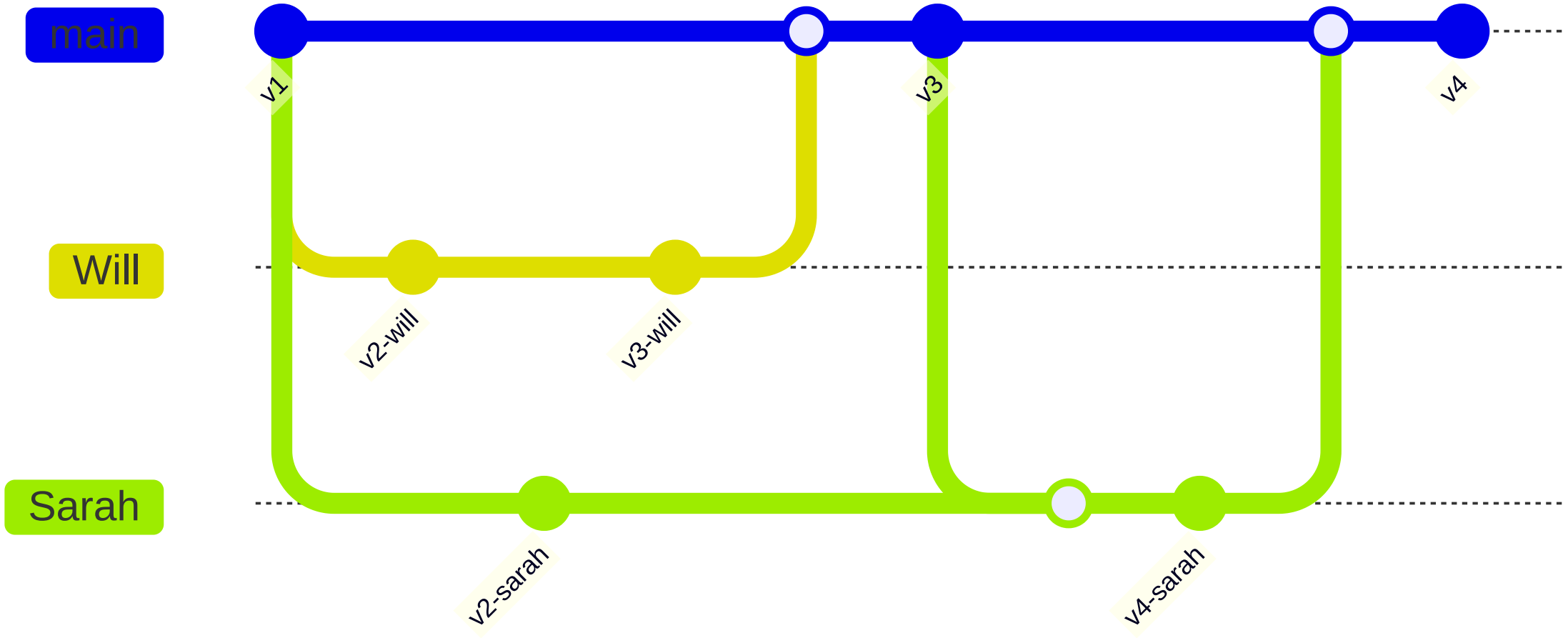
What we want



What we want



What we want



But those are local branches, aren't they?

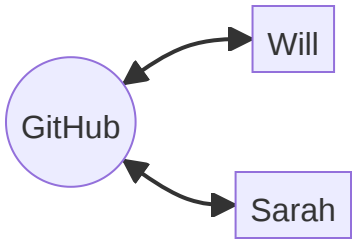
- Previous graph is an illustration with local branches
- But using branches is exactly what we are going to do!
- We just need to talk about the **remote**

Introducing remotes

Goals

- share changes from one's computer to others'
- receive changes from others' to one's computer

Introducing remotes



Local history vs remote history

Recap: data structures so far

- **branch** = set of commits
- **history** = set of branches

Local history vs remote history

What is stored?

