ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Unix/Linux, C basics, GIT

Francesco Conti

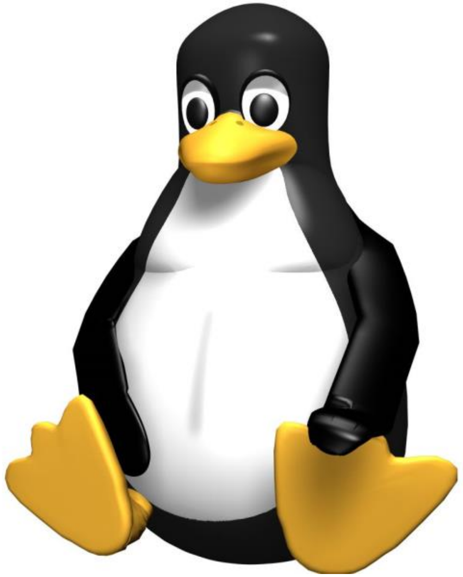
Manuele Rusci

Giuseppe Tagliavini

What is Linux?

Linux is an operating system

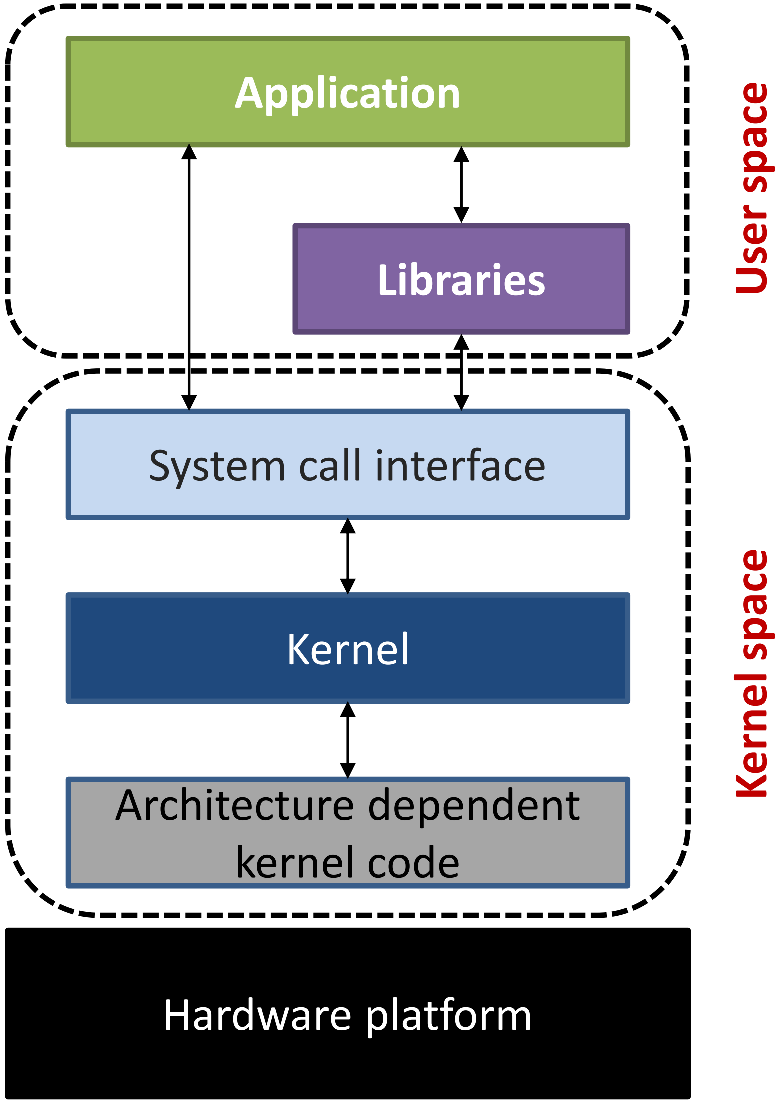
- **Free & Open Source** → GPL license, no cost
- **Reliability** → Build systems with 99.999% upstream
- **Secure** → Monolithic kernel offering high security
- **Scalability** → From mobile phone to stock market servers



What does "Free & Open Source" really mean?

1. The freedom to run the program as you wish, for any purpose (freedom 0)
2. The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1)
3. The freedom to redistribute copies so you can help others (freedom 2)
4. The freedom to distribute copies of your modified versions to others (freedom 3)

