# Scientific Software Development with Python

Project management and version control


CHALMERS
UNIVERSITY OF TECHNOLOGY

Simon Pfreundschuh
Department of Space, Earth and Environment

# Agenda



1. Course overview
2. Agile software development
3. Version control with git
4. Project pitches

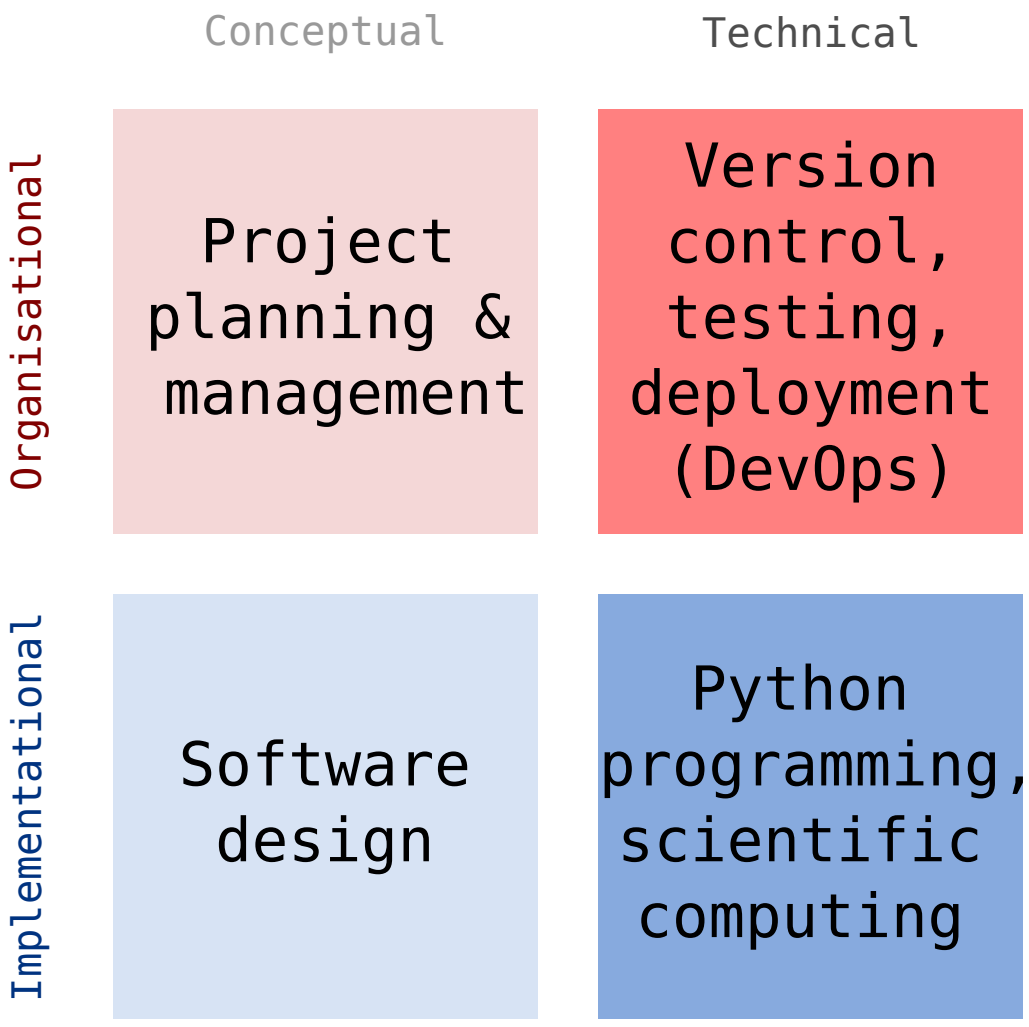
**The three fundamental functions of software**[1]**:**

1. Performing its given task
2. Affording change
3. Communicate to its readers

[1] Adapted from *Martin, Robert C. Agile software development: principles, patterns, and practices*.