Remote or local store a whole *history*.

Local history vs remote history

Where is it stored?

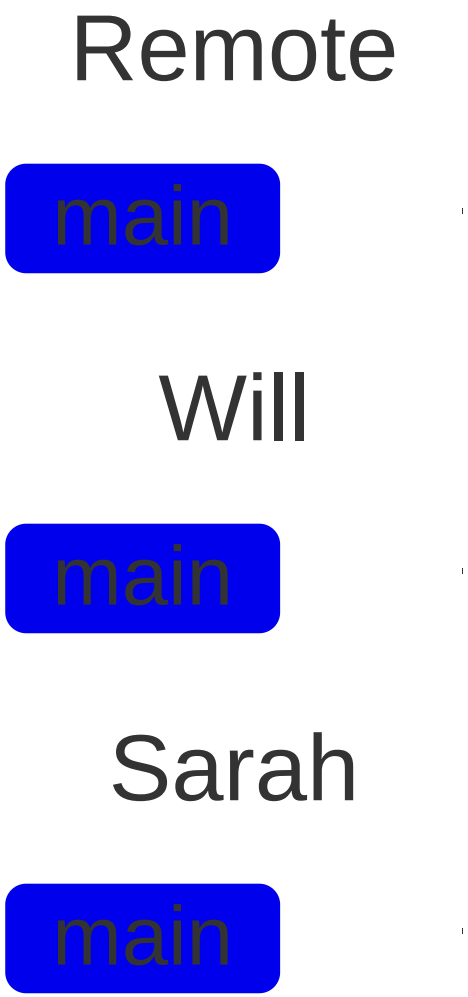
- Local: on your computer
- Remote: on a server provided by a service (e.g. GitHub, GitLab, Bitbucket, ...)

Local history vs remote history

What do they mean?

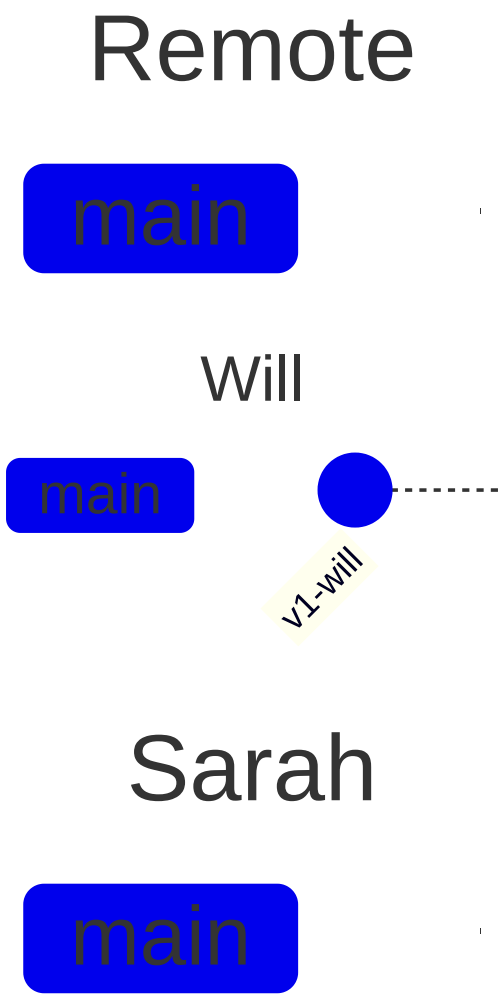
- Remote: the single source of truth for everyone
- Local: a copy you can work on

