AI TRAINING SERIES

Git

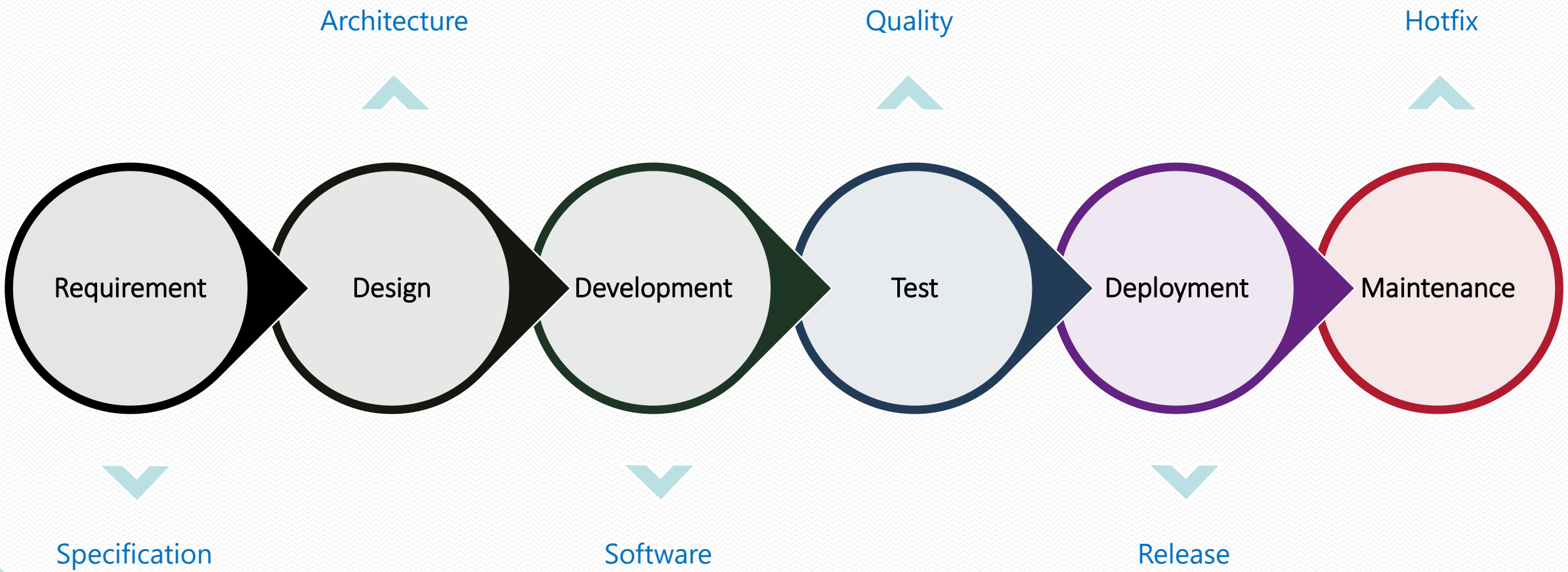
“As more and more artificial intelligence is entering into the world”

CONTENTS

- *Before Start*
  - Why VCS?
- *Git*
  - Getting Started
  - Setting Up a Git Repository
  - Saving Changes
  - Undoing Changes
  - Uploading Commits
  - Syncing
  - Create Branch

Why VCS?

SDL (Software Development Lifecycle)



SDL (Software Development Lifecycle)

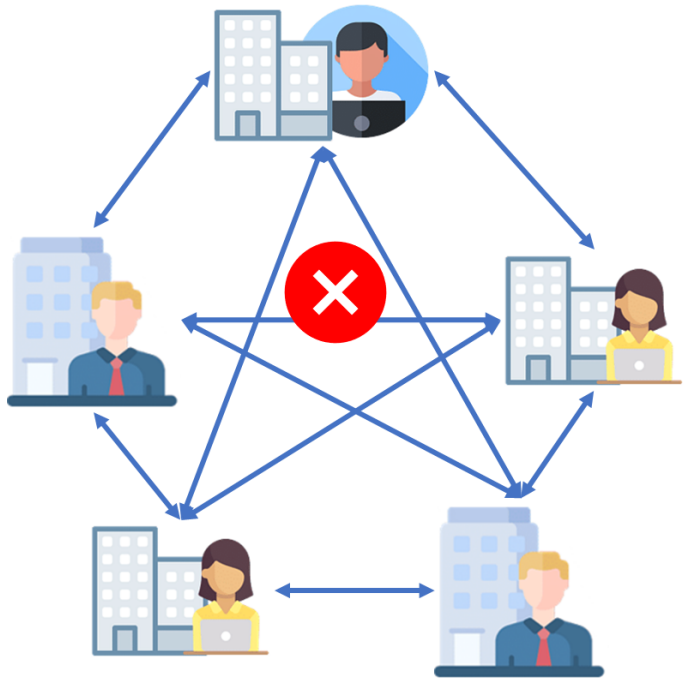
- Benefits
  - Minimize project risks and meet customer expectation
  - Increase visibility of the development process for all stakeholders involved
  - Efficient estimation, planning, and scheduling
  - Systematic software delivery and better customer satisfaction

- Methodologies
  - Waterfall, Iterative, Spiral, Agile, CI (Continuous Integration), etc

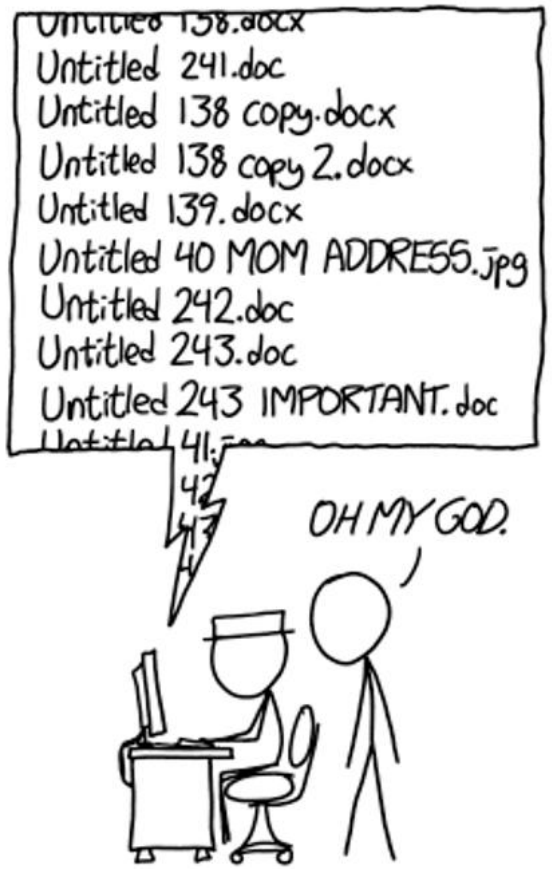
- Next
  - What is the best way to manage SW in SDL? And how?

Why is Software Management needed?

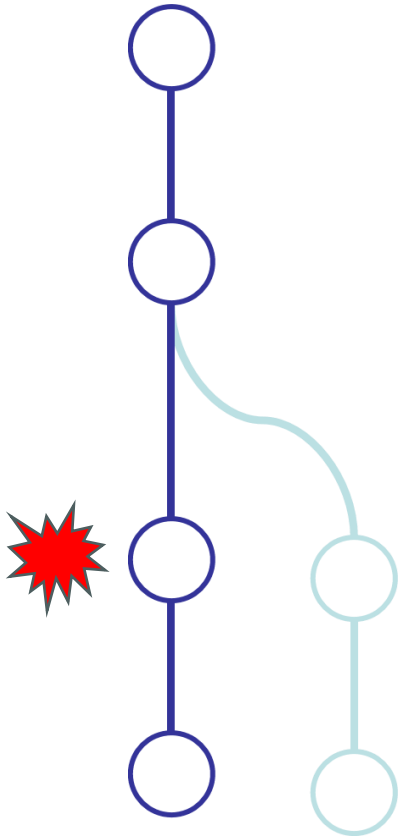
- Collaboration



- Storing/Restoring



- Figure out what happened



Software VC (Version Control)