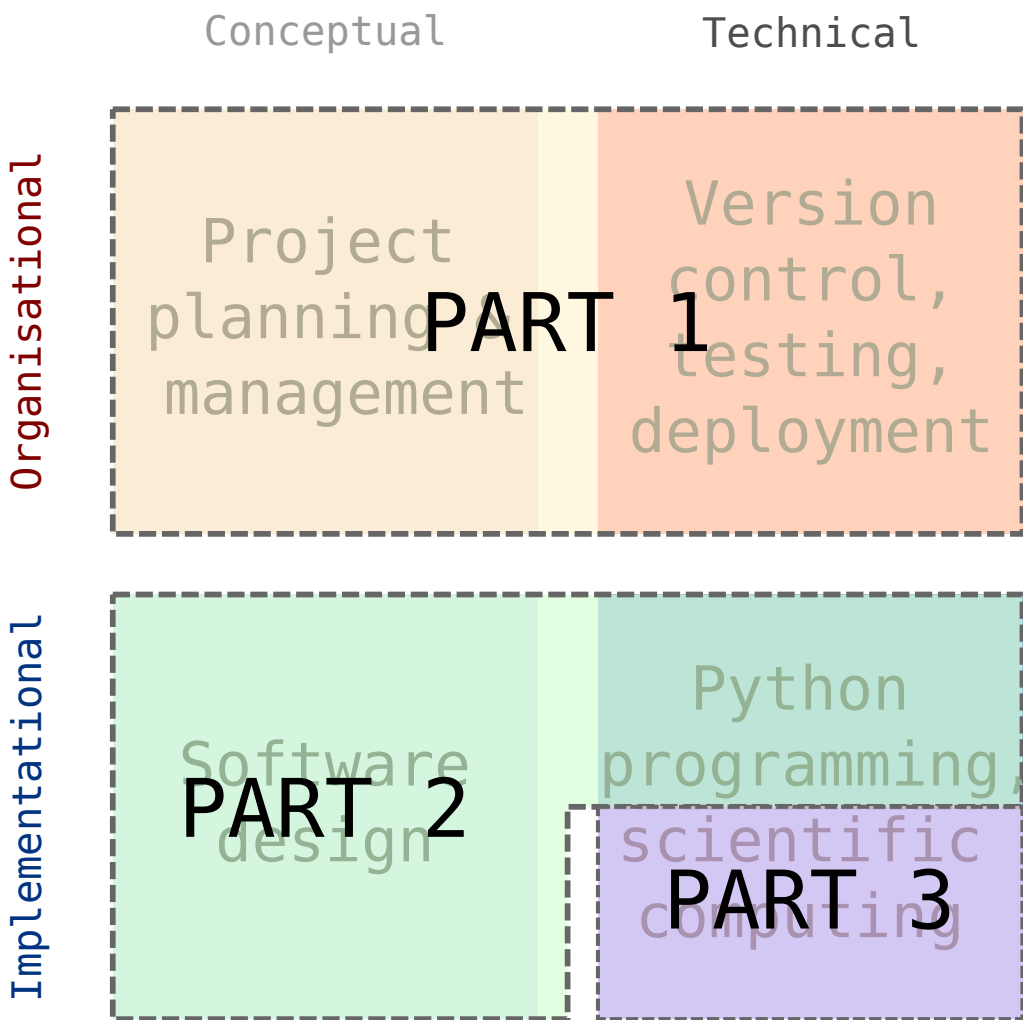
# Dimensions of software development





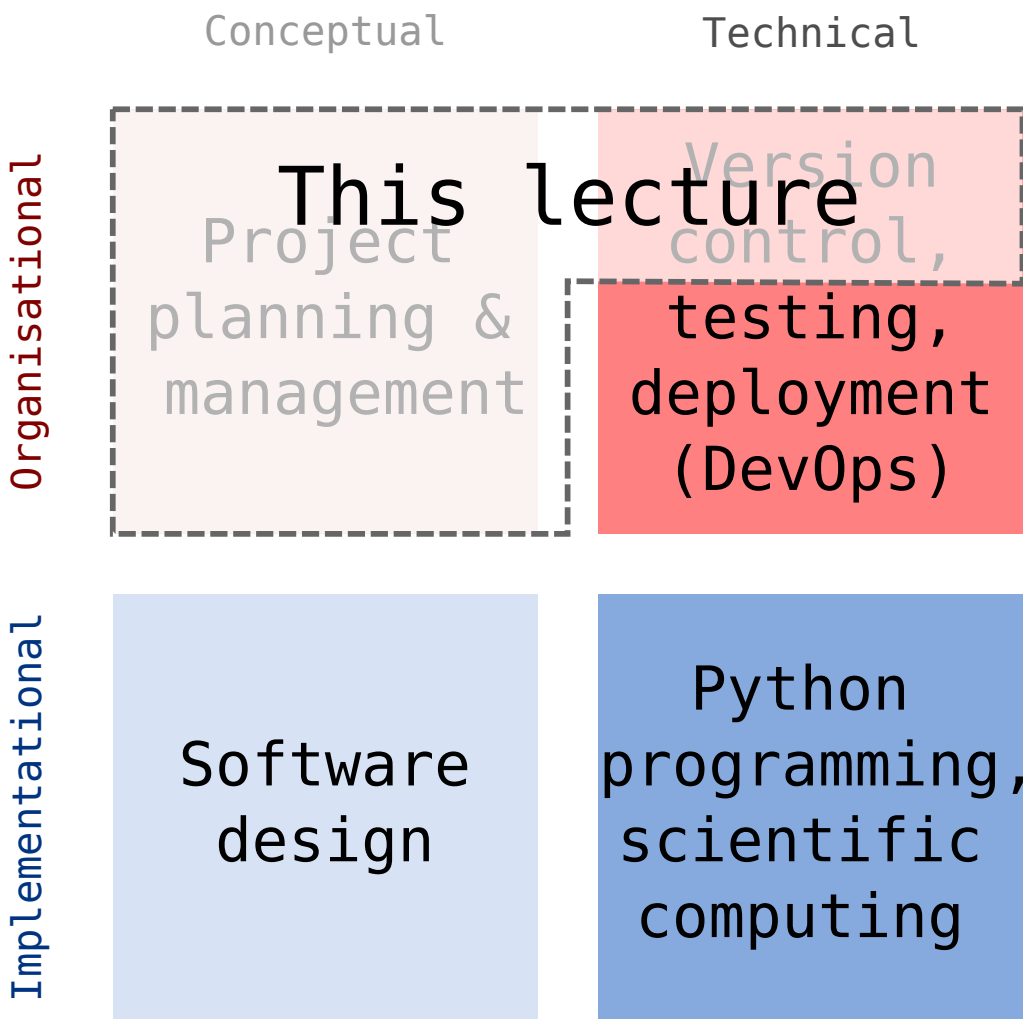
# Dimensions of software development





# Dimensions of software development





# Agenda



1. Course overview

**2. Agile software development**

3. Version control with git

4. Project pitches

# Introductory discussion

1. Discussion in breakout rooms (5 min):
   - Presents yourself to your colleagues
   - Discuss following questions:
     - How big are the projects that your are involved in?
     - How are these projects managed?
2. Presentation of results from breakout rooms (< 1 min / room)
   - Select a speaker to summarize results.

**90s and Early 00s:**

- *Heavyweight development processes* prevalent in the software industry:
  - Detailed documentation and planning
  - Sequential process with big, heavy releases
- Problems:
  - Unsuited for complex/explorative projects
  - Can't incorporate change

**Emergence of agile practices:**

- 90s: Development of programming and design practices to address problems of heavyweight processes
- 00s: Generalization of agile practices to a management framework.

# The agile manifesto (2001)

## Values

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

# The agile manifesto (2001)

## Principles

1. Customer satisfaction highest priority
2. Welcome changing requirements
3. Deliver working software frequently
4. Business people and developers must work together daily throughout the project
5. Build projects around motivated individuals
6. Face-to-face conversation to convey information

## Principles — in science

1. Customer satisfaction highest priority (✓)
2. Welcome changing requirements ✓
3. Deliver working software frequently ✓
4. Business people and developers must work together daily throughout the project ✗
5. Build projects around motivated individuals ✓
6. Face-to-face conversation to convey information (✓)

# The agile manifesto (2001)

## Principles

7. Working software is the primary measure of progress
8. Sustainable working pace
9. Technical excellence and good design
10. Simplicity: Maximizing the amount of work not done
11. The best results emerge from self-organizing teams
12. Regular reflection on how to become more effective

## Principles — in science

7. Working software is the primary measure of progress ✓
8. Sustainable working pace ???
9. Technical excellence and good design ???
10. Simplicity: Maximizing the amount of work not done ✓
11. The best results emerge from self-organizing teams ✓
12. Regular reflection on how to become more effective ???

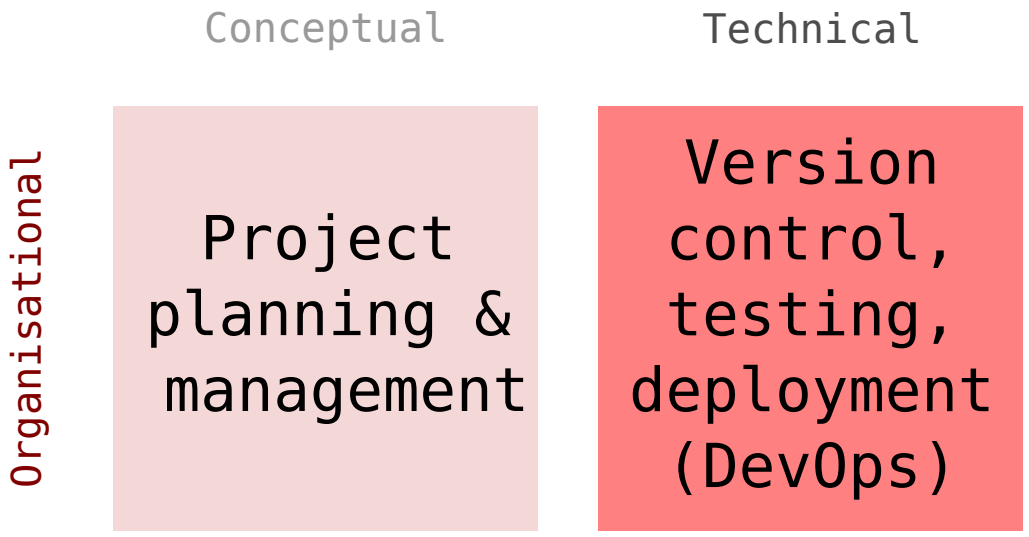
# The agile manifesto (2001)

**Agile principles, summarized**

- Self-organizing, cross-functional teams
- Early delivery, evolutionary development
- Flexible response to change

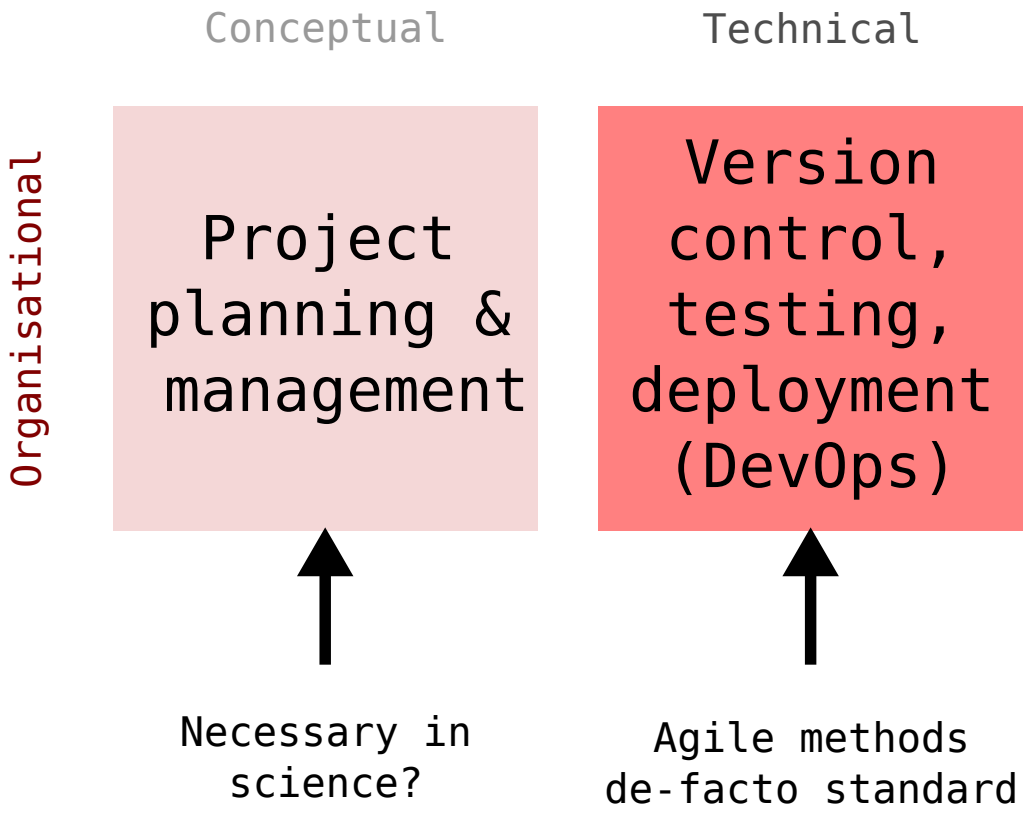
There is considerable overlap between agile principles and research. **This makes many agile approaches applicable also for our work.**

# So, what does it mean?




- Organizational-conceptual level: Agile management practices
- Organizational-technical: Agile development techniques

# So, what does it mean?




- Organizational-conceptual level: ???[2]
- Organizational-technical: Agile techniques are de facto standard

[2] I, personally, don't know. Hopefully, we'll find out during the course.

## Iterative development

- Release early and often
- Work is performed in *sprints*.
- A sprint consists of three phases:
  1. Planning
  2. Implementation
  3. Reflection

## Project planning

- Acknowledge uncertainty:
  - Not all can be known details from the start
  - Requirements are likely going to change during the process
- Keep it simple:
  - Focus on functionality from user perspective (**features**)
  - Omit implementation details
- Be flexible:
  - The project plan may change during the project

### User story

- informal, natural language description of a feature
- Represents a requirement

### Backlog

- Collection of user stories that define the project
- Task pool for next development iteration
- Can be extended throughout the project

The backlog defines the project scope. It is the essential planning tool of agile project management.

## 1. Sprint planning

- Defines the work to be done in a sprint by selecting user stories from the backlog
- The team decides how the stories should be implemented
- Amount of work based on *velocity* estimates from previous iteration
- Defines how increment should be delivered

Result of the sprint planning should be a tangible **sprint goal**, which the team commits to as a whole.

## During the sprint

- The team self-organizes its work to reach the sprint goal
- Short but regular meetings to optimize the probability to reach the sprint goal
- During meeting, each team member explains[3]:
  1. What did I do since the last meeting that helped the team meet the sprint goal?
  2. What will I do now to help the team reach the sprint goal?
  3. Do I see any impediments that may prevent the development team from reaching the sprint goal?