The workflow



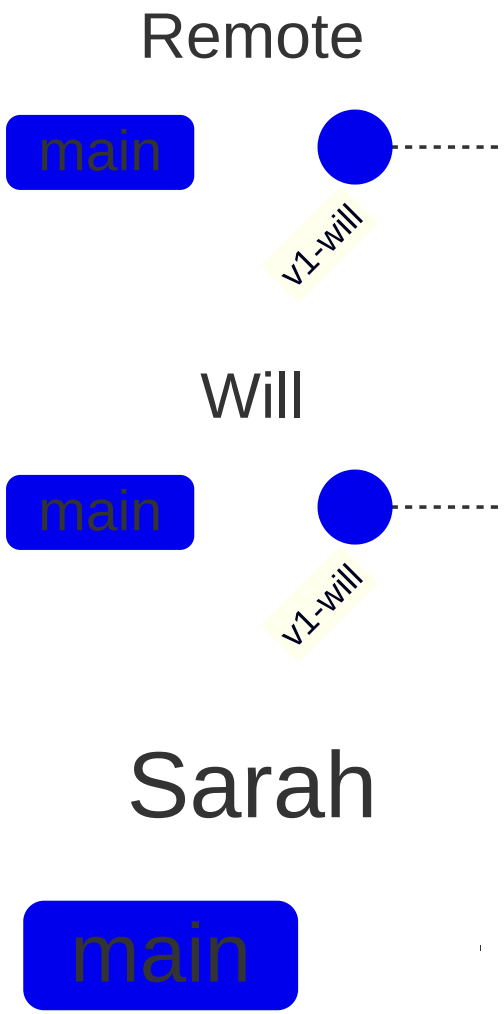
The workflow

```
will: git commit -m "v1-will"
```