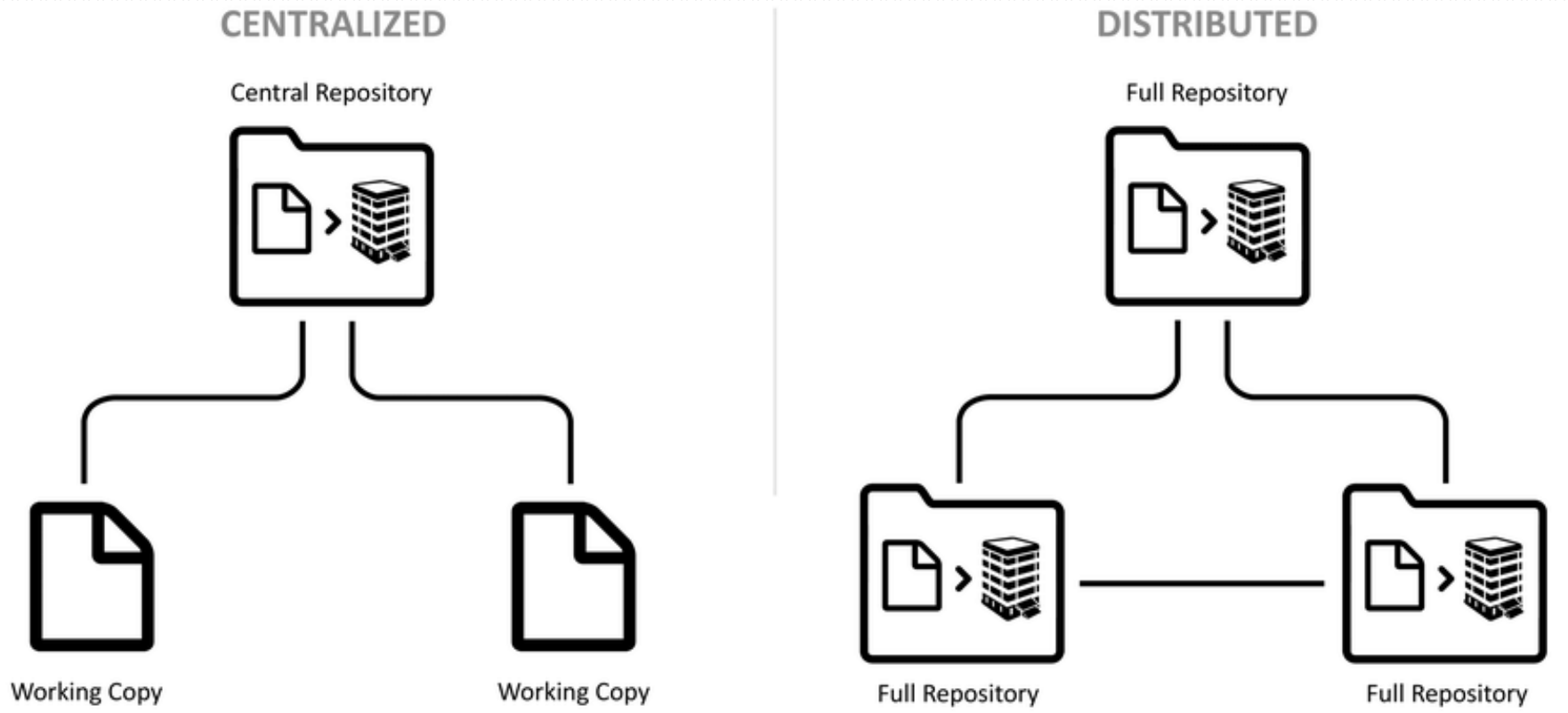
- Everyone on the team is working on the latest version of the file, and all these people can work simultaneously on the same project.

- Track and store changes in files without losing the history of your past changes.

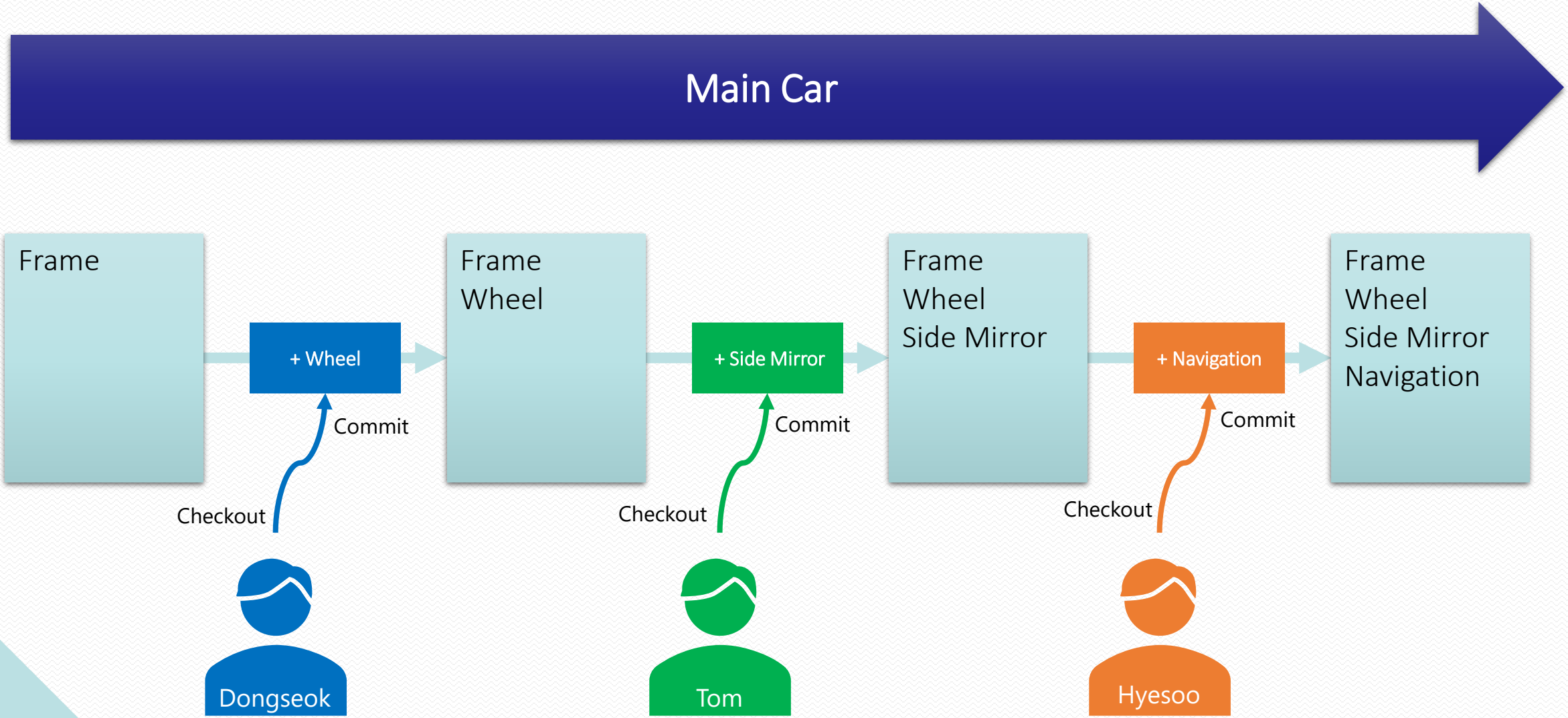
- Records all the made changes to files so a specific version may be rolled if required.

Software VCS (Version Control System)

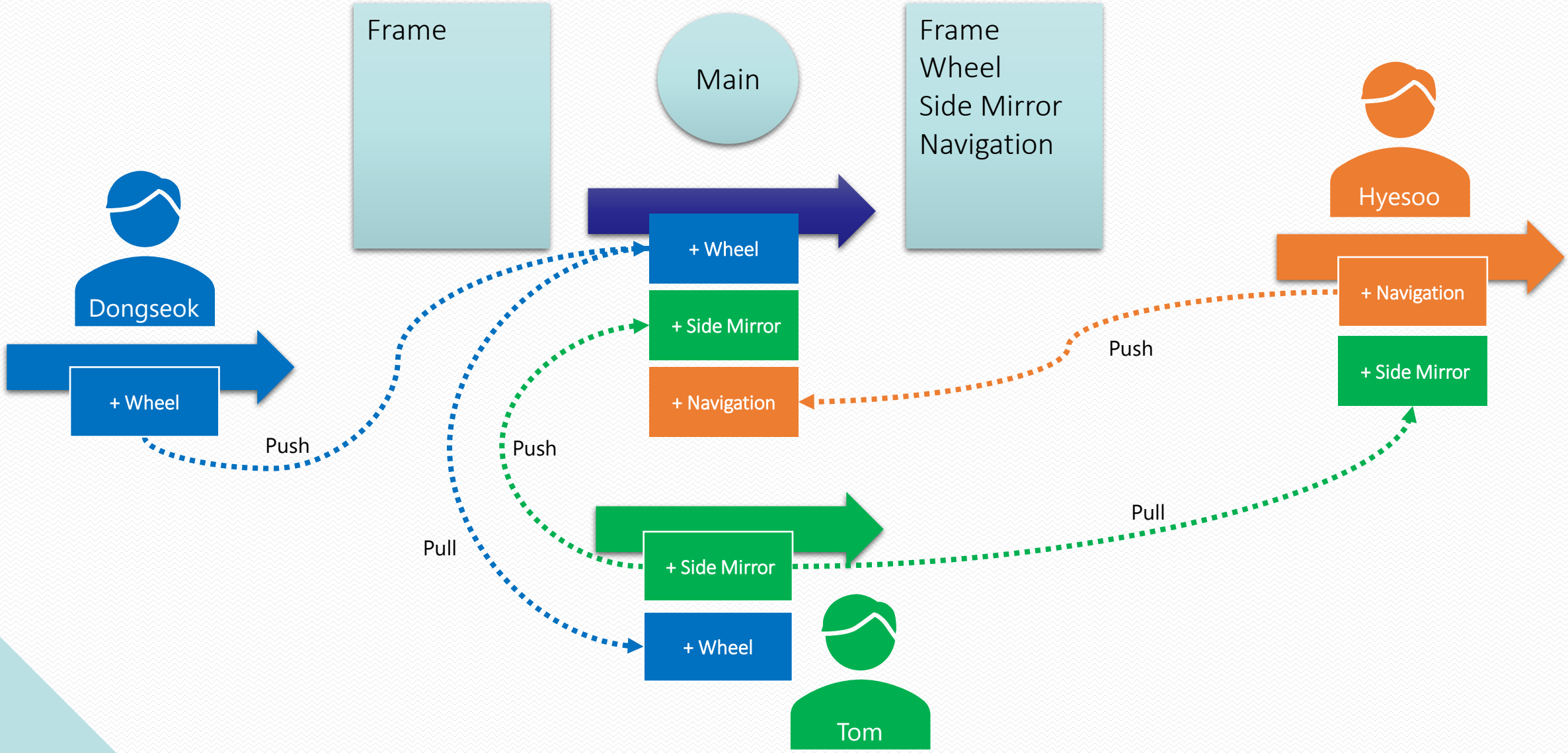
- Version control systems are software tools that helps in recording changes made to files by keeping a track of modifications done in the code.