Components of a Linux system



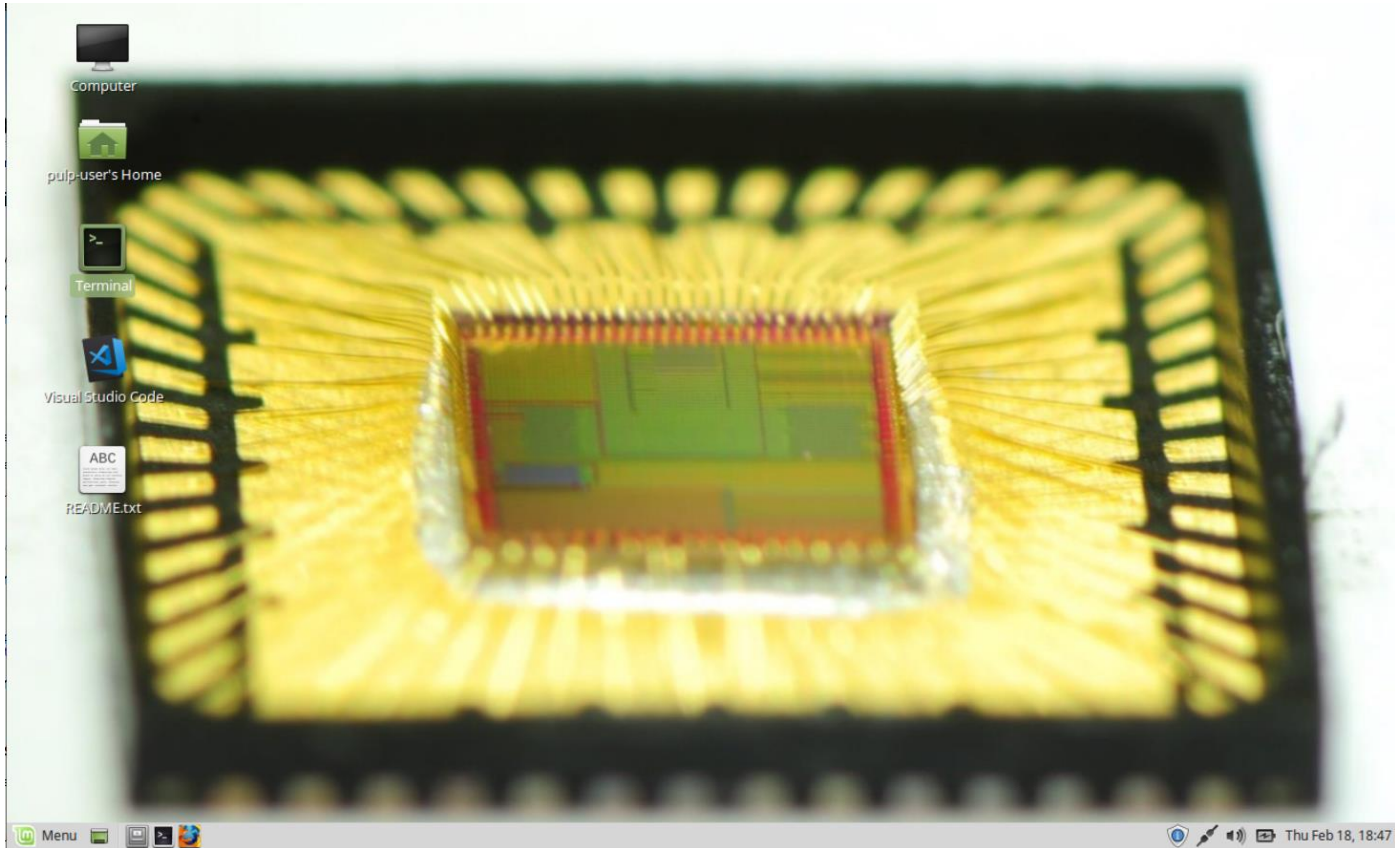
- **Kernel** → the core of the OS, typically a program executed on startup with the aim to control the full system
  - **User space** → execution environment for the user processes
  - **Kernel space** → execution environment for the OS kernel

- The main role of the kernel is to manage **multiple applications** (executed from **multiple users**) from messing with each other and the machine
  - *Hardware resources* (processors/cores, memory, peripherals)
  - *Software resources* (services, data structures, algorithms)

- **System calls** → interface between user programs and OS functions

- **Architecture dependent kernel code** → software layer to abstract the specific hardware