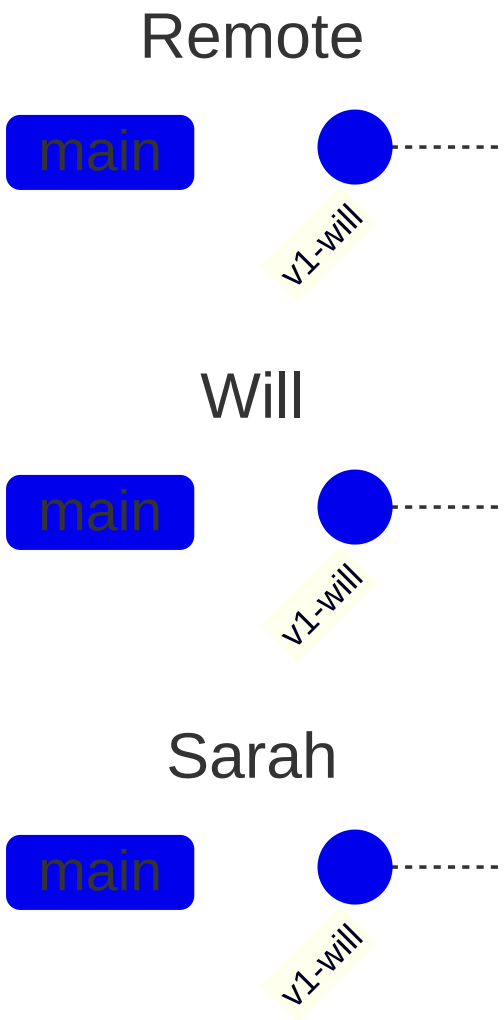
The workflow

```
will: git push
```



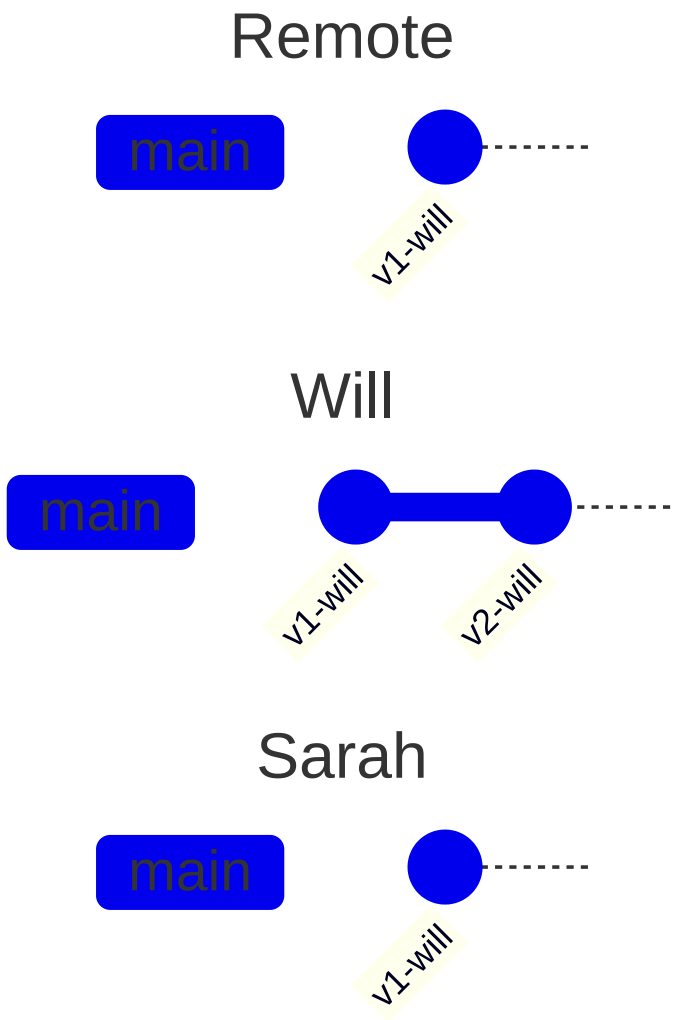
The workflow

```
sarah: git pull
```



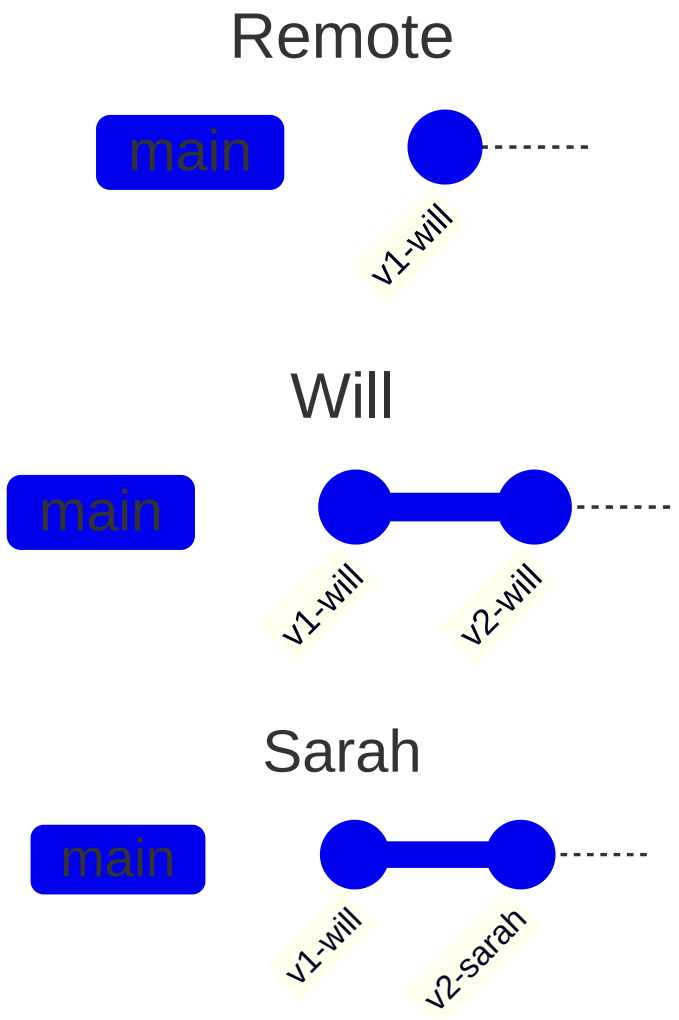
The workflow

```
will: git commit -m "v2-will"
```