Centralized VCS



Distributed VCS



Centralized VCS vs Distributed VCS

Centralized VCS	Distributed VCS
Get local copy of source from server, do the changes and commit those changes to central source on server.	Get a local branch as well and have a complete history on it. Client need to push the changes to branch which will then be pushed to server.
Do not provide offline access.	Workable offline as a client copies the entire repository on own local machine.
Slower as every command need to communicate with server.	Faster as mostly user deals with local copy without hitting server every time.
Server is down, developers cannot work.	Server is down, developer can work using own local copies.

Git

What is Git?

- A free and open-source version control system
- Originally created by Linus Torvalds in 2005.
- Distributed VCS
- Developer has the full history of their code repository locally. This makes the initial clone of the repository slower, but subsequent operations such as commit, blame, diff, merge, and log dramatically faster.
- The most widely used version control system in the world today and is considered the modern standard for software development.

Install Git

- Windows
  - Download an installer for Git
    https://git-scm.com/download/win

- Linux (ex, Ubuntu)
  - Run following commands in terminal

```
$ sudo apt-get update
$ sudo apt-get install git
```

Getting Started

- Check if Git is properly installed:

```
$ git --version
```

- Make sure to set Git with your name and email address using the following commands on the command-line with your name and email address:

```
$ git config --global user.name "FirstName LastName"
$ git config --global user.email "email@example.com"
```

Git will use this information when you work on a project.

Practice : Getting Started

- Goal
  - Install Git
  - Set name & email

```
$ sudo apt-get update
$ sudo apt-get install git

$ git config --global user.name "FirstName LastName"
$ git config --global user.email email@example.com

$ cat ~/.gitconfig
```

Setting Up a Git Repository – git init

- A Git repository is a central storage location for managing and tracking changes in files and directories.
- The "git init" command creates a new Git repository.

- Non-bare repository
  - Non-bare repository is generally used by individual developer
  - It includes both the versioned data and a working directory.

```
$ git init <Repo Name>
```

Create non-bare repository named <Repo Name>.

- Bare repository
  - A server-side repository that does not have a working directory.
  - It only contains the versioned data and the Git history.

```
$ git init --bare <Repo Name>
```

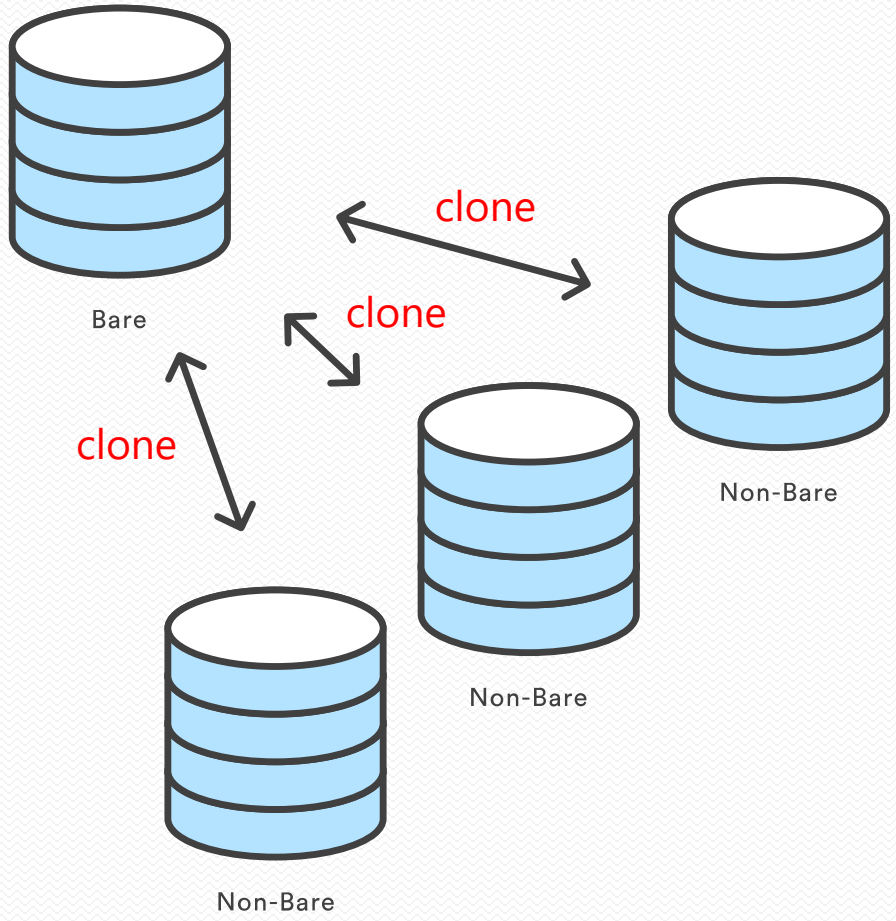
Create bare repository named <Repo Name>.

Setting Up a Git Repository – git clone

- The "git clone" command is used to target an existing repository and create a clone, or copy of the target repository.

```
$ git clone <Repo URL> -b <Branch or Tag Name> <Directory>
```

Clone Branch called <Branch or Tag Name> from the repository located at <Repository GitHub URL> into the folder called <Directory> on the local machine.



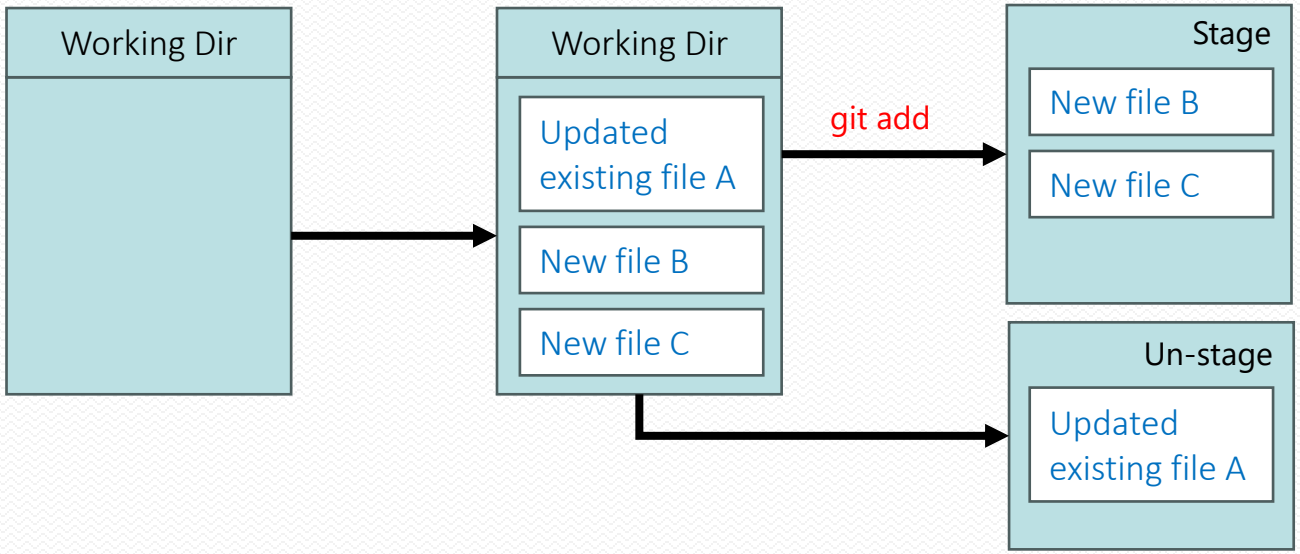
Practice : Setting Up a Git Repository

- Goal
  - Create “git-training” directory in your home and move directory to “git-training” directory.
  - Create bare repository named “my-project.git”
  - Clone the bare repository created with “my-project.git” into “project-x”

```
$ mkdir ~/git-training
$ cd ~/git-training

$ git init --bare my-project.git

$ git clone ~/git-training/my-project.git project-x
```

Saving Changes – git add

- The "git add" command adds a change in the working directory to the staging area.