User interface: GUI



Distribution: **Linux Mint 18.3 Sylvia**

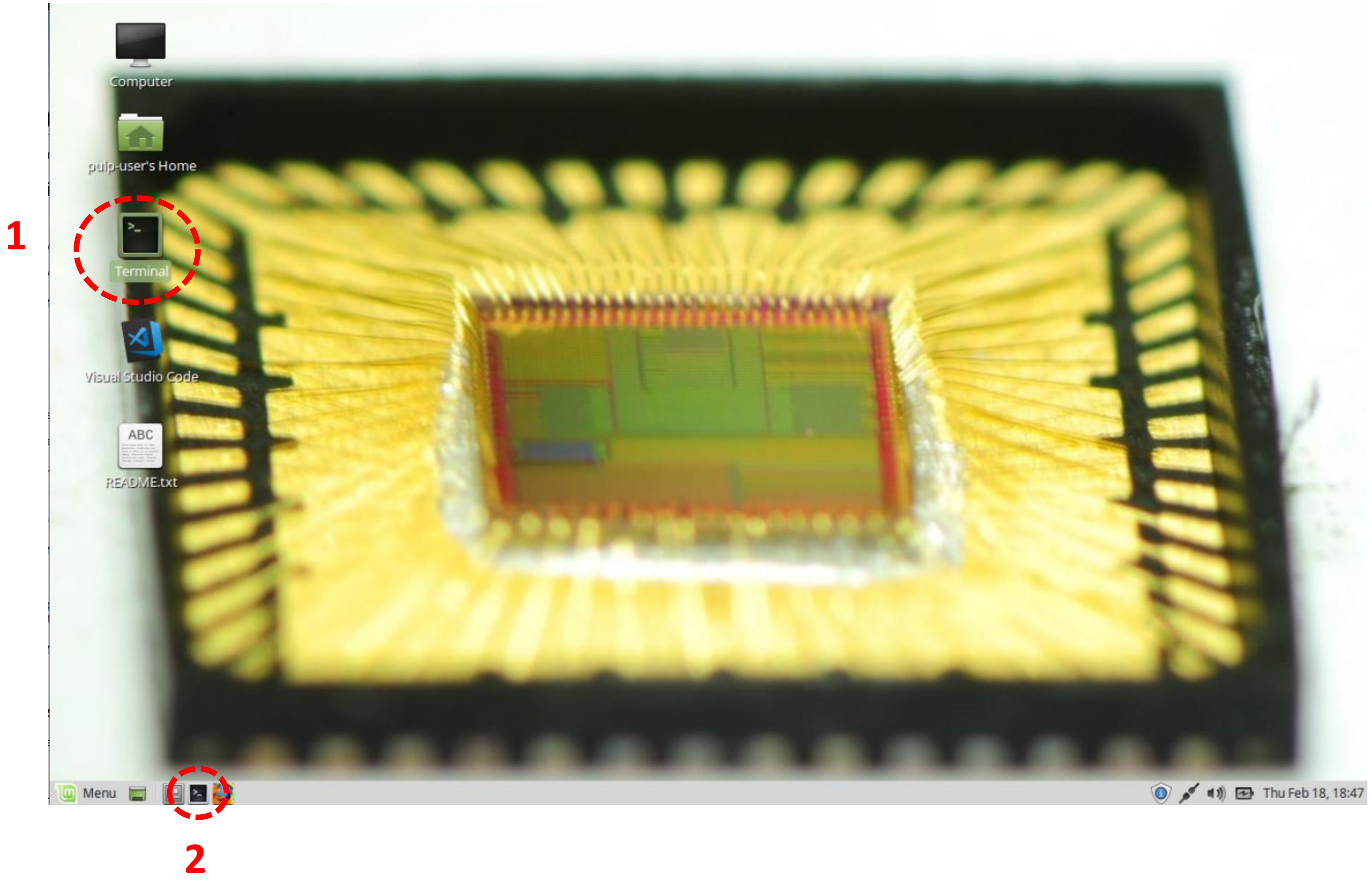
Desktop environment: **MATE**

“An Argentine user of Arch Linux started the MATE project to fork and continue GNOME 2 in response to the negative reception of GNOME 3”

(source: Wikipedia)

User Interface: CLI

The Linux ***command line interface*** (**CLI**), also called ***terminal***, is a textual interface used to execute requested commands



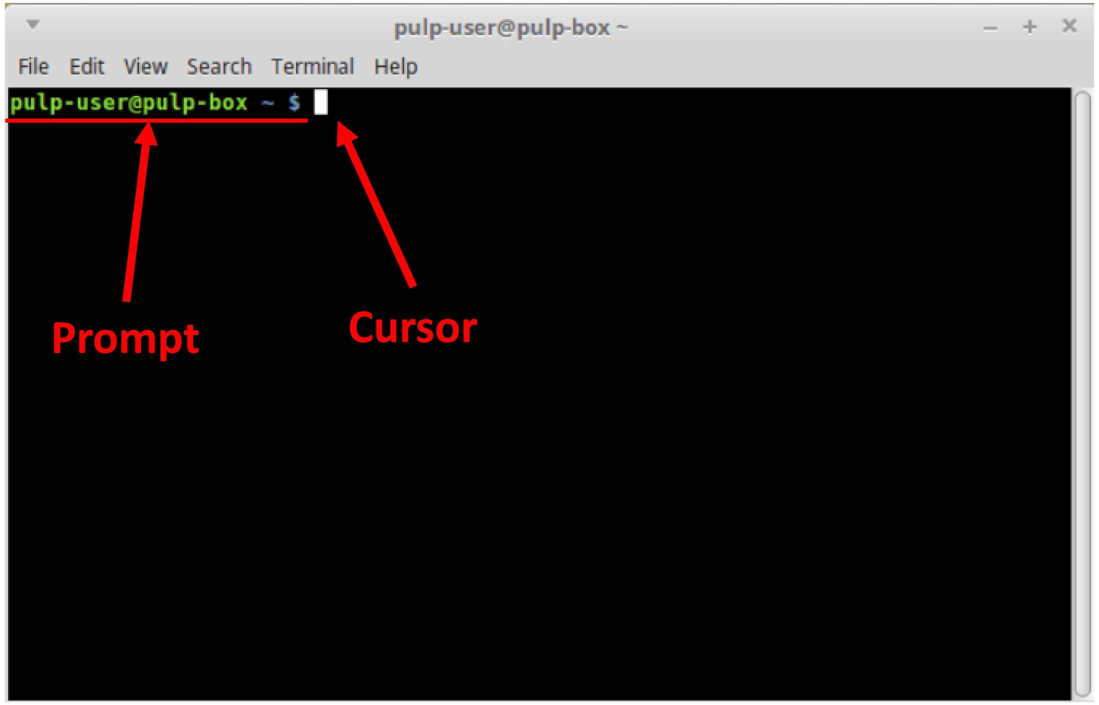
To open the terminal:

1. Desktop icon
2. Taskbar icon
3. Keyboard: CTRL + ALT + T

User Interface: CLI

The interaction with the user is managed by an application called **shell**, working as a command interpreter that

- gets a command from user, executes it through the OS (***job***)
- provides a programming environment to write scripts using an interpreted language (***shell scripts***)
- inherited from the UNIX operating system, which was predecessor to Linux



```
pulp-user@pulp-box ~ $ 
```

Prompt

Cursor

Bourne-Again shell (bash)

Bash is the default interactive shell for users on most Linux systems

Main features:

- **Intelligent completion** of the command line
- **Command history**, which lets you recall previously entered commands
- **Aliases**, which allow you to define shorthand names for commands or command lines
- **Job control**, including the fg and bg commands and the ability to stop jobs with CTRL-Z
- **Conditional execution**, that make execution of a command contingent on the exit code set by a precedent command
- **Brace expansion**, for generating arbitrary strings from a set of alternative combinations
- **Tilde expansion**, a shorthand way to refer to directories
- **Directory manipulation**, with the pushd, popd, and dirs commands