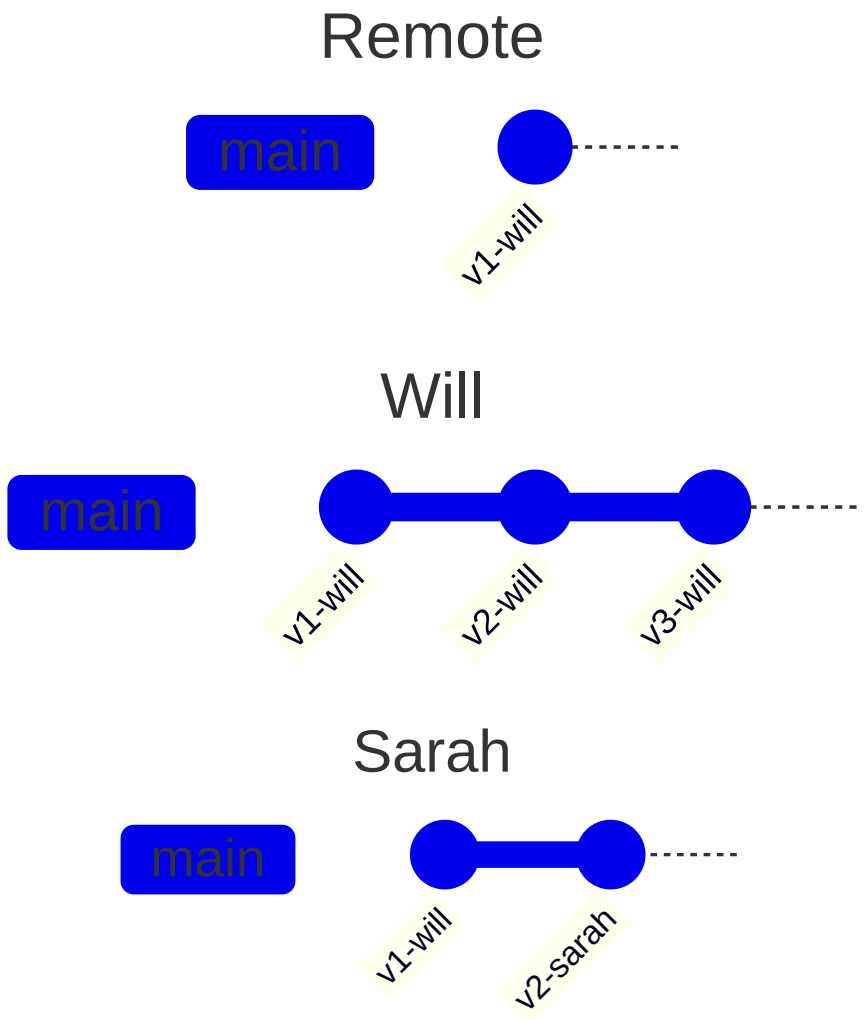
The workflow

```
sarah: git commit -m "v2-sarah"
```