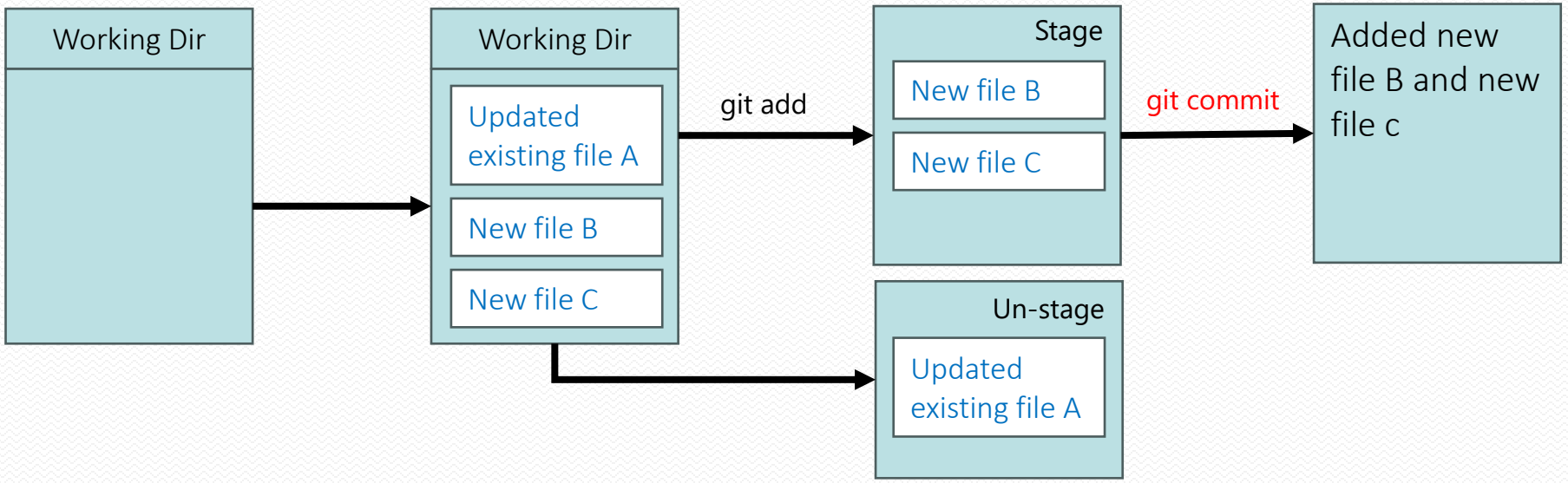



```
$ git add <File name 1> <File name 2> <Directory 1>
```

Allow <File name 1>, <File name 2> <Directory 1> to stage files to be committed.

Saving Changes – git commit

- The “git commit” command captures a snapshot of the project's currently staged changes.




```
$ git commit -m <Comment>
```

Creates new commit with commit message called <Comment>

Saving Changes – git status & git log

- The “git status” command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven’t, and which files aren’t being tracked by Git.

```
$ git status
```

- The “git log” command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes.

```
$ git log
```

Ignoring Unwanted Files – .gitignore

- The “.gitignore” file can ignore certain types of files that you don't need to track.
- Create and open .gitignore under your git project
- The .gitignore uses globbing patterns to match against file names.

```
$ vi .gitignore

*.o
*.a
*.mod
```

Object files (*.o), Library files (*.a) and Module files (*.mod) won’t be tracked any more in the git project.

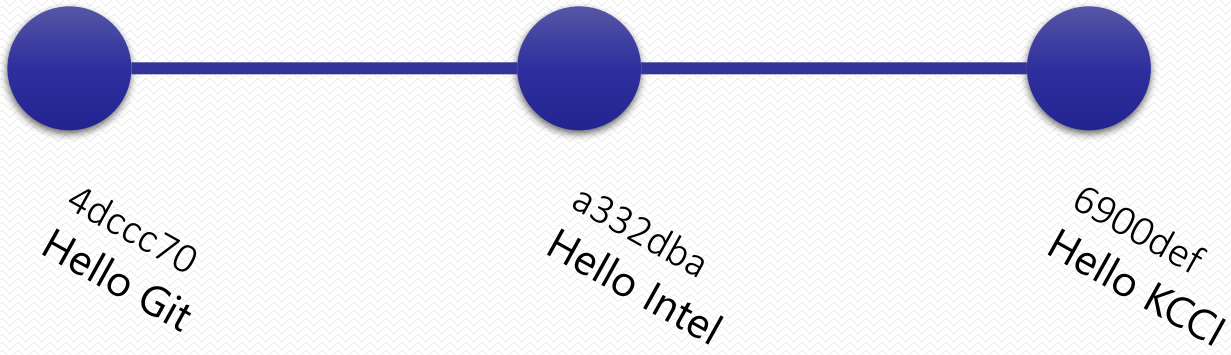
Practice : Saving Changes

- Goal
  - Go to your cloned project (ex, project-x).
  - Implement a code to display “Hello Git” prompt using any languages (c, cpp, python, shell, etc)
  - Create .gitignore not to track object file.
  - Create a commit
  - Repeat above steps to display two more prompts (ex, “Hello Intel”, “Hello KCCI”)

```
$ cd ~/git-training/project-x

$ vi .gitignore

$ vi hello.c

$ git add hello.c .gitignore
$ git status
$ git commit
```

Practice : Saving Changes

```
$ git log --oneline

6900def (HEAD -> master) hello KCCI
a332dba hello Intel
4dccc70 hello Git
```

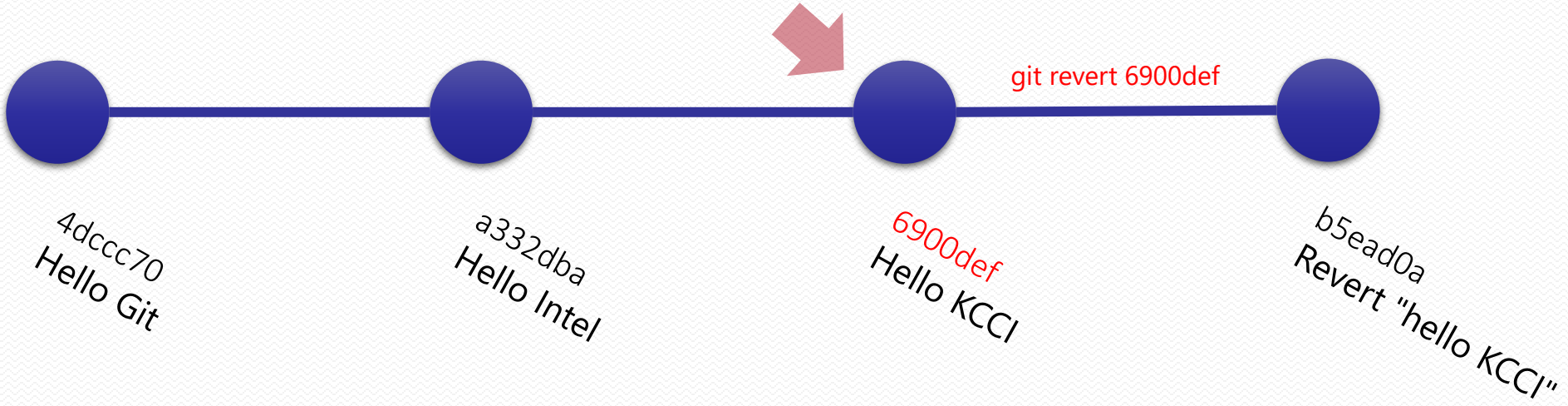


Undoing Changes – git revert

- The “git revert” command inverses the changes from that commit and create a new "revert commit".

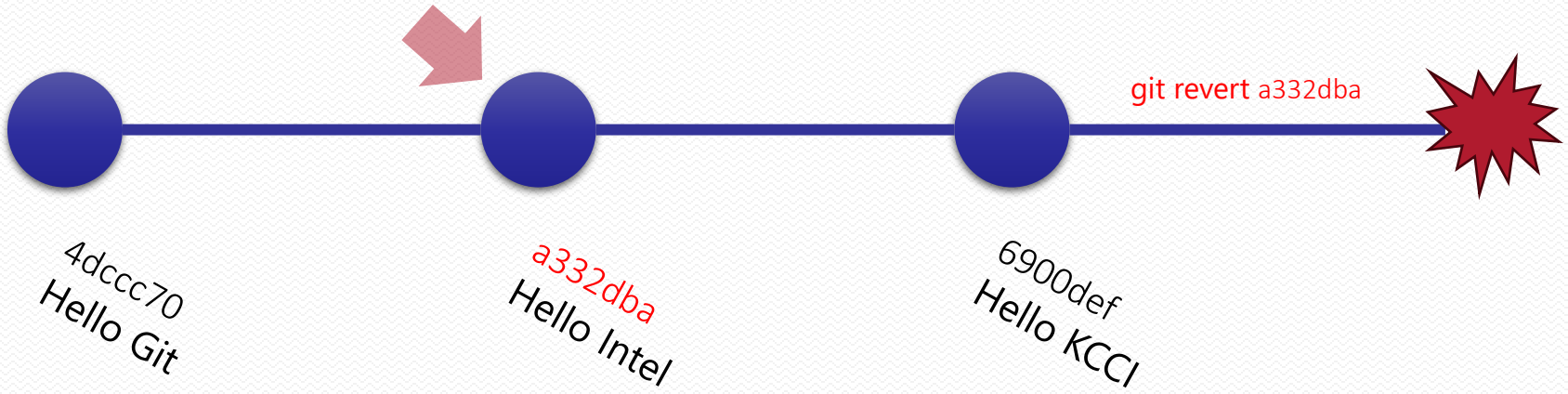
```
$ git revert <Commit ID>
```

Revert <Commit ID> and will create a new “revert commit”