Bash: Basic commands

- **pwd** → print the *absolute* path of the current folder (working directory)
- **ls** → list directories in the current path (option **–l** to get more details)
- **cd** *path* → change directory to the specified *absolute* or *relative* path
  - *Relative paths* are relative to the current folder → **.** points to the current folder, **..** to the parent folder, ~ points to the user home folder
- **whoami** → name of the current user
- **mkdir** *foldername* → create a new folder
- **cp** *source destination* → copy a file
- **mv** *source destination* → move a file or folder
- **rm** *file* → remove a file
- **rm –r** *folder* → recursively remove a folder and its content
- **echo** *argument* → print its argument
- **cat** *file1 file2 file3* → print the files to the terminal
- **wc –l** *file* → count the lines in a file

Bash: Other commands and features

Userful for programmers:

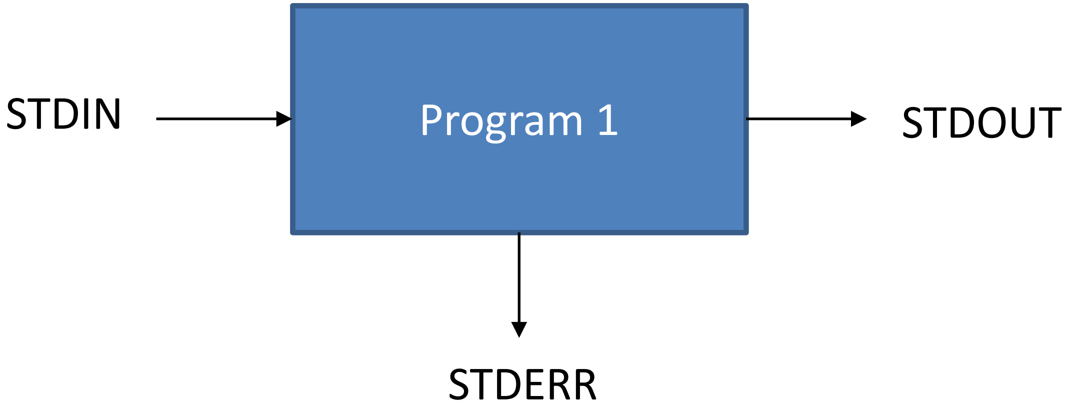
- **wc –l** *file* → count the lines in a file
- **grep –rn** *text folder* → look for text into a specified folder
- **find** *folder* **–name** *filename* → look for a filename in a specified folder
- **touch** *filename* → create a file (or modify tthe modification date of an existing file)
- **less** *filename* → basic viewer to open a file in a terminal (no modification is allowed)
- **vi** / **emacs** → text editor (expert users!!!)

Wildcards are placeholders that can be interpreted as a multiple characters:

- A question mark (**?**) can be used to indicate "any single character" within the file name
- An asterisk (*****) can be used to indicate "zero or more characters"
- Multiple characters can be specified inside square brackets (**[]**) to match a single character

Bash: Redirection and piping

Every program we run on the command line automatically has three data streams connected to it



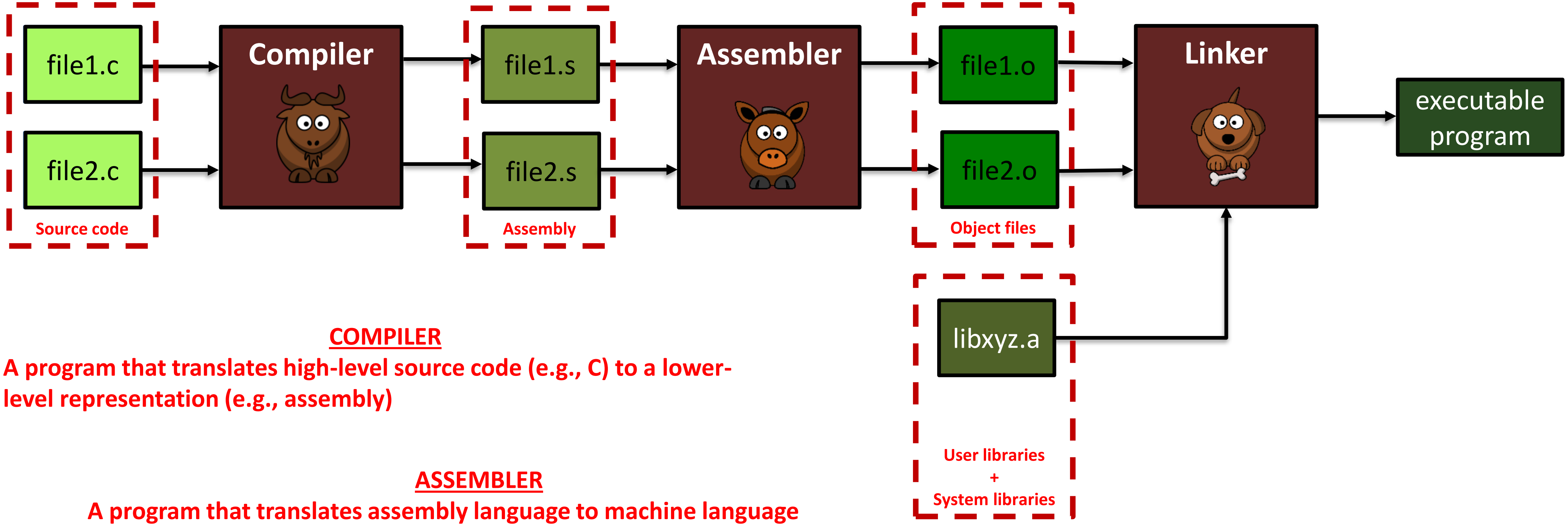
Streams can be **redirected** or connected (**piping**):