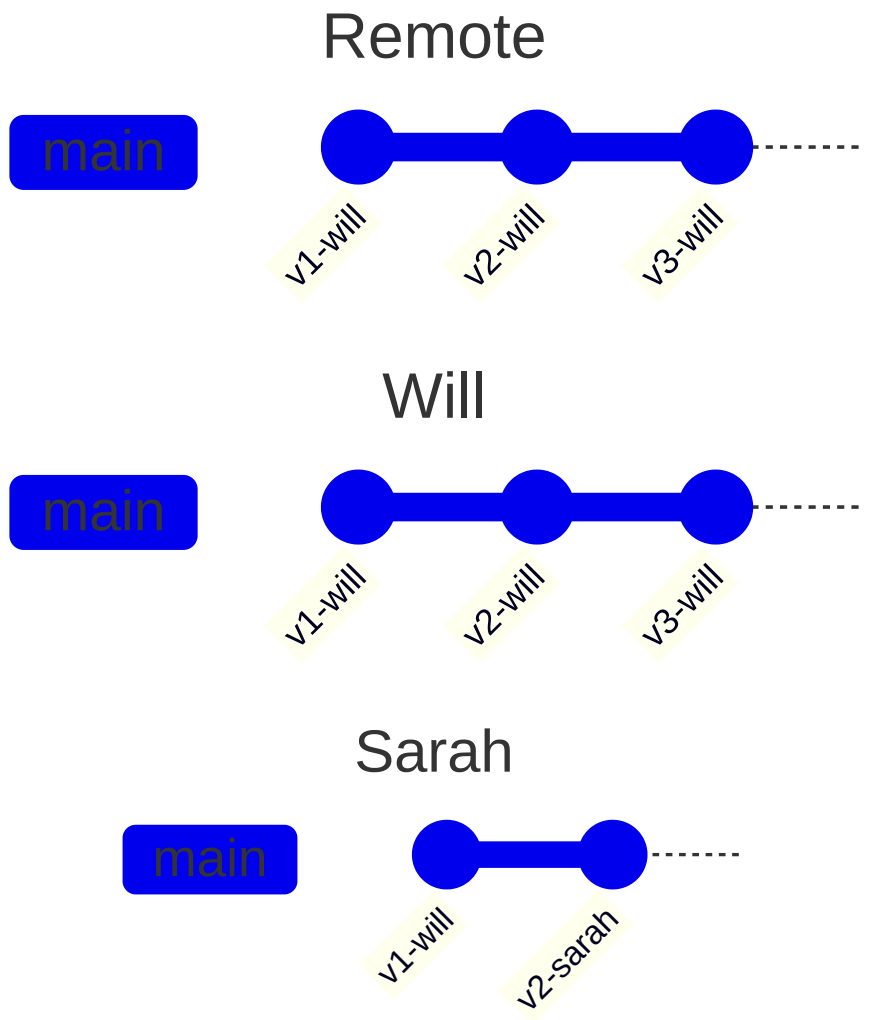
The workflow

```
will: git commit -m "v3-will"
```