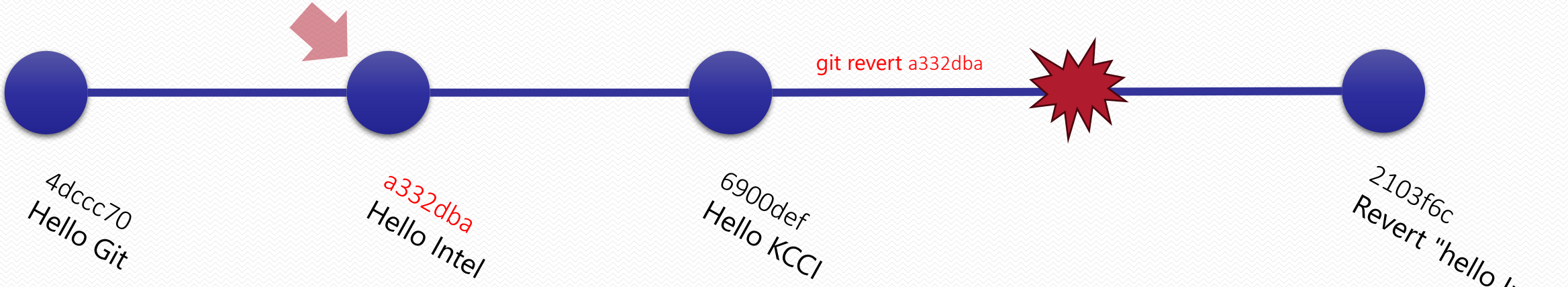


Undoing Changes – git revert (conflict)

- Conflicts can occur in various places, such as applying patches or merging source code, and can also occur during the process of reverting.



Undoing Changes – git revert (conflict)




```
$ git status
On branch master
You are currently reverting commit a332dba.
  (fix conflicts and run "git revert --continue")
  (use "git revert --skip" to skip this patch)
  (use "git revert --abort" to cancel the revert operation)

Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add/rm <file>..." as appropriate to mark resolution)
        deleted by them: hello
        both modified:   hello.c

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

```
#include <stdio.h>

int main () {
        printf("Hello Git\n");
<<<<<<< HEAD
        printf("Hello Intel\n");
        printf("Hello KCCI\n");
=======
>>>>>>> parent of a332dba... hello intel
        return 0;
}
```

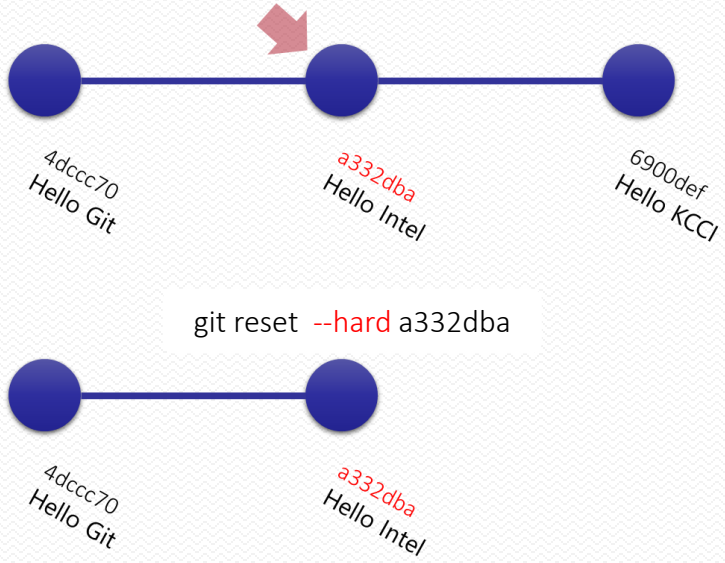
Git shows where conflicts occur using indicators

- Fix conflicts
- git add
- git commit

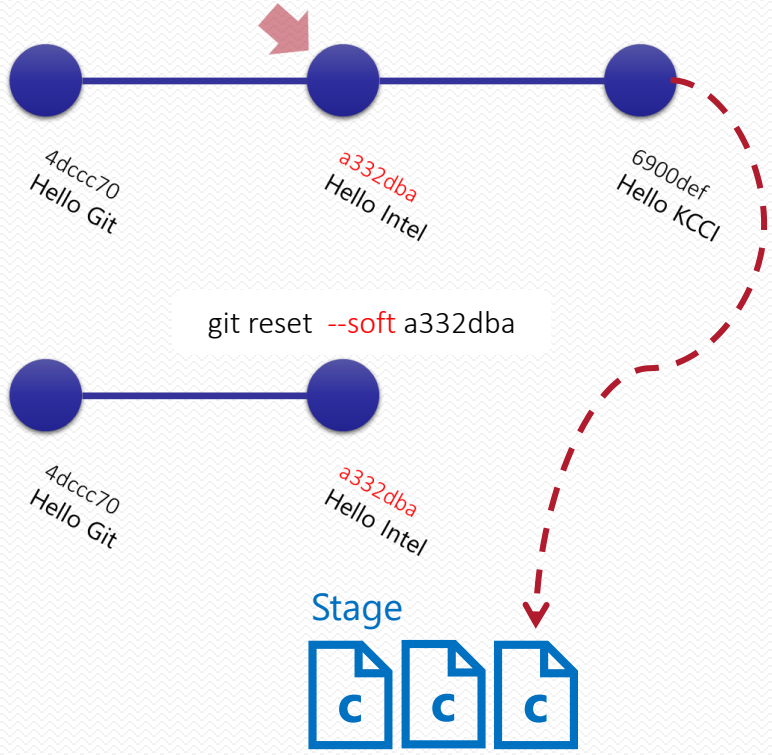
Undoing Changes – git reset

- The "git reset" command is used to undo local changes to the state of a Git repo.

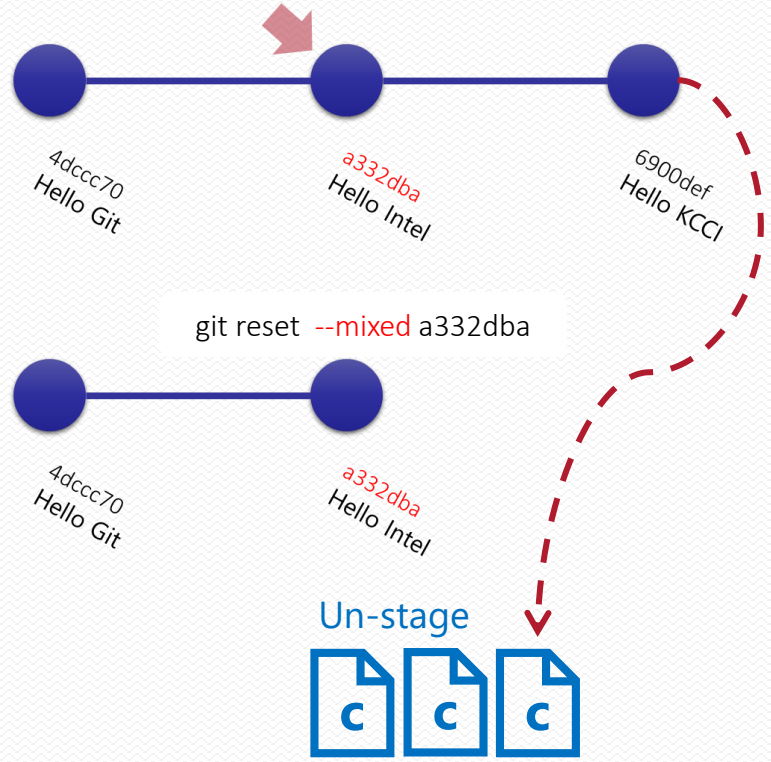
$ git reset --hard <Commit ID>



$ git reset --soft <Commit ID>

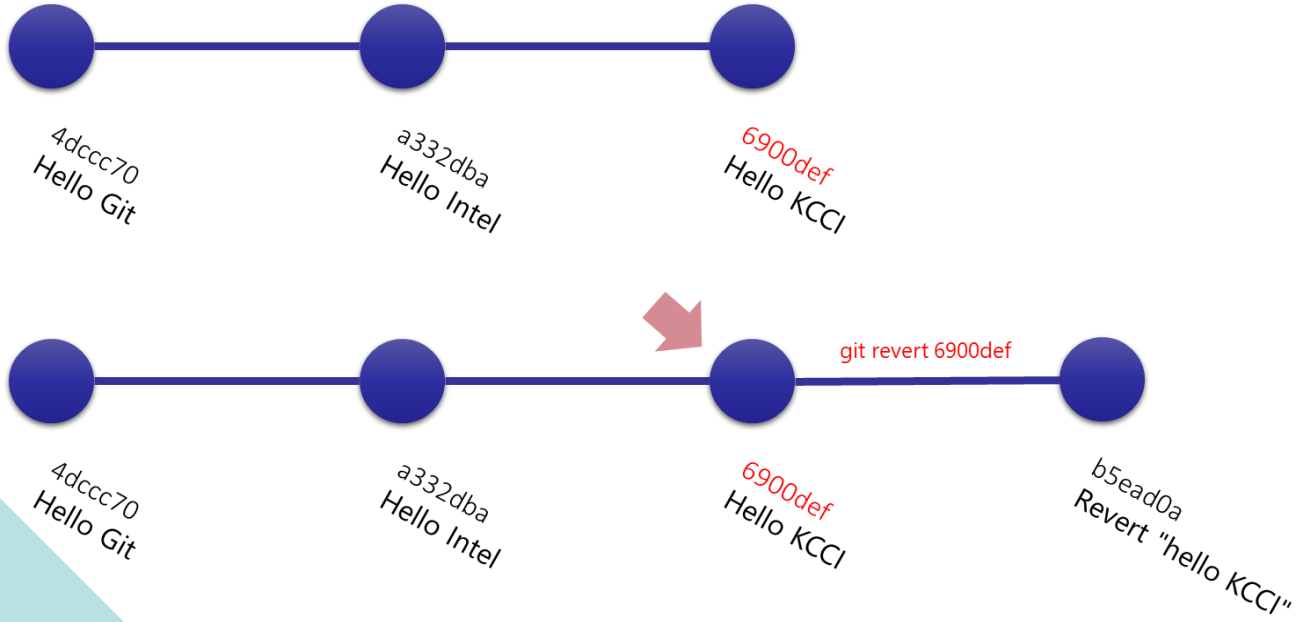


$ git reset --mixed <Commit ID>

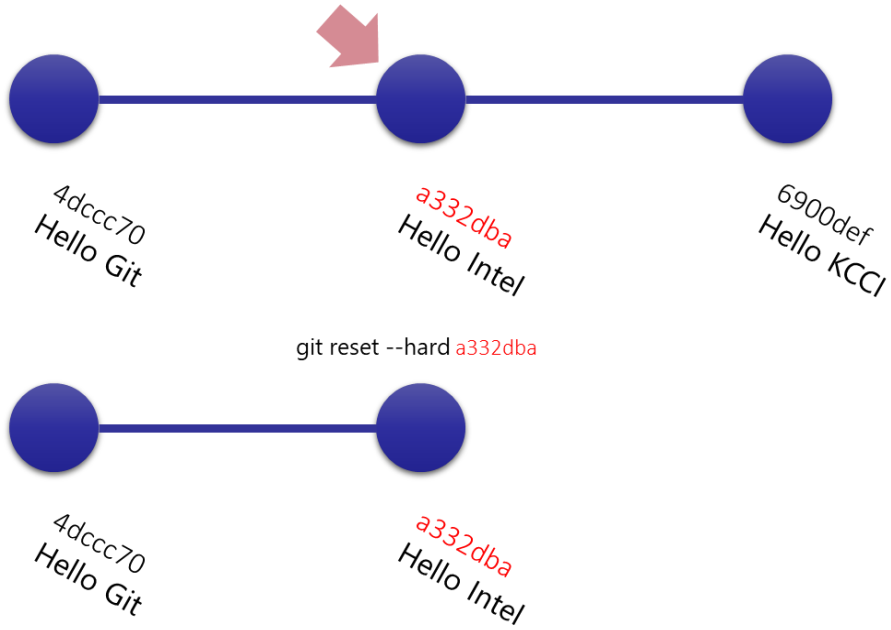


git revert vs git reset

- git revert
  - Go back to previous states while creating a new commit.
  - Better when you want to undo changes on a public branch, for safety.

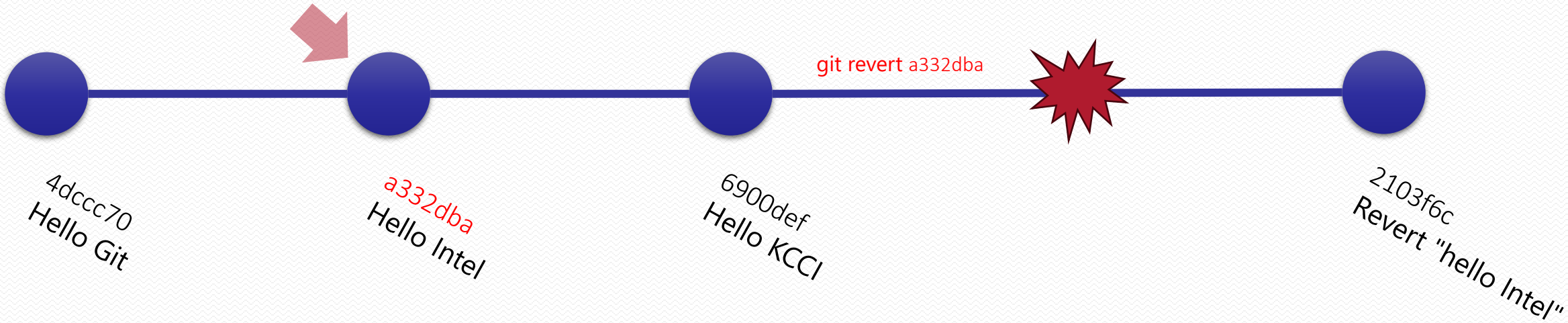


- git reset
  - Go back to the previous commits, but can't create a new commit.
  - Better when undoing changes on a private branch.

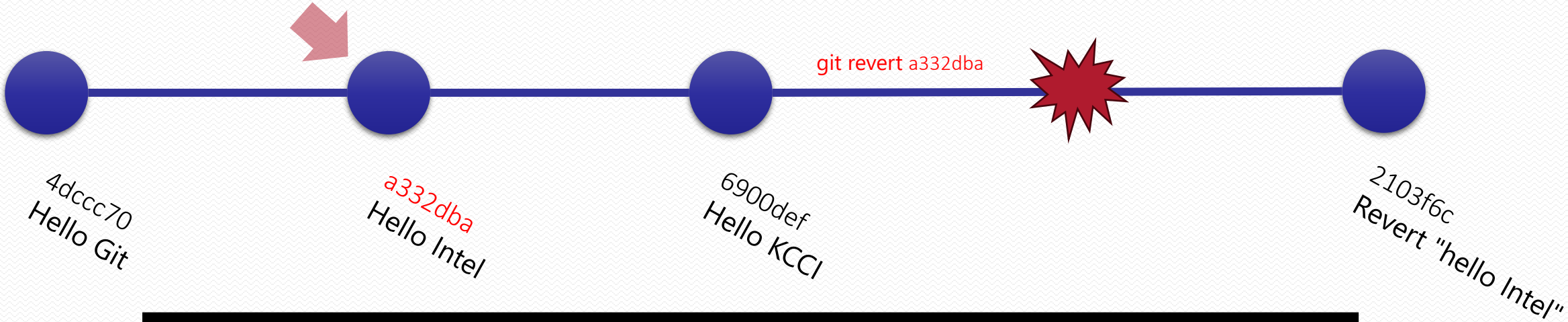


Practice : Undoing Changes

- Goal
  - Go to your cloned project (ex, project-x).
  - Revert the 2nd commits
  - If conflict occurred, try to fix conflicts.



Practice : Undoing Changes