- The symbol **<** redirects the standard input from a file (e.g., **wc –l < file_list.txt**)
- The symbol **>** redirects the standard output to a file (e.g., **ls > file_list.txt**)
- The symbol **2>** redirects the standard error to a file (e.g., **ls foo.ttt 2> errors.txt > filename.txt**)
    - The symbol **2&>1** redirects the text in the standard error stream to the standard output stream
- The symbol | is used to connect the output of a command to the input of the next one (e.g., **ls | wc –l**)

This compositional approach is fundamental to integrate different toolchain components

Compilation toolchain

A compilation toolchain includes several tools to achieve its final goal



COMPILER

A program that translates high-level source code (e.g., C) to a lower-level representation (e.g., assembly)

ASSEMBLER

A program that translates assembly language to machine language

LINKER

A program that combines multiple object files into a single executable

Example: compiling and executing on Linux (1/3)

```c
// File: mean.h

float mean(float *, int);
```

```c
// File: mean.c
#include <stdio.h>

float mean(float *vals, int n) {
  float sum = 0.0f;
  for(int i=0; i<n, ++i)
    sum += vals[i];
  return sum/n;
}
```

```c
// File: main.c

#include <stdio.h>
#include "mean.h"

#define N 5

float data[] = {1.2f, -11.0f, -3.23f, 1.14f, 7.9f};

int main() {
  float m = mean(data, N);
  printf("The mean is: %f\n", m);
  return 0;
}
```

Example: compiling and executing on Linux (2/3)

1. Open a Terminal
2. Create a folder for the project: **mkdir mean_test**
3. Enter the folder: **cd mean_test**
4. Open VS Code edit: **code .**
5. Create the source files listed in the previous slide → use the commands in the File menu or press CTRL+N and CTRL+S to create a new file and save it, respectively
6. Invoke the compilation toolchain from the terminal: **gcc mean.c main.c -o myprogram**
7. Execute the program: **./myprogram**

Try it!!!

Example: compiling and executing on Linux (3/3)

- The gcc command in the previous slide invokes all the components of the toolchain
- Alternatively, we can execute the single steps:
  1. gcc -S mean.c -o mean.s
  2. gcc -S main.c -o main.s
  3. gcc -c mean.s -o mean.o
  4. gcc -c main.s -o main.o
  5. gcc main.o mean.o -o myprogram

- Tools for build automation (e.g. make) typically adopt this approach

Build automation: Make

- **Build automation** is the process of automating the main steps required to create a software, including *compiling*, *assembling*, *linking* and (possibly) *testing*
- **Make** is one of the most widespread utilities → configuration files are called **Makefiles**
- A Makefile contains rules in the form:

  ```
  target: prerequisites
  <TAB> command
  ```

  where *target* is filename or a word (included in a special target called .PHONY), *prerequisites* is a list of filenames and other targets, *command* is a shell command
- Programmers can also specify variables
  - Variables are defined by name and read using the notation $(name)
  - different assignment policies (e.g., ?= set a variable only if it has not been already set)
- Automatic variables are used to generalize the rules (e.g., $@ is the filename of the target, $< is the filename of the first prerequisite)
- Usage: **make target1 … targetN var1=val1 … varM=valM**

Makefile for compiler C code on Linux

```
# Executable name
TARGET_EXEC := myprogram

# Source files
SRCS := mean.c main.c

# Compiler flags
CFLAGS ?= -O3

# Linker flags
LDFLAGS ?=

# Directory containing object files and executable
BUILD_DIR ?= ./build

# List of the object files that will be created
OBJS := $(SRCS:%=$(BUILD_DIR)/%.o)

# Link object files
$(BUILD_DIR)/$(TARGET_EXEC): $(OBJS)
    $(CC) $(OBJS) -o $@ $(LDFLAGS)

# Compile C sources
$(BUILD_DIR)/%.c.o: %.c
    mkdir -p $(dir $@)
    $(CC) $(CFLAGS) -c $< -o $@

# Define make targets

.PHONY: all clean

all: $(BUILD_DIR)/$(TARGET_EXEC)

clean:
    $(RM) -r $(BUILD_DIR)
```

Try this Makefile in the Linux environment!

Hint: pay attention to use *tabs* and not *spaces*

Cross compiling

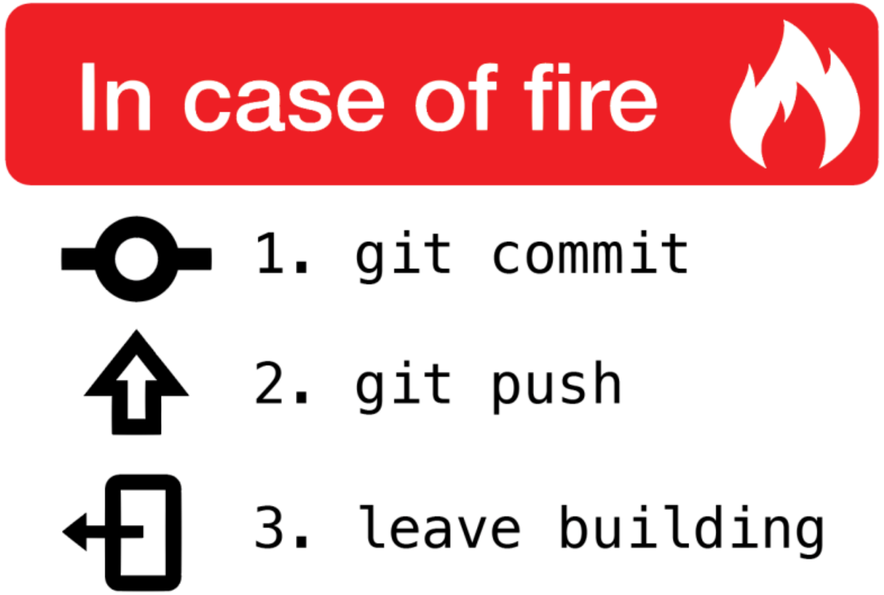
- A **cross compiler** creates binary code for a platform different from the one on which it is running
  - Typically used to compile code for embedded platforms on a development host (e.g., PULP)
- Cross compilation takes into account three environments (*environment = ISA + OS*):
  - **Build**: where we build the compiler
  - **Host**: where we run the compiler
  - **Target**: where we executed the output binary