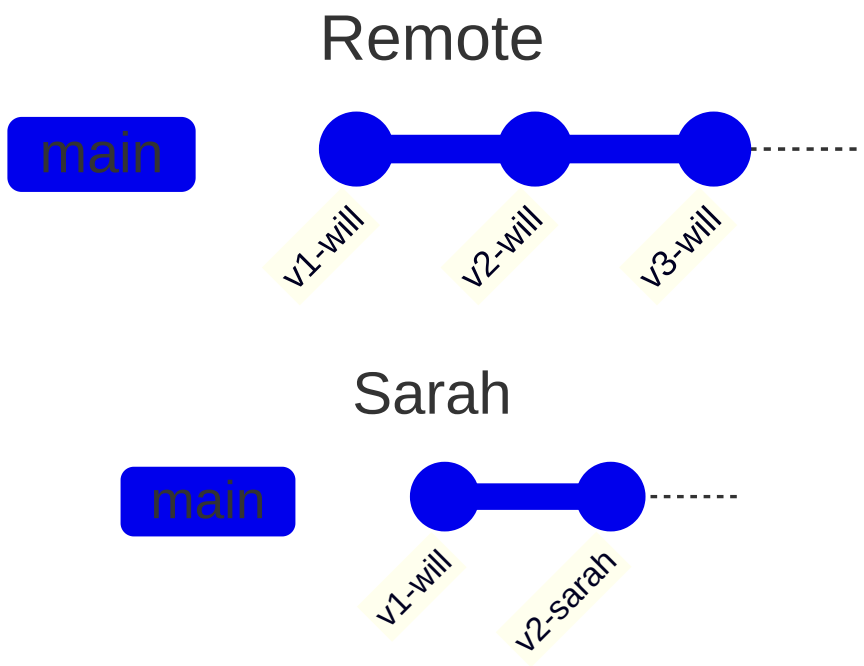
The workflow

```
will: git push
```



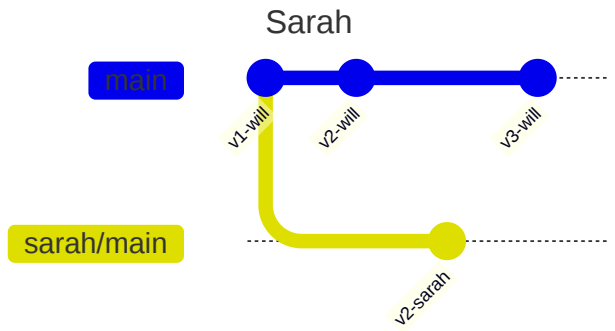
The workflow

```
sarah: git pull
-> CONFLICT!
```



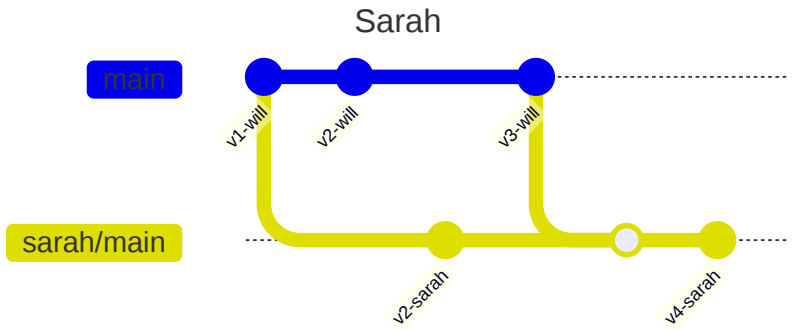
The workflow

```
sarah: git pull
-> CONFLICT!
```