```
$ cd ~/git-training/project-x

$ git revert a332dba

$ vi hello.c

$ git add hello.c
$ git commit
```

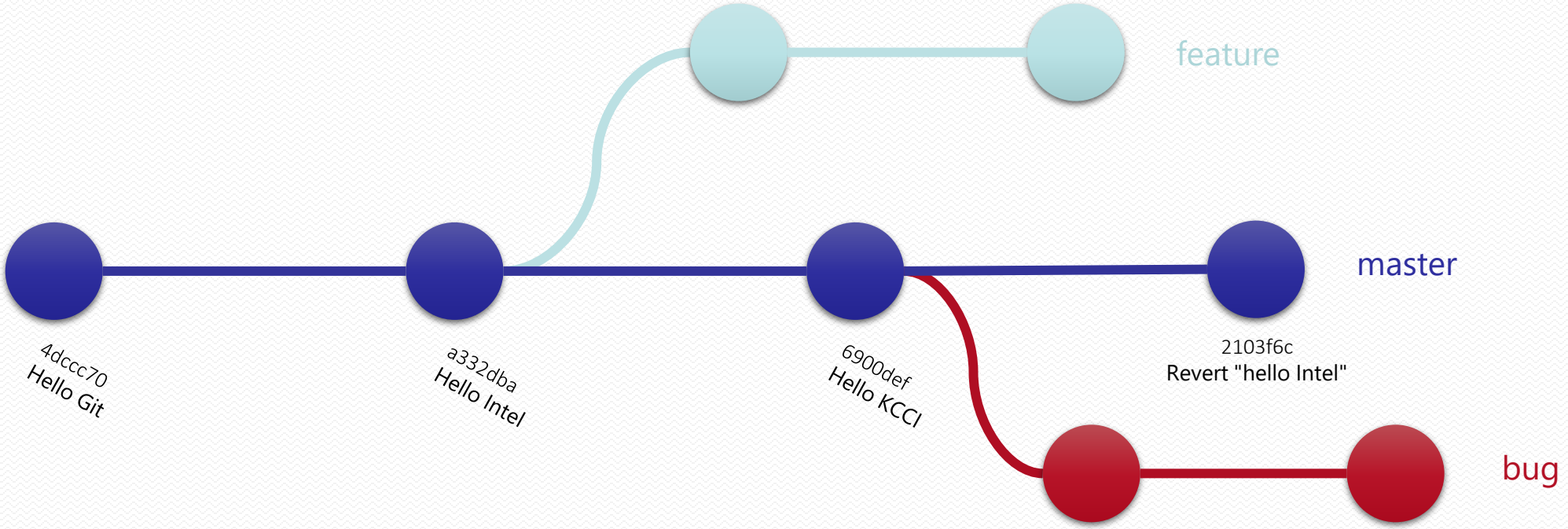
Uploading Commits – git remote

- The “git remote” command helps for you to create, view, and delete connections to other repositories.
- Remote connections are more like bookmarks rather than direct links into other repositories.