[3] Adapted from `https://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf`

## After the sprint

1. Sprint review:
   - Which user stories have been implemented?
   - What problems occured?
   - What is the current state of the backlog?
   - How to proceed?
     - Input for next sprint planning
2. Retrospective
   - The team reflects on its development process
   - The team identifies ways to:
     - Make work more effective and enjoyable
     - Increase product quality
   - The team decides which improvements it wants to implement in the next iteration

# Your project work

- Project work will be done in four sprints
- After each sprint, all teams will present their progress (retrospectives in schedule)[3]

The purpose of the retrospectives is to learn from each other. Share your experiences. Focus on your learnings and problems.

# Agenda



1. Course overview

2. Agile software development

**3. Version control with git**

4. Project pitches

## git

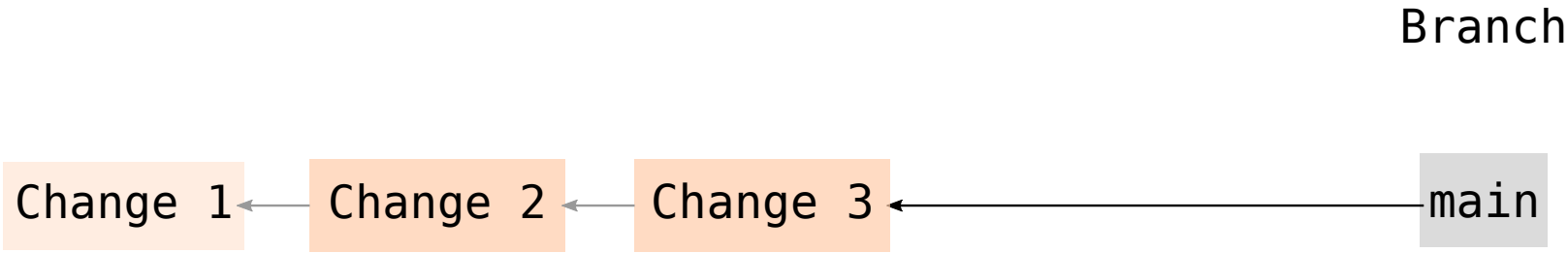
- Created by Linus Torvalds for the development of the Linux kernel
- Distributed version-control system:
  - Keeps track of changes in source code
  - Allows synchronizing changes with repositories in arbitrary locations
- Usage:

```
$ git <command> <args>
```

## GitHub

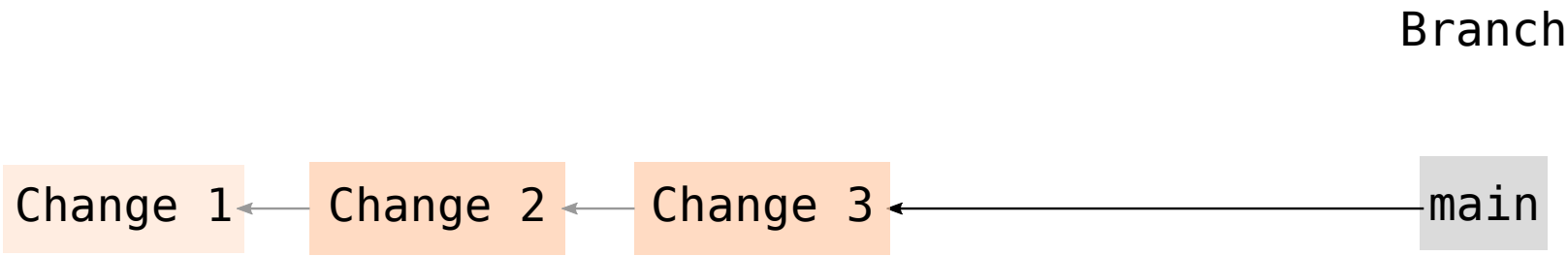
- One of several hosting services for git repositories
- Additional features:
  - Simple web hosting
  - Communication platform
  - Issue tracking

If you don't have an account, get one at `https://github.com`!




## Principles

- git tracks development as a sequence of changes in **repositories**
- Examples of changes:
  - Adding a file
  - Removing/renaming a file
  - Changing its content




## Principles

- A single sequence of changes is called a **branch**
- the current state of your directory is defined by which branch is **checked out** (grey rectangle)
- The currently checked out files are referred to as the **working tree**

# The most important git command



```
$ git status
```

- Prints all relevant information about the current repository:

```
On branch main

Your branch is up to date with 'origin/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:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  STUDENT_LIST.md

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

## Your identity

```
$ git config --global user.name "your_name"
$ git config --global user.email your_email@chalmers.se
```

## Your editor

```
$ git config --global core.editor vim # or emacs, nano, ...
```

## Default branch name

- Change default branch name to main[4]:

```
$ git config --global init.defaultBranch main
```

[4] https://tools.ietf.org/id/draft-knodel-terminology-00.html

Branch

main

## Creating an empty repository

```
$ git init
```

- Creates and initializes an empty repository in the current directory


## Committing changes

1. **Staging**: `git add` marks changed or new files for the next commit
2. **Commiting**: `git commit` adds the changes from the staged files as an atomic change to the repository[5]

[5] Explicit staging can be skipped by using `git commit` followed by a list of file names.


## Committing changes

```
$ git add README.md LICENSE.md # Stage files for commit
$ git commit                   # Commit to branch
```

or

```
$ git commit README.md LICENSE.md
```

# Committing changes



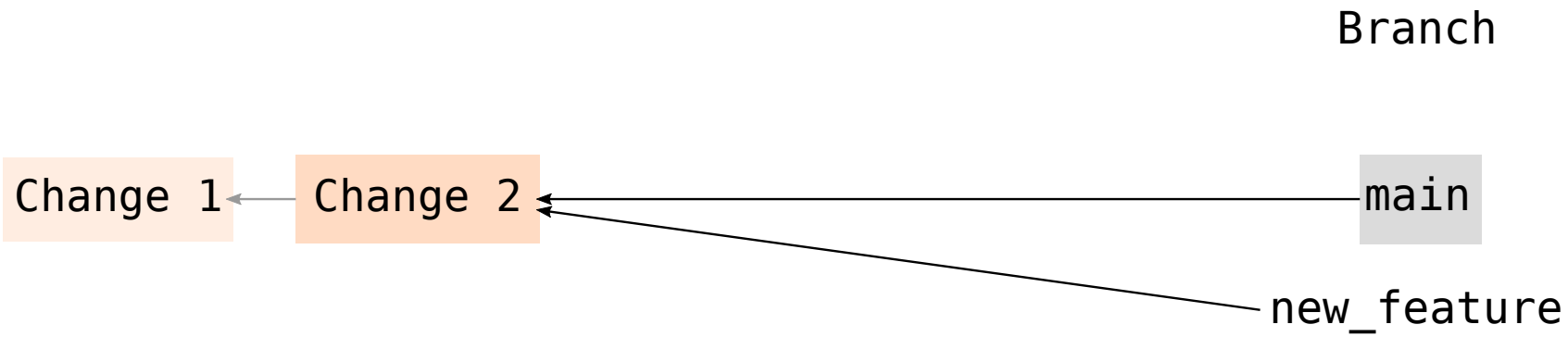

- Use `git log` to list the most recent changes in branch
- Commits are identified by their checksum (the long, seemingly random number on the second line)

```
$ git log
commit 21959efd7528a00fab48062473f0409acd74e113 (HEAD -> main)
Author: Simon Pfreundschuh <simon.pfreundschuh@chalmers.se>
Date:   Mon Sep 7 21:09:44 2020 +0200

    Added README and LICENSE files.
```

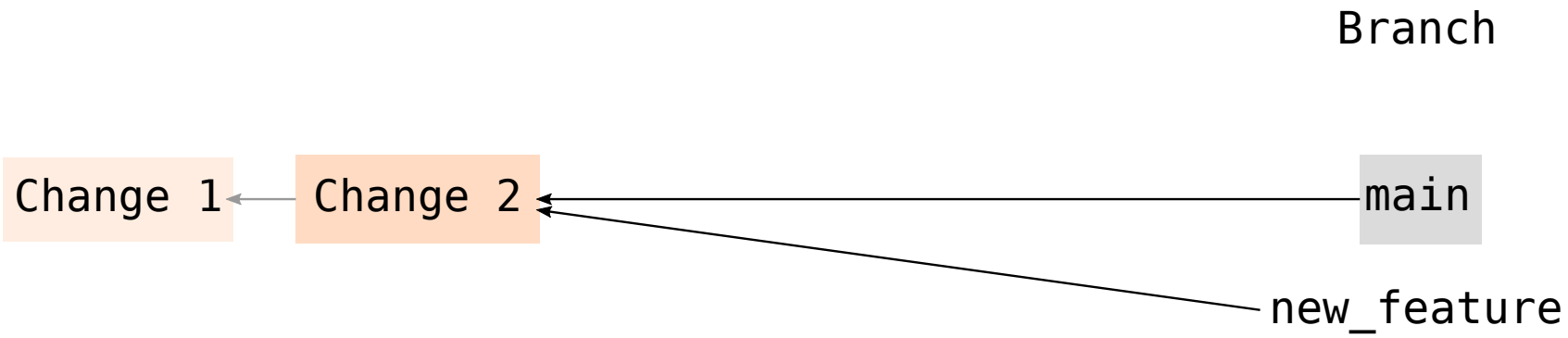

Branch
Change 1
Change 2
main

# Branching



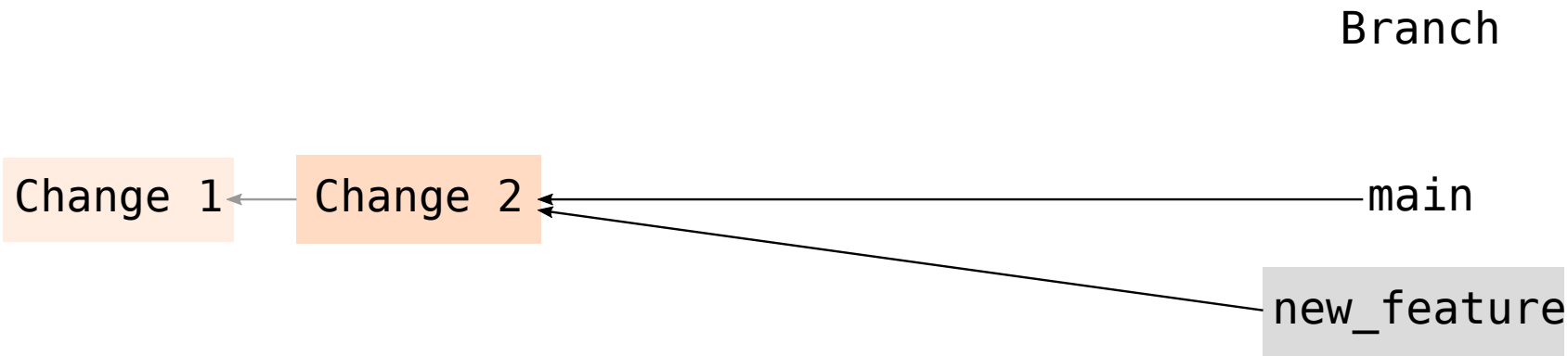

- Branching allows experimenting with new features at the same time as keeping a snapshot of the currently working code
- Use `git branch` to create a new branch:

```
$ git branch new_feature
```




- However, currently we are still on the `main` branch[6]
- All git commands affect the branch *that is currently checked out*, i.e. `main`.

[6] Indicated by grey rectangle in figure. Use `git status` to verify.

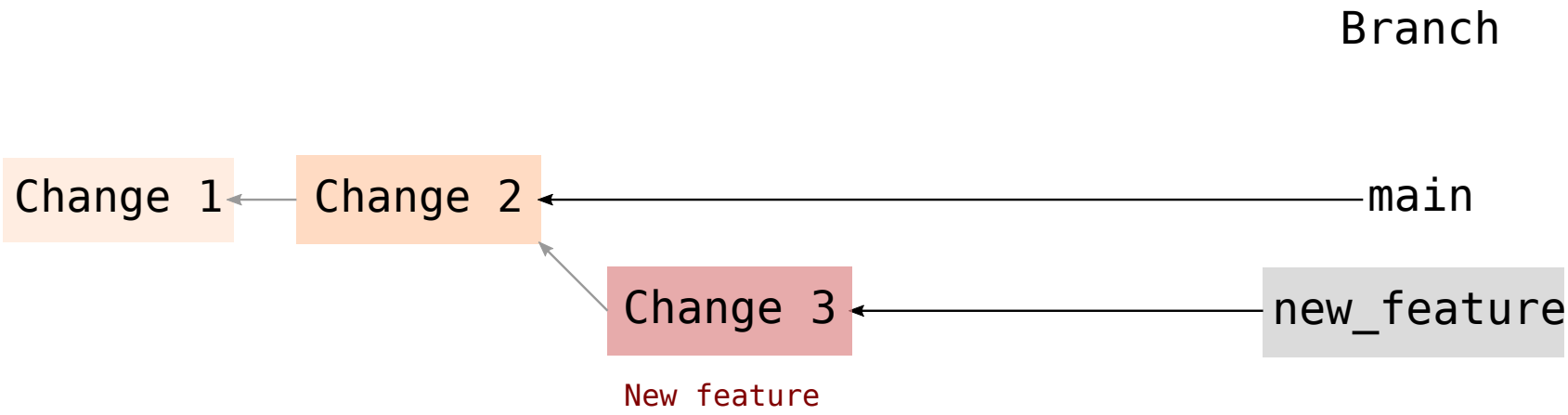



- To switch to the new branch we need to *check it out*[7] :

```
$ git checkout new_feature
```

- Since the development in both branches is identical, the working directory doesn't change.

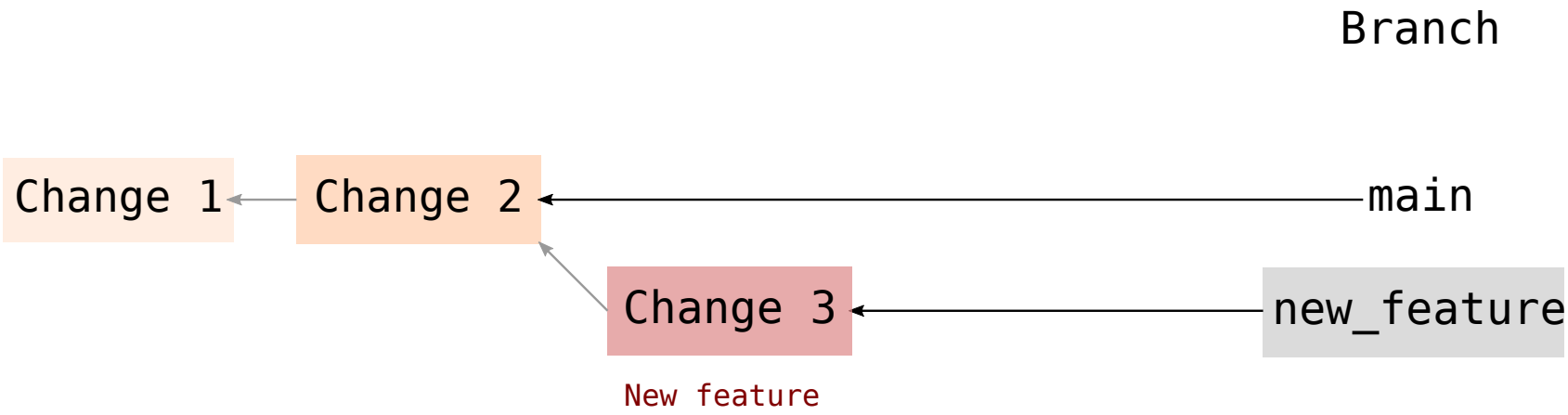
[7] Alternatively, use `git checkout -b <branch_name>` to create new branch and to check it out immediately.




- We can now add the new feature to the branch:

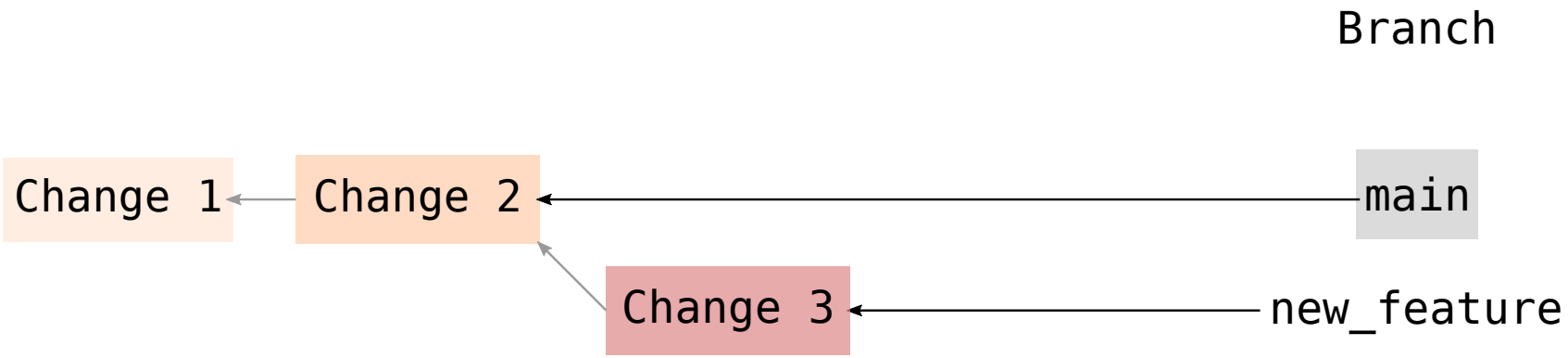
```
$ git add new_feature.py
$ git commit
```

# Merging




- Integrating changes from another branch is called merging
- To merge the changes in `new_feature` into `main`:

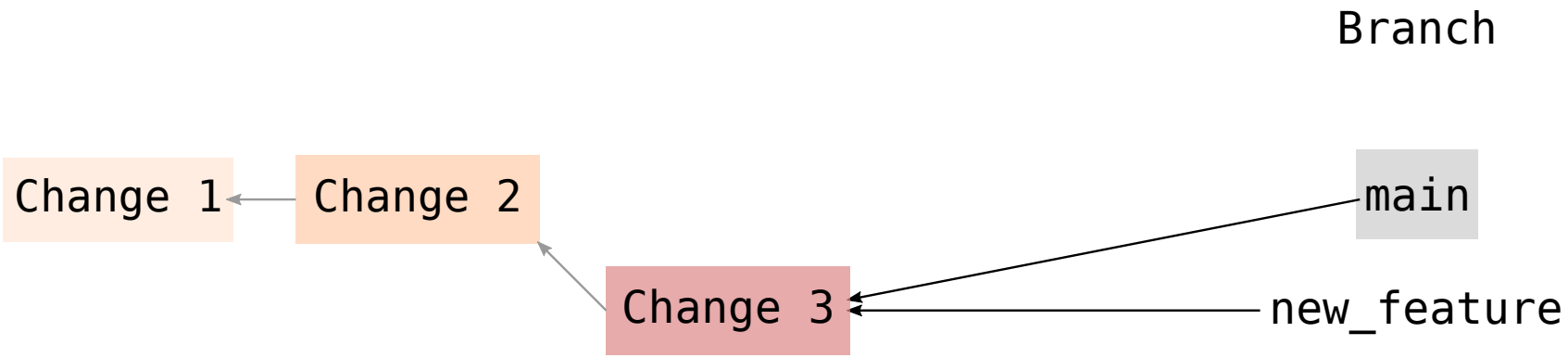
# Merging




- Integrating changes from another branch is called merging
- To merge the changes in `new_feature` into `main`:
  1. Checkout main