- Cross compiler for PULP:

  build == host == x86_64-linux
  target = riscv-none

Versioning control systems: Git

- Version control systems record changes to a set of files over time keeping the full history → users can recall specific any version later
- Git is a **distributed version control system**

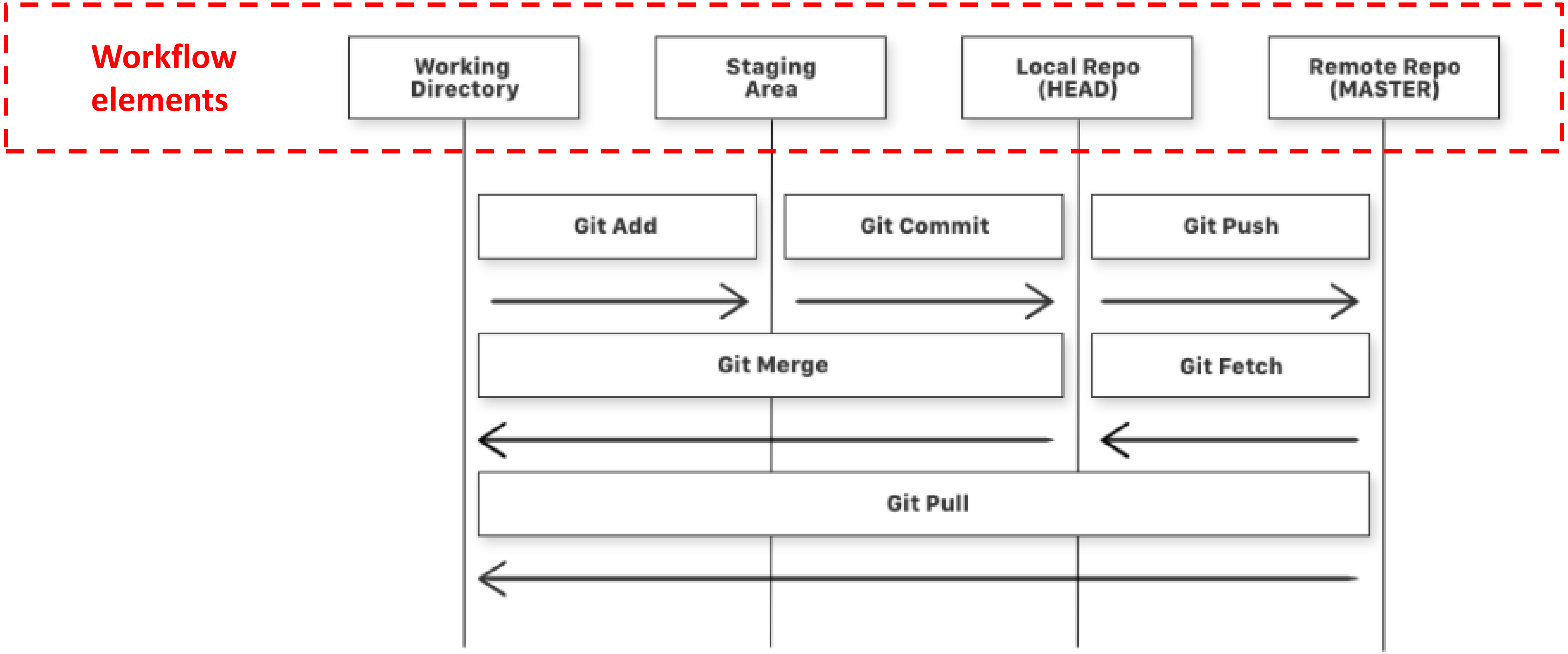


In case of fire

1. git commit
2. git push
3. leave building

Git Workflow

- A **repository (repo)** is a collection of files



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Creating a local copy of a git repository