```
$ git remote -v
```

Uploading Commits – Branches

- Branch is a new or separate version of the main/local repository for new project, new feature, bug fix and so on.



Uploading Commits – git push

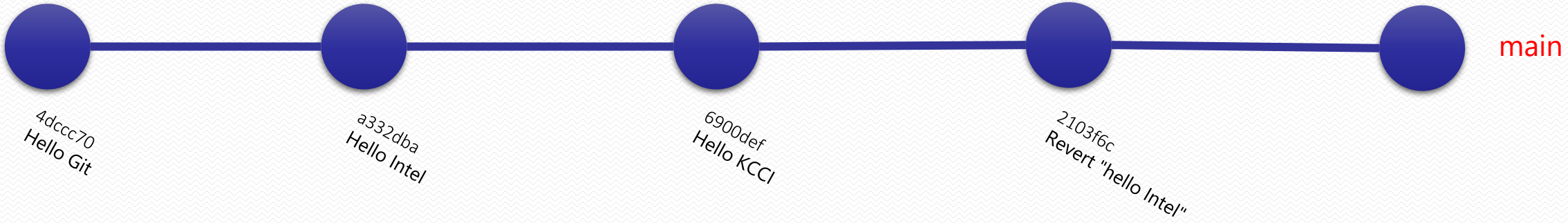
- The “git push” command is used to upload local repository content to a remote repository.
- Pushing is how you transfer commits from your local repository to a remote repo.

```
$ git push <Remote> HEAD:<Branch Name>
```

- Push your local changes called HEAD: to specific branch (<Branch Name>) in the remote repository (<remote repository>)
- You can use commit id or local branch name instead of HEAD:

Practice : Uploading Commits

- Goal
  - Upload your changes to “main” branch in bare project




```
$ cd ~/git-training/project-x

$ git remote -v
$ git push origin HEAD:main
```

Syncing – git fetch

- The “git fetch” command downloads commits, files, and refs from a remote repository into your local repo.

```
$ git fetch <Remote>
```

  - Fetch all of the branches from <Remote> the repository.
  - This also downloads all of the required commits and files from the other repository.

- Fetched content has to be explicitly checked out using the git checkout command. Or if it needed, you can merge fetched content.

Practice : Syncing

- Goal
  - Go to your cloned project (ex, project-x).
  - Fetch “main” branch from the bare repository.
  - Checkout “main” branch
    - Hint) git branch -a

```
$ cd ~/git-training/project-x

$ git fetch origin

$ git branch -a

$ git checkout remotes/origin/main

$ git log
```

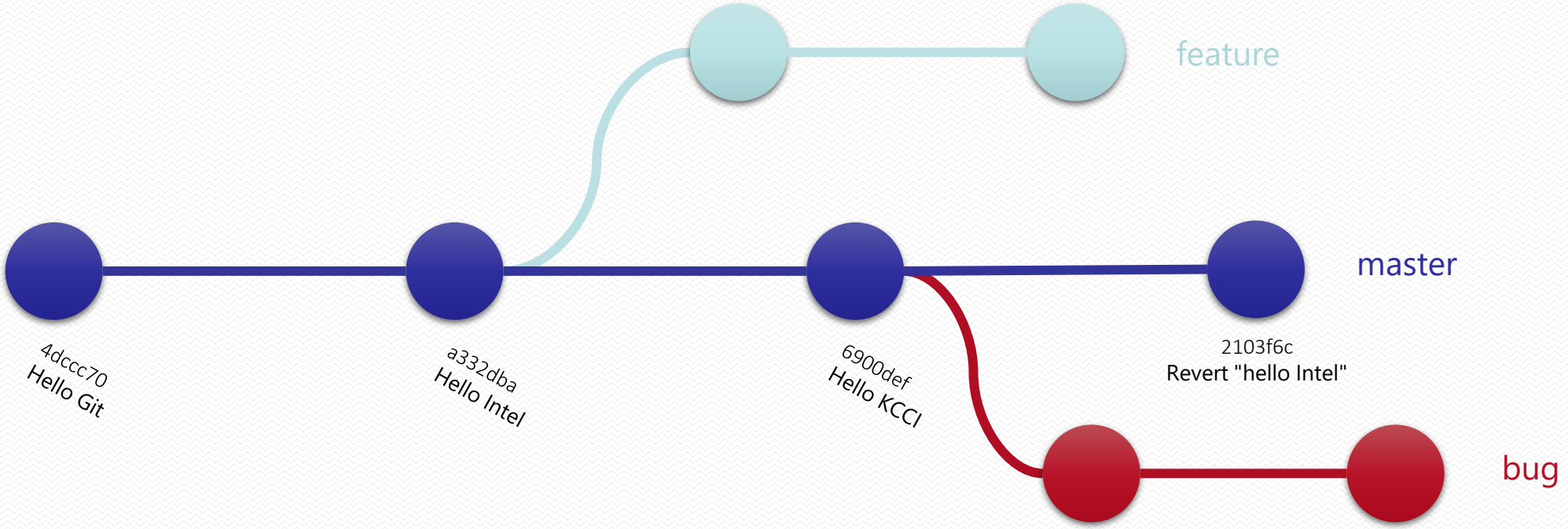
Practice : Syncing

- Goal
  - Clone “main” branch from the bare repository created with “my_project.git” into new repository named “project-y”

```
$ cd ~/git-training

$ git clone ~/git-training/my-project.git -b main project-y

$ cd ~/git-training/project-y

$ git log
```

Creating Branches

- Branch is a new or separate version of the main/local repository for new project, new feature, bug fix and so on.

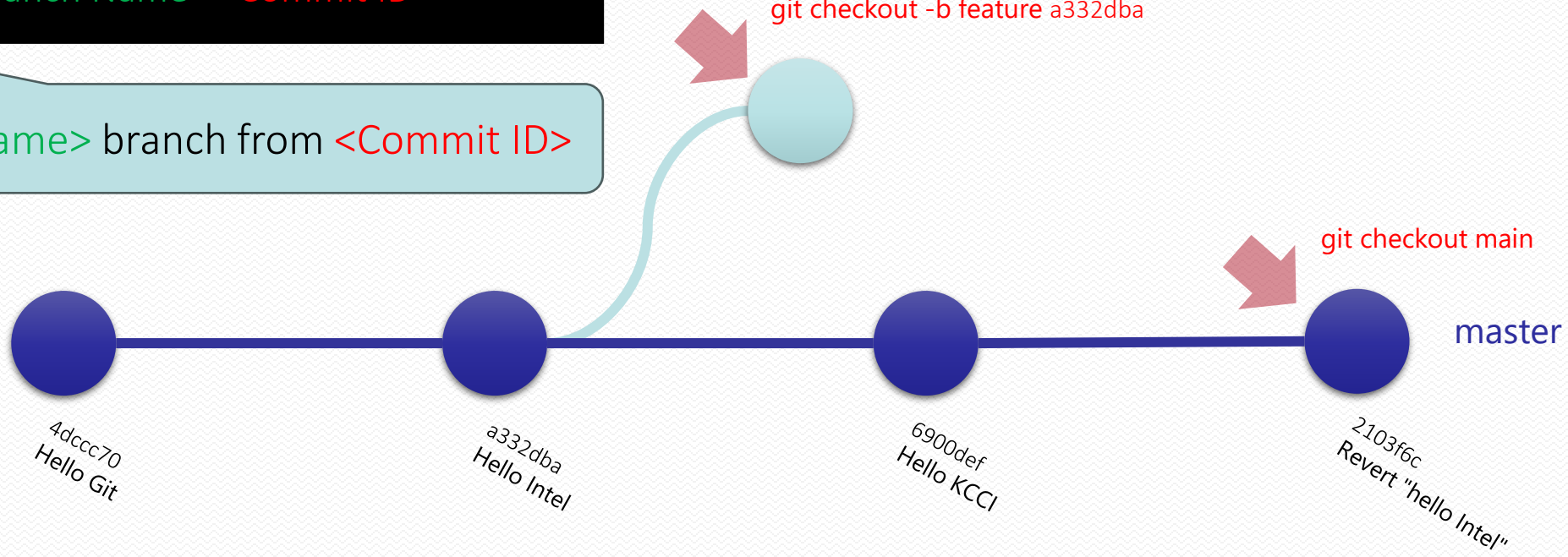


Creating Branch – git checkout

- The “git checkout” create a new branch or switch an existed branch.