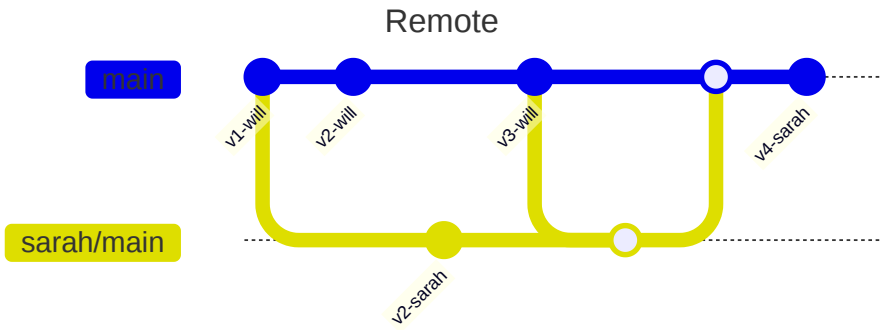
The workflow

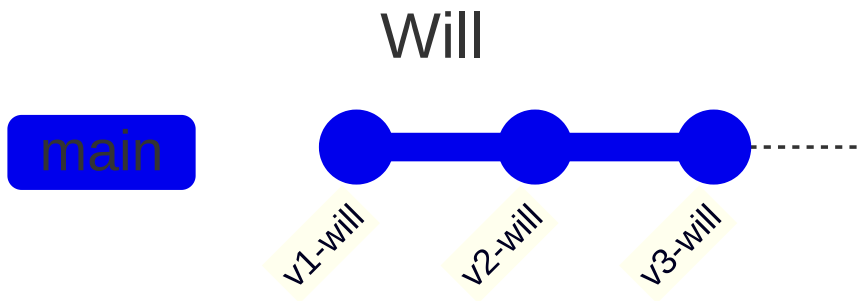
```
sarah: Solve conflicts
```

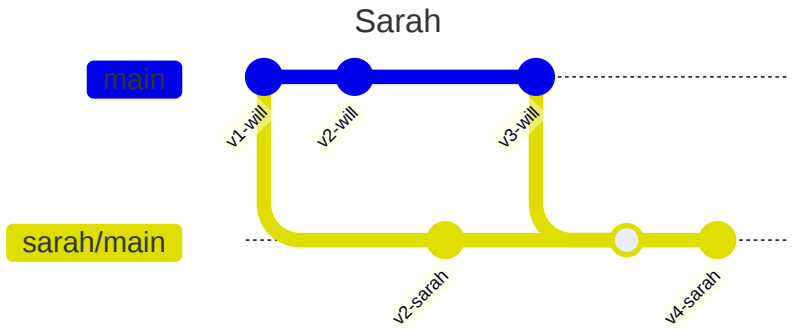


The workflow

```
sarah: git push
```

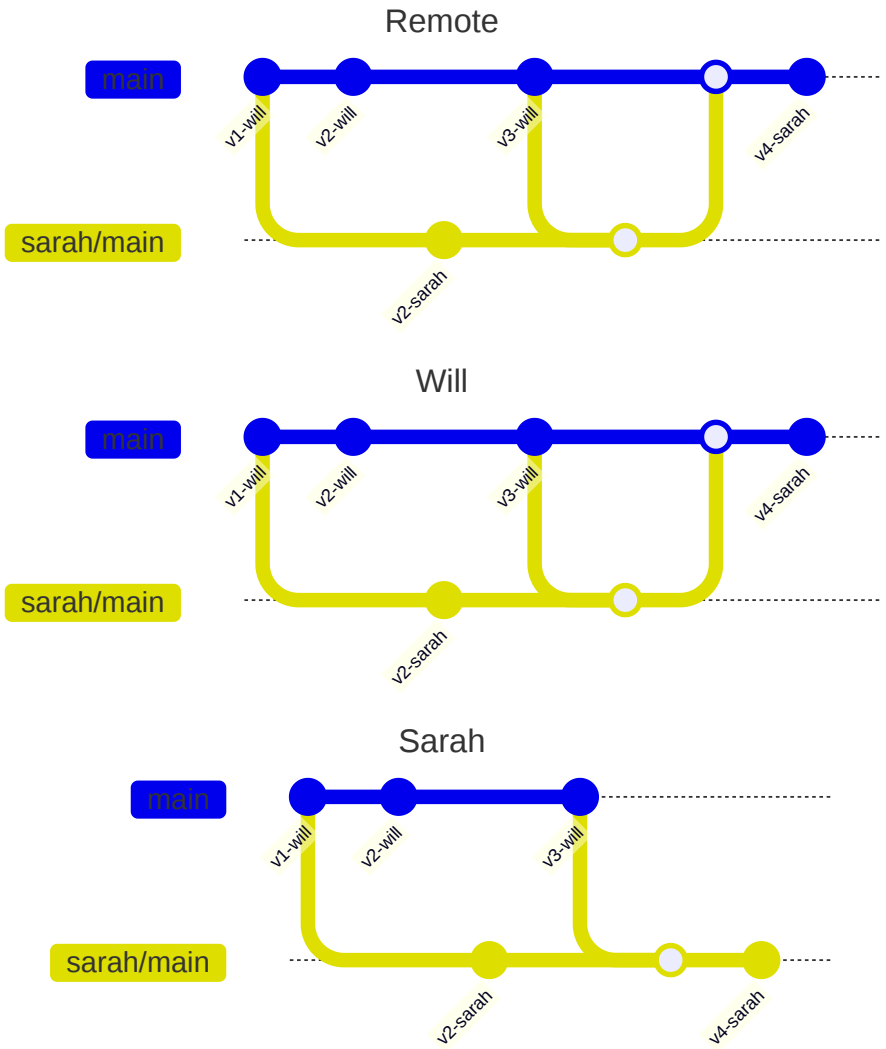






The workflow

```
will: git pull
```



Recap

- collaboration is centralized using a **remote**
- it stores a *history*
- it acts as a *single source of truth*
- you share by *pushing/pulling a branch*

Pitfalls

- don't diverge, merge often
- conflicts *will* happen, **solve them**!

All in all

- Using git we will
  - keep track of your changes progressively
  - collaborate

Go beyond

- rebase
- reflog
- worktree

Play to learn

Oh My Git!