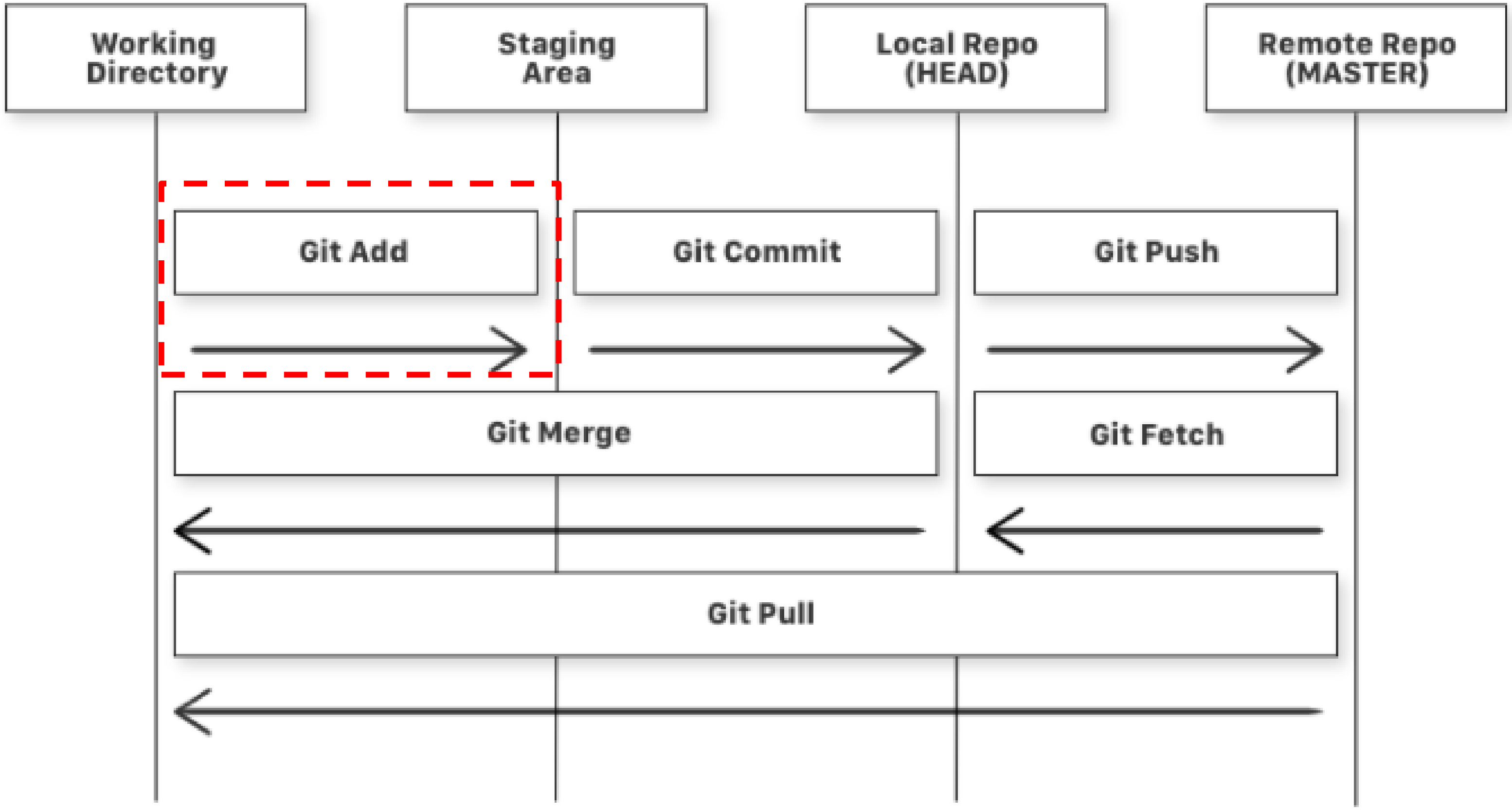
Users can create a Git repository in one of two ways:

1. Entering a local directory that is currently not under version control, and turn it into a Git repository → Follow instructions provided by GitHub after creating a project
2. Clone a remote Git repository: **git clone** *url*