```
$ git checkout main
```

# Merging

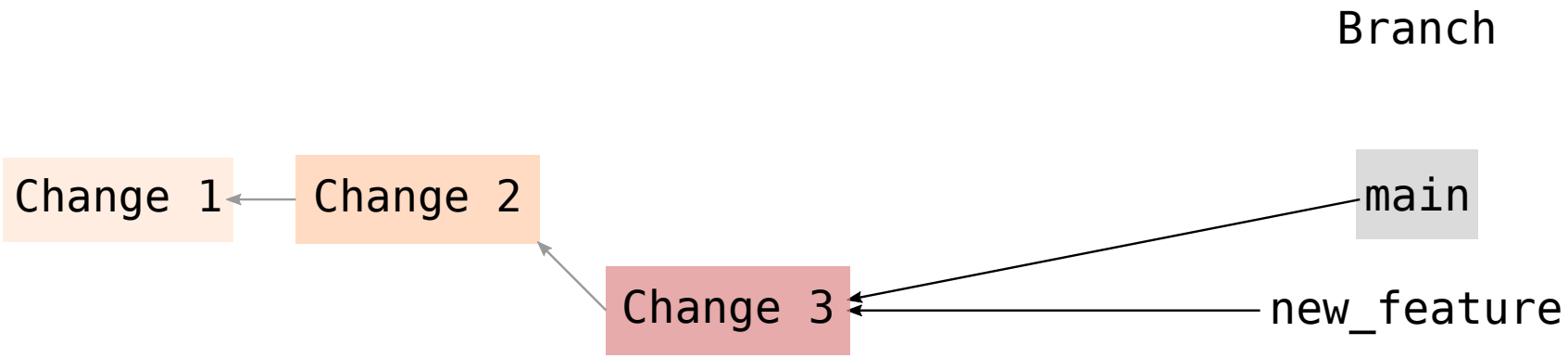



- Integrating changes from another branch is called merging
- To merge the changes in `new_feature` into `main`:
  1. Checkout main

```
$ git checkout main
```

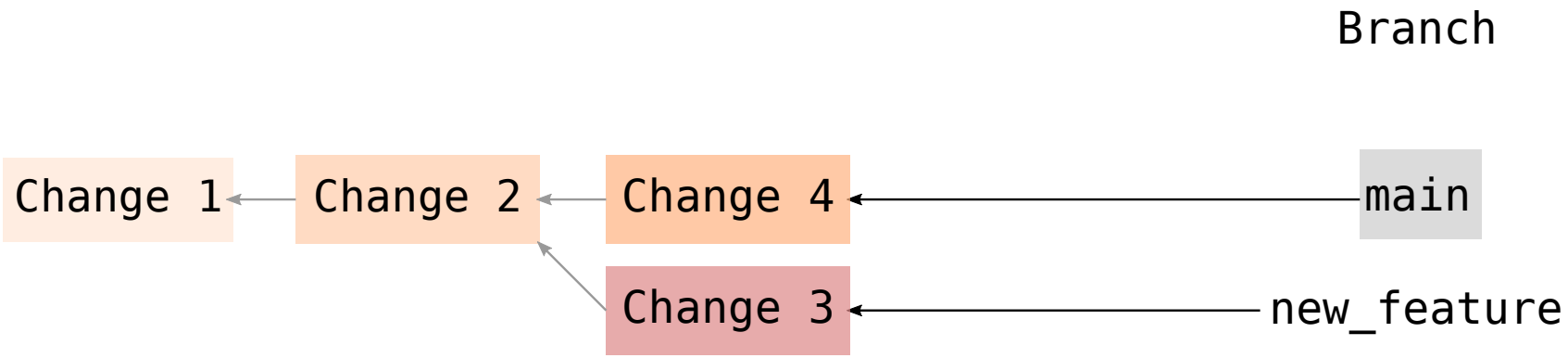
  2. Merge:

```
$ git merge new_feature
```



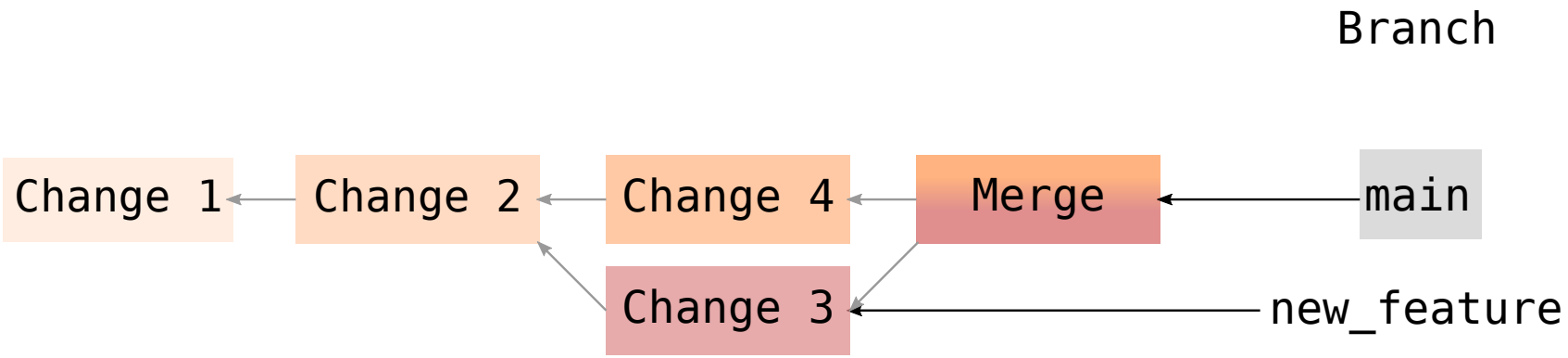

## Fast forwarding

- The special case where the branch to merge is simply ahead of the other branch is called **fast forwarding**.
- Fast forwarding is trivial



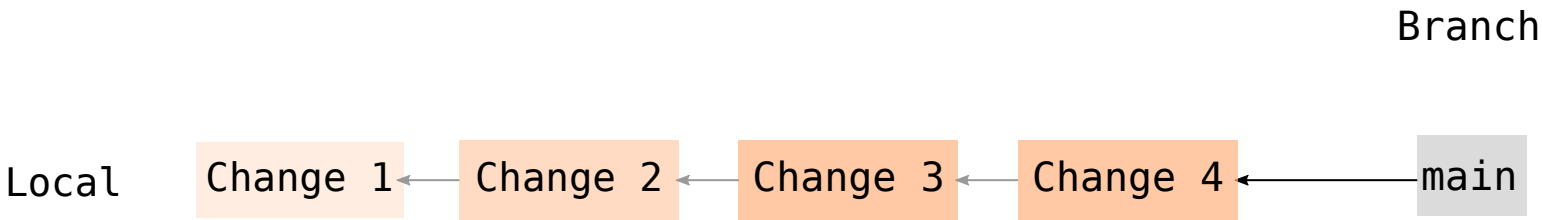

## Diverging branches

- However, typically the branches will *diverge*
- In this case a merge operation is required, which itself is a change commited to the repository



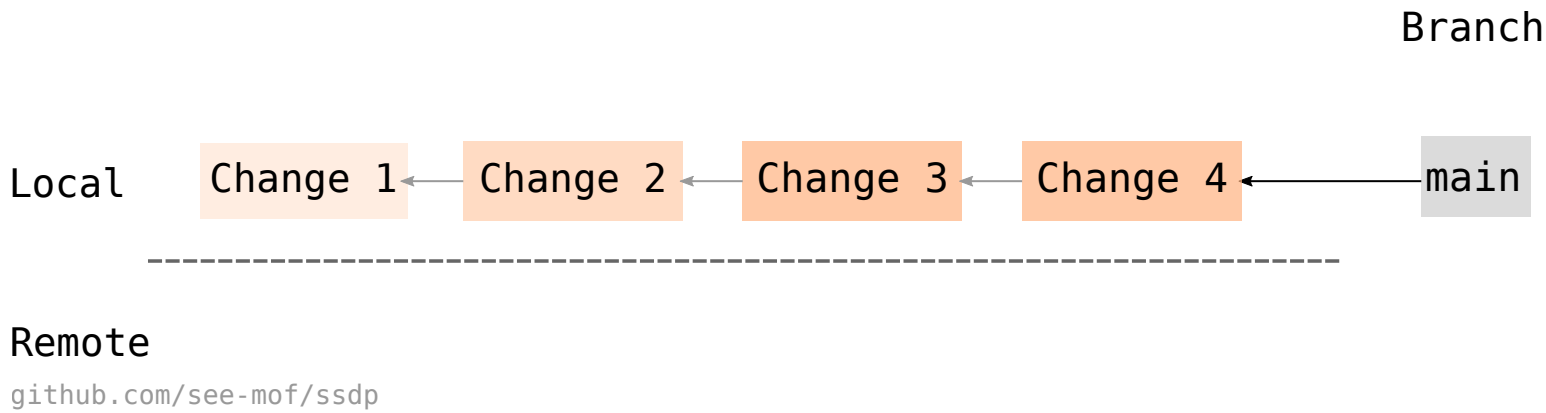

## Diverging branches

- However, typically the branches will *diverge*
- In this case a merge operation is required, which itself is commited to repository
- If the changes to merge affect identical lines in the same files, a conflict occurs, which has to be resolved manually.




## Remotes

- Distributed version control: Every repository can be synchronized with multiple other repositories in different locations.
- A separate repository setup to track the same development is called a **remote**

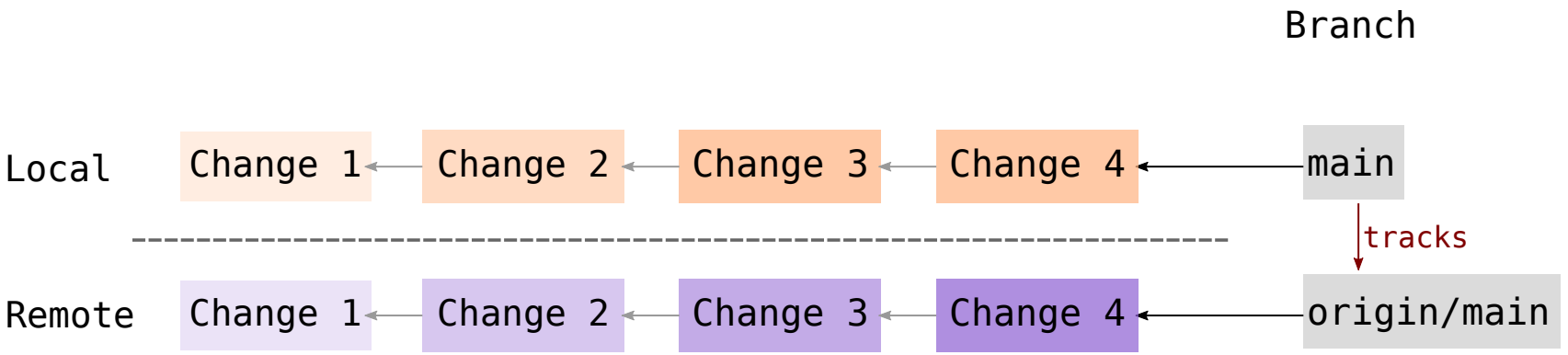



## Adding a remote