```
$ git checkout -b <Branch Name> <Commit ID>
```

Create <Branch Name> branch from <Commit ID>



Switch <Branch Name>

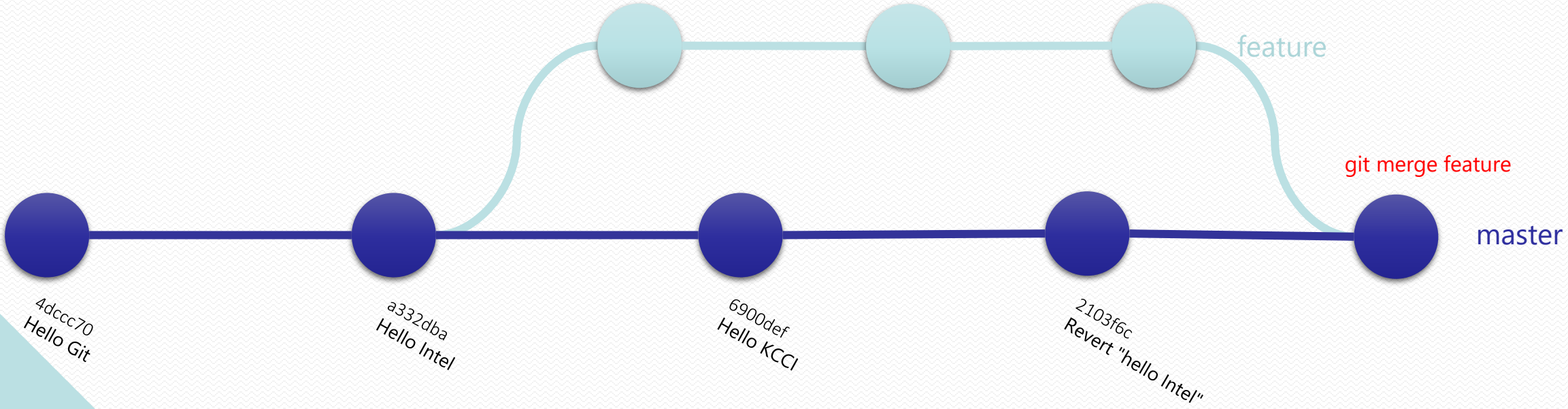
```
$ git checkout <Branch Name>
```

Creating Branch – git merge

- The “git merge” command combines multiple sequences of commits into current branch for one unified history.

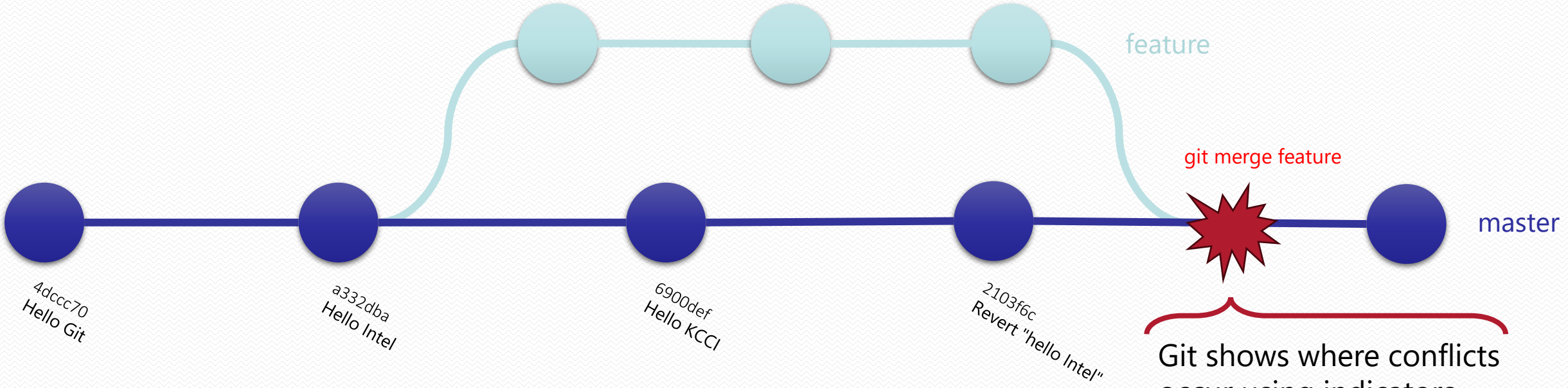
```
$ git merge <Branch Name>
```

Merge <Branch Name> branch to current branch



Creating Branch – git merge (conflict)

- Conflicts can occur in various places, such as applying patches or merging source code.

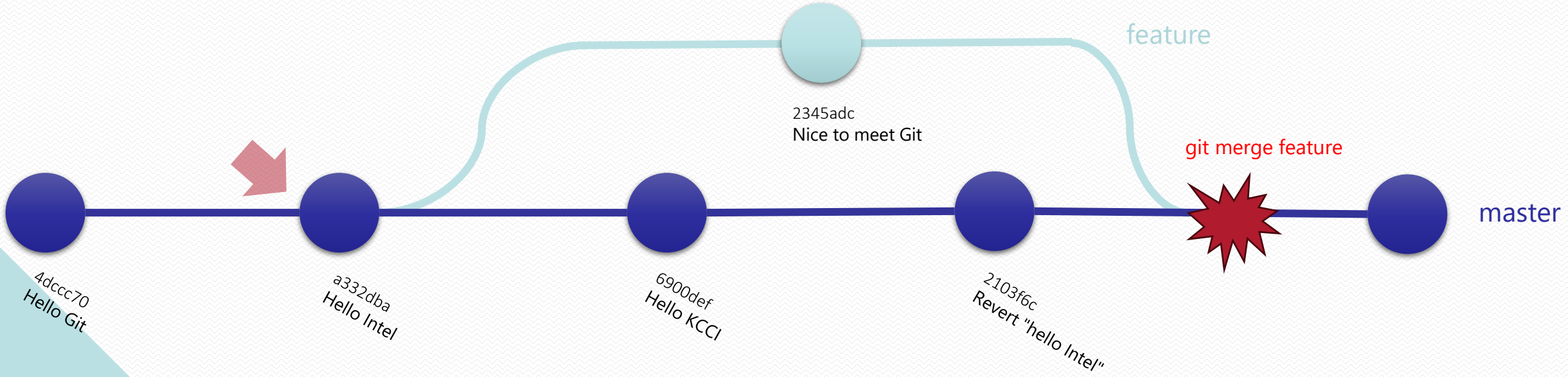


Git shows where conflicts occur using indicators

- Fix conflicts
- git add
- git commit

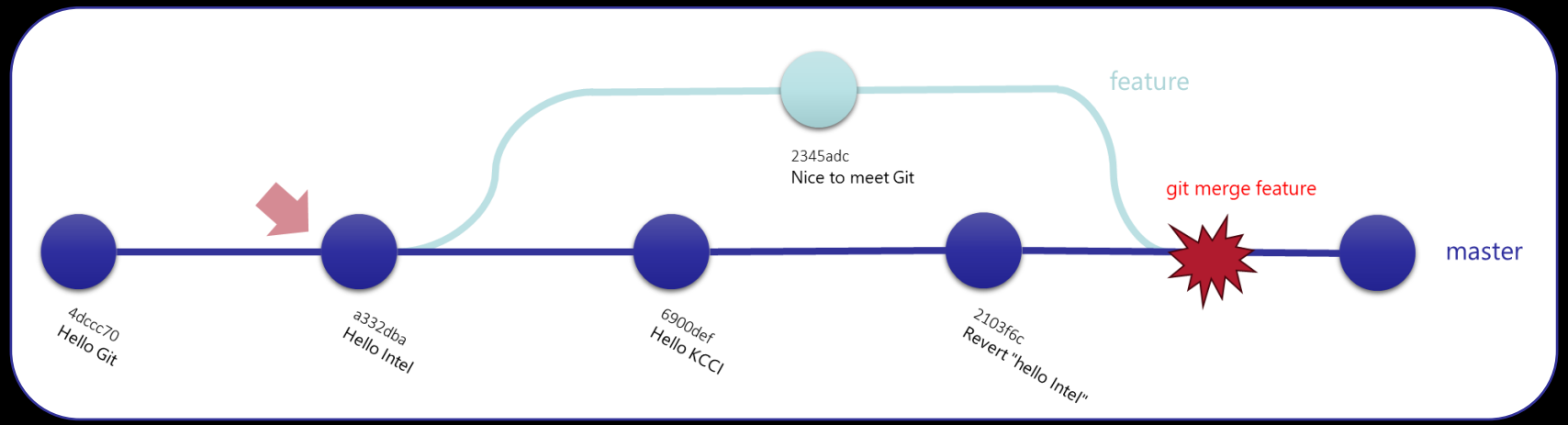
Practice : Uploading Commits

- Goal
  - Create new branch named “feature” from the 2nd commit.
  - Add codes to display “Nice to meet Git” prompt into an existed code.
  - Create new commit in “feature” branch
  - Checkout “master” branch and merge “feature” branch to “master” branch.
  - Upload “master” branch to “main” branch in bare project



Practice : Uploading Commits

```
$ cd ~/git-training/project-x

$ git log --oneline

$ git checkout -b feature a332dba

$ vi hello.c
$ git add .
$ git commit

$ git checkout master
$ git merge feature

$ vi hello.c
$ git add .
$ git commit

$ git remote -v
$ git push origin HEAD:main
```



git log graph for Graphic Overview

- The git log graph command creates a graphic overview of how a developer's various development pipelines have branched and merged over time.

```
$ git log --graph --decorate --abbrev-commit --all --pretty=oneline
*   6d80fd5 (HEAD -> master) Merge branch 'test2'
|\
| * 92d35cf (test2) hello.cpp
* | 09d8d5a (origin/main) hello kcci
* | 0916fd8 (test) hello intel
|/
* d44e0b4 hello git
```

Next? GitHub

THANK YOU