Example: **git clone https://github.com/EEESlab/HSDES-LAB01-PULP_Helloworld**

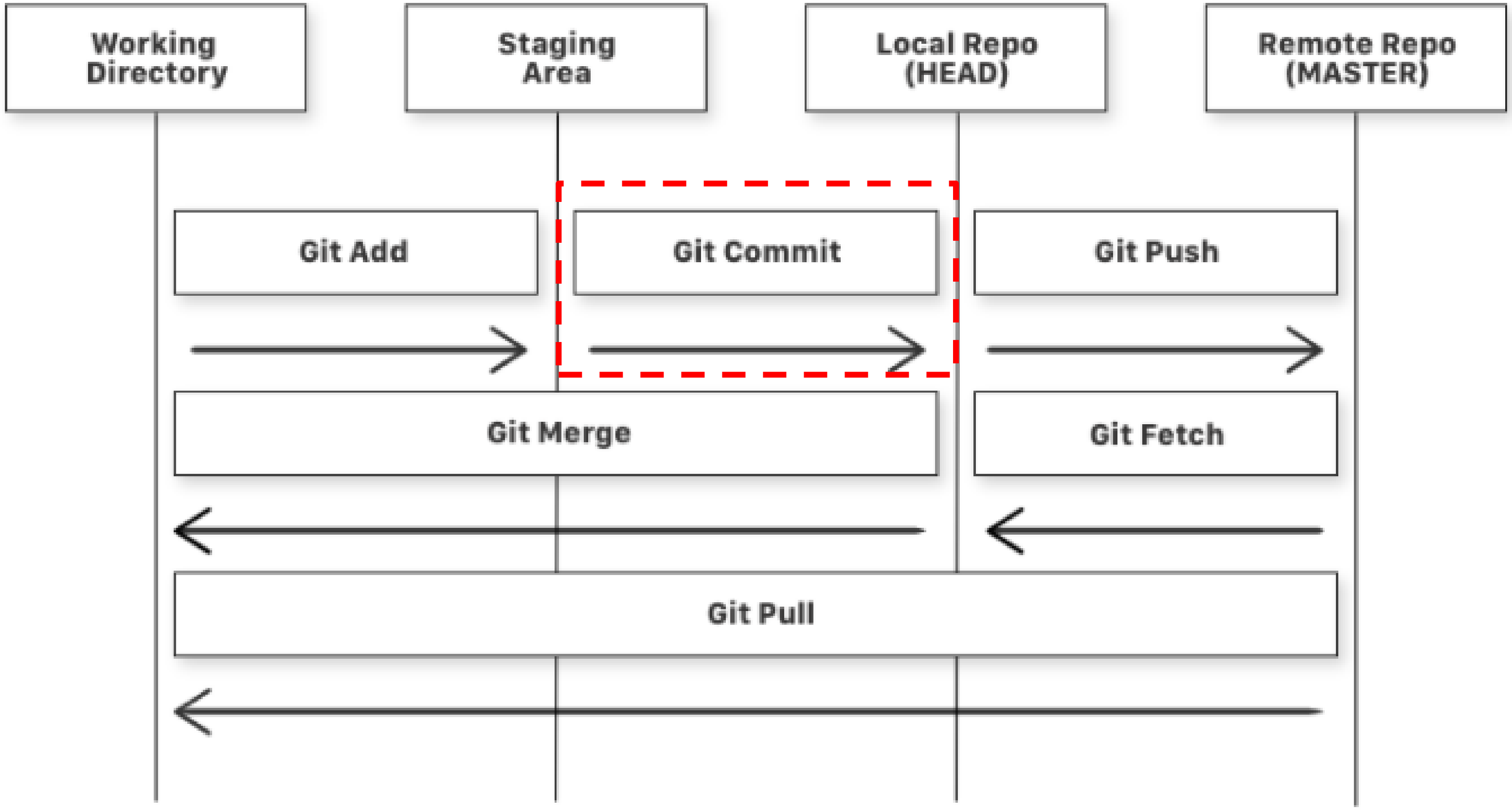
Git Workflow: Stage

- **git add** - marked the changed files to be committed to the local repository (but not yet committed)



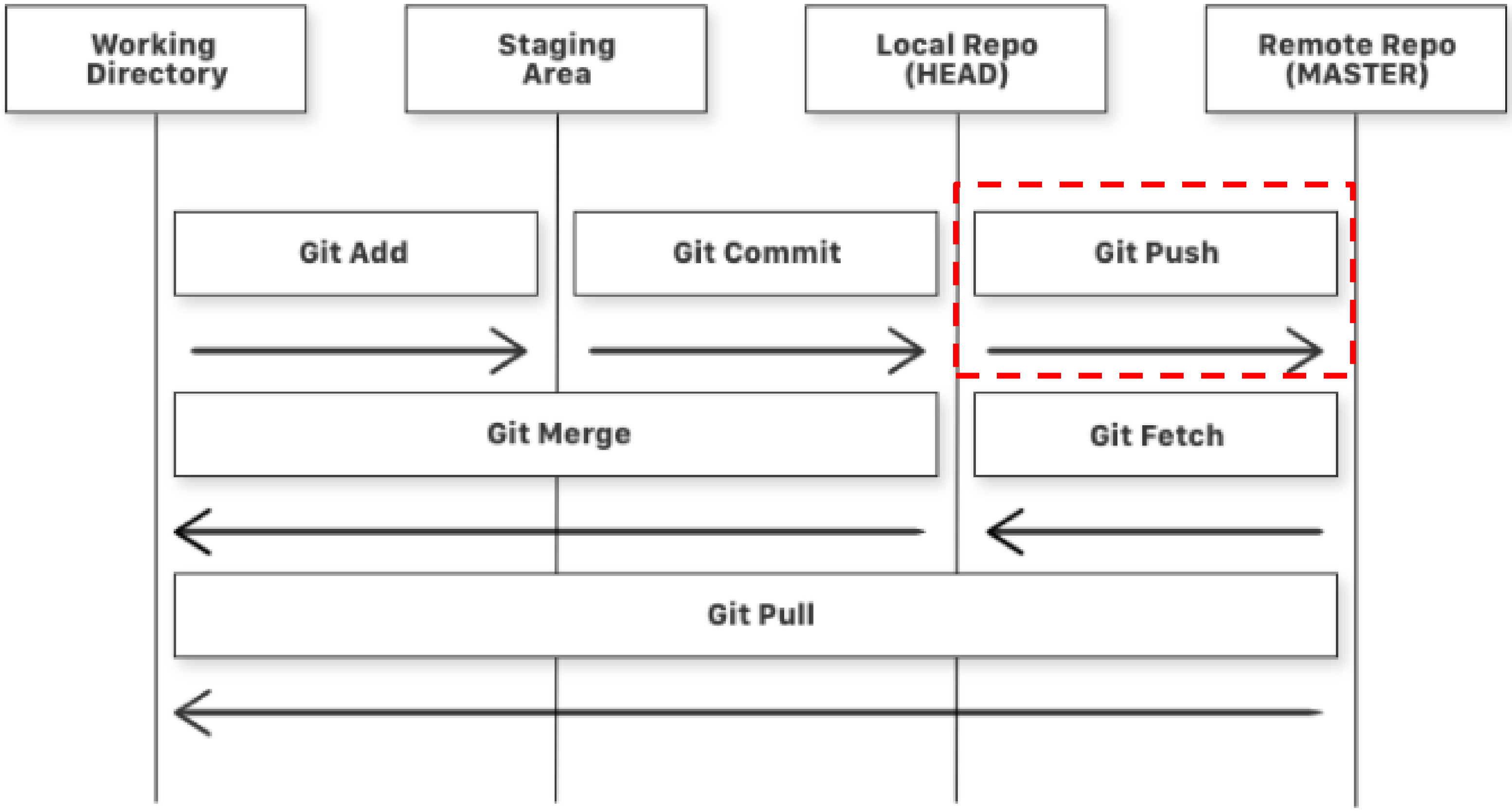
Git Workflow: Commit

- **git commit** - add all files that are staged to the local repository.