```
$ git remote --add <remote_name> https://github.com/see-mof/ssdp
```

- If the remote is a newly created repository it is still empty.

# Updating the remote repository



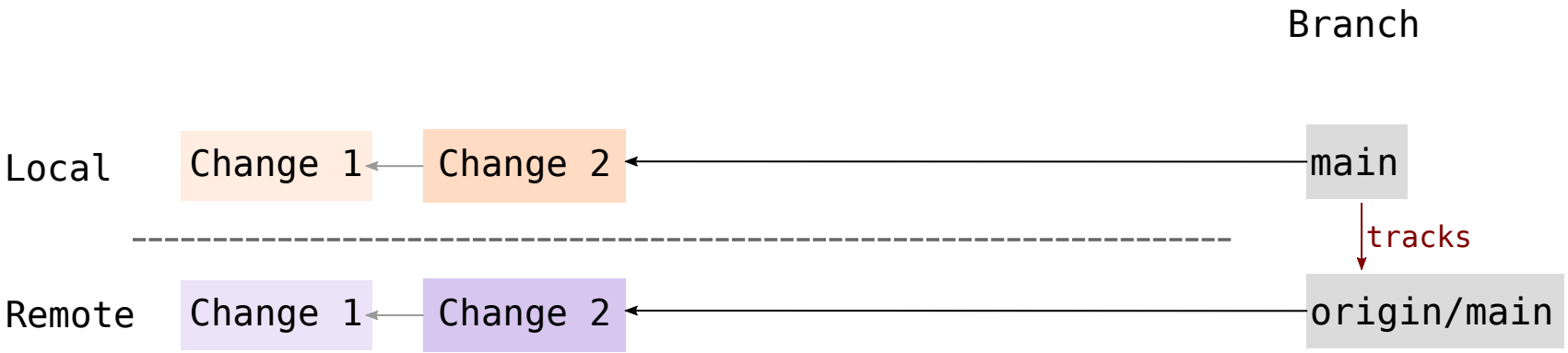

## Publishing local development

- The `-u` option tells `git`, which remote branch `main` should track

```
$ git push -u <remote_name> <branch> # here: git push -u origin main
```

- This becomes the default location for future pushes:

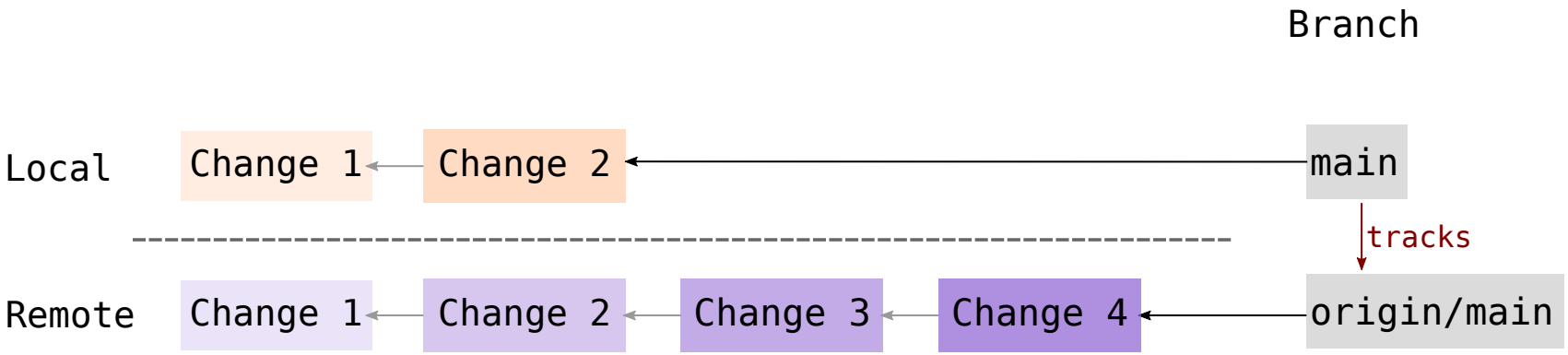
```
$ git push # Same as: git push origin/main
```




## Fetching changes from remote

- `git` keeps local copy of remote branches
- They are denoted by `<remote_name>/<branch_name>`
- `git fetch` updates the local copy of the remote branch:

```
$ git fetch origin main
```

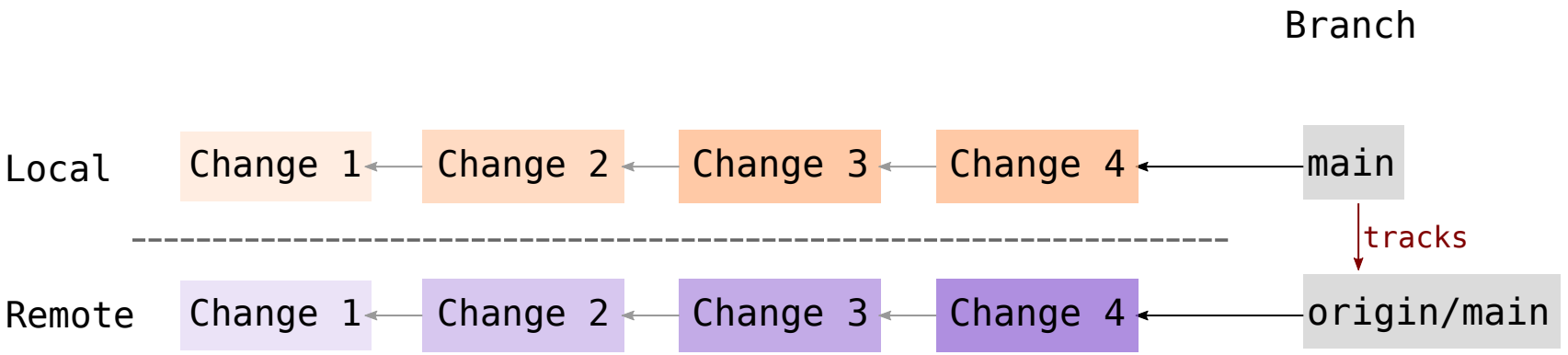



## Fetching changes from remote

- `git` keeps local copy of remote branches
- The are denoted as `<remote_name>/<branch_name>`
- `git fetch <remote_name> <branch_name>` updates the local copy of the remote branch:

```
$ git fetch origin main
```

# Updating your local repository

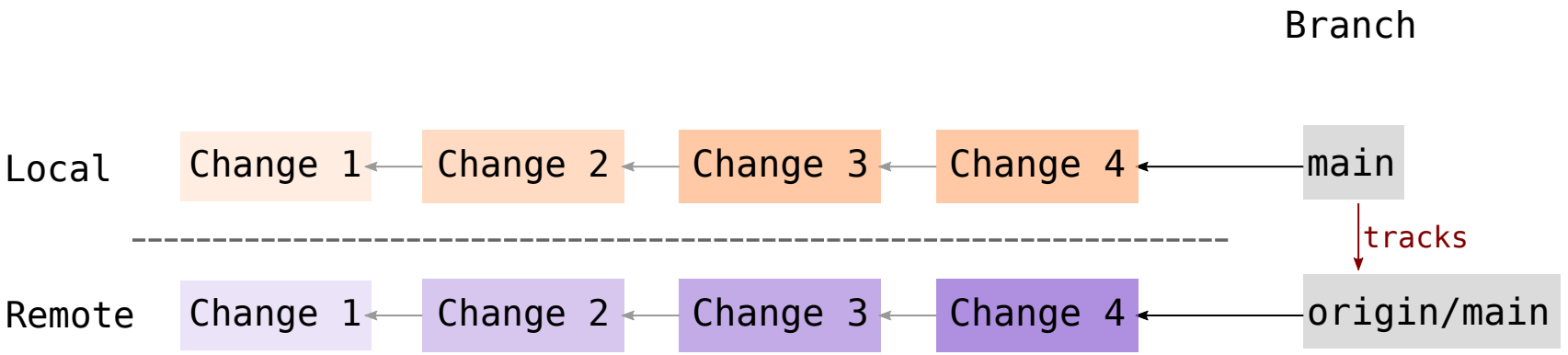



## Merging the local branch

- To update the local branch `main`, merge the local copy of the remote branch

```
$ git merge origin/main
```

# Updating your local repository



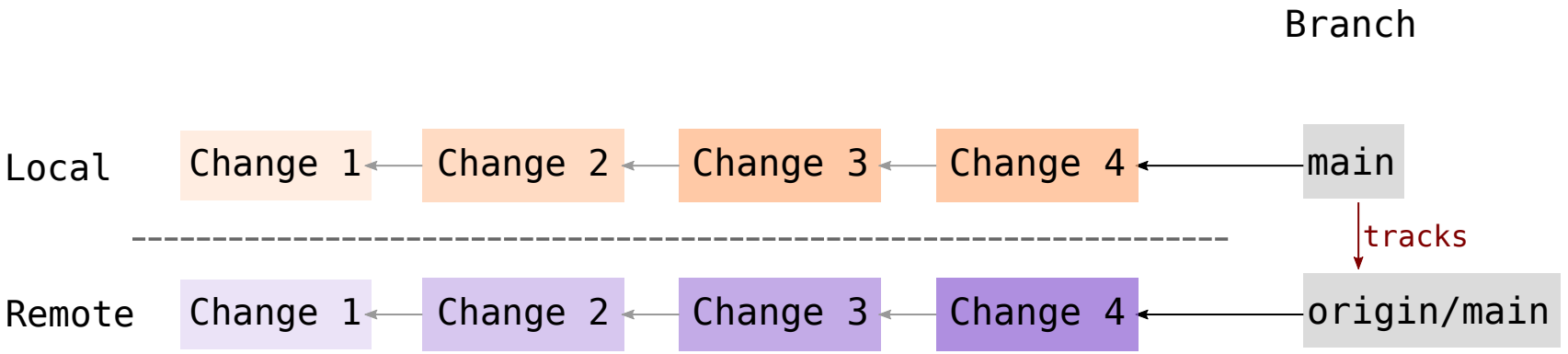

## Pulling changes from remote

- `git pull` combines the `fetch` and `merge` commands:

```
$ git pull origin main
```

or

```
$ git pull # sufficient, if local branch tracks remote
```



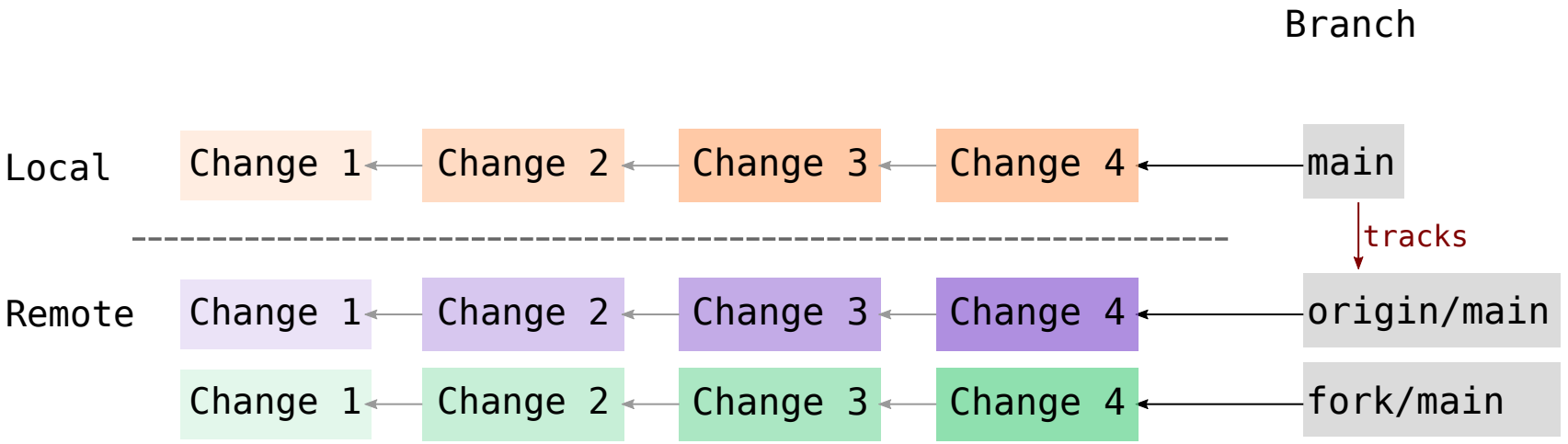

## Cloning a repository

```
$ git clone https://github.com/see-mof/ssdp
```

- This creates a new local repository, adds the target as remote and pulls all branches

# git exercise

- Solve tasks 1 and 2 from the git task sheet
- Time 10 minutes
- Help each other in breakout rooms

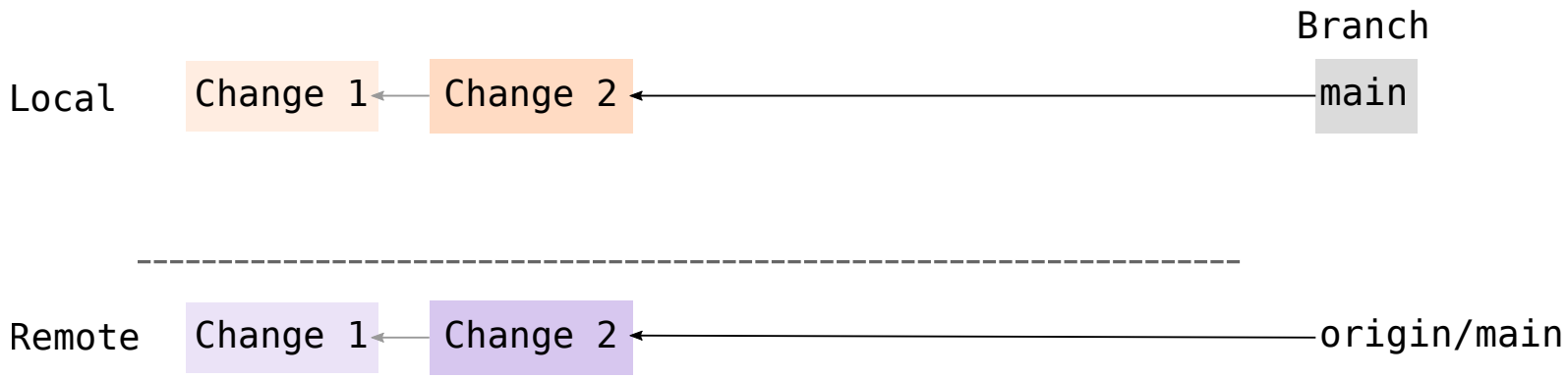



## Multiple remotes

- It is possible to have multiple remotes[8]:

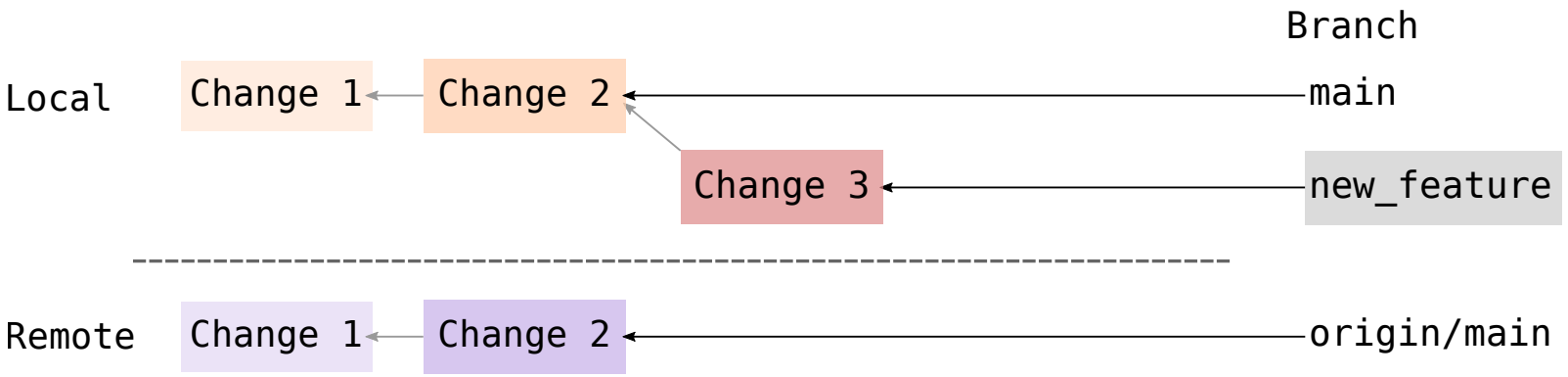
```
$ git add remote fork https://github.com/simonpf/ssdp
$ git push fork
```

[8] For example a personal fork of a public repository.



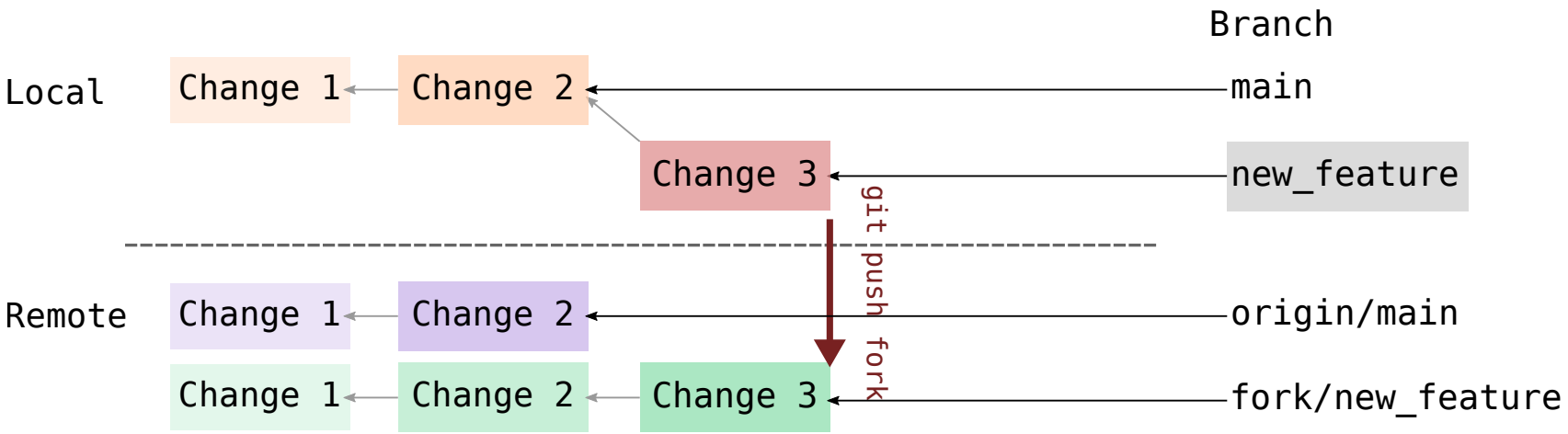

## Contributing to a public repository

1. Clone public repository



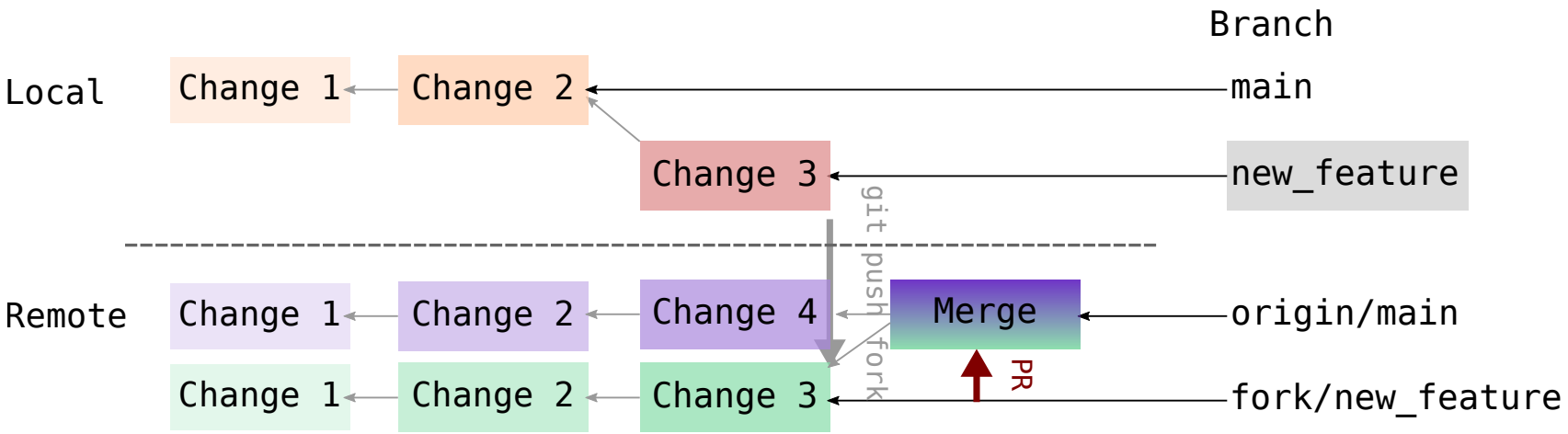

## Contributing to a public repository

1. Clone public repository
2. Add feature in new branch



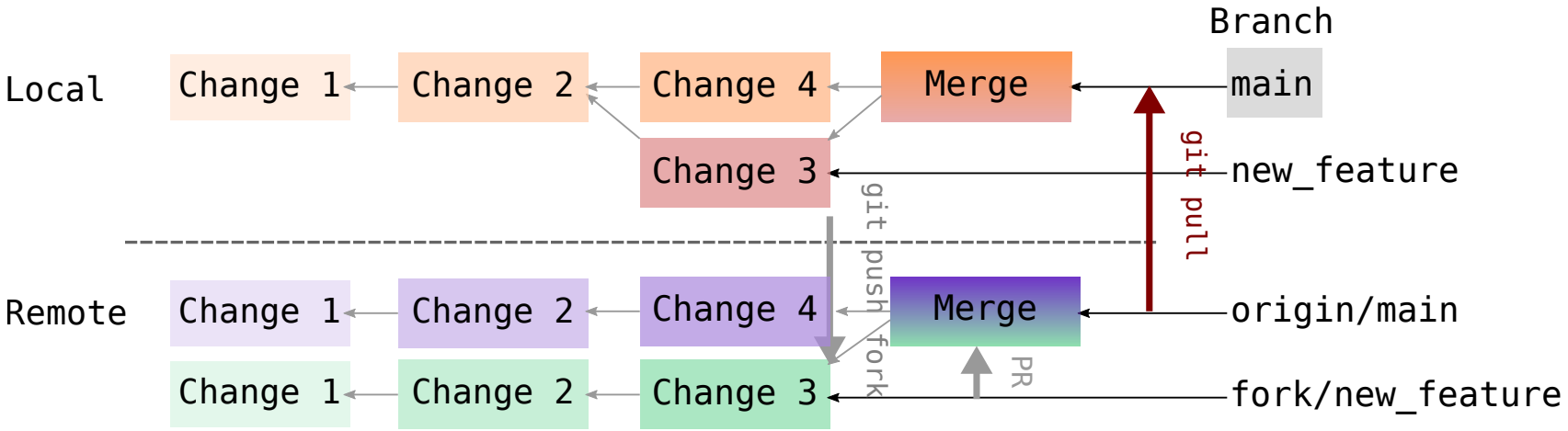

## Contributing to a public repository

1. Clone public repository
2. Add feature in new branch
3. Push new branch to personal fork of public repository



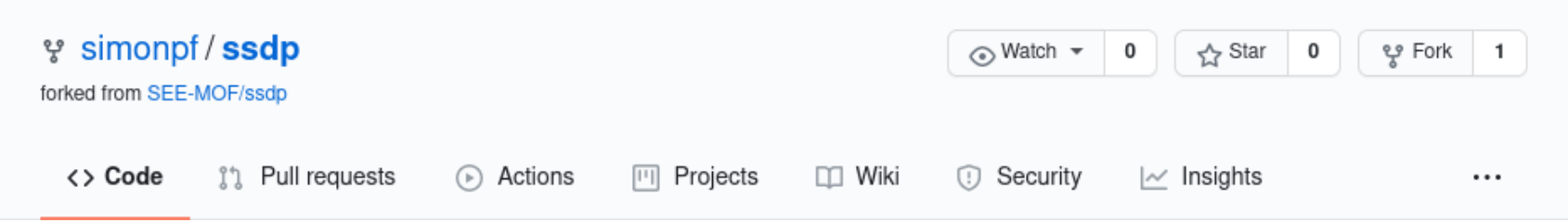

## Contributing to a public repository

1. Clone public repository
2. Add feature in new branch
3. Push new branch to personal fork of public repository
4. Make pull request (PR) from fork




## Contributing to a public repository

1. Clone public repository
2. Add feature in new branch
3. Push new branch to personal fork of public repository
4. Make pull request (PR) from fork
5. Update local `main` branch




## Forks and pull requests

- To fork a repository, click the fork symbol in the upper right corner
- To make a pull request, click on `Pull reqeusts` and then `New pull request`

# git exercise

- Solve tasks 3 and 4 from the git task sheet
- Time: 10 minutes
- Help each other in breakout rooms

# `git status` revisited

```
On branch main

Your branch is up to date with 'origin/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:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  STUDENT_LIST.md

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

# Agenda



1. Course overview
2. Agile software development
3. Version control with git
4. **Project pitches**

# Project requirements



- Final project published a Python package
- Installable via `pip`
- Automated test suite with high coverage (> 90%)
- Automated deployment
- Online documentation hosted on GitHub or Read The Docs.

# Project pitches



- Maximum 5 minutes per presentation
- Ideally more than two persons per project
- If you work alone, please find someone to discuss your progress with and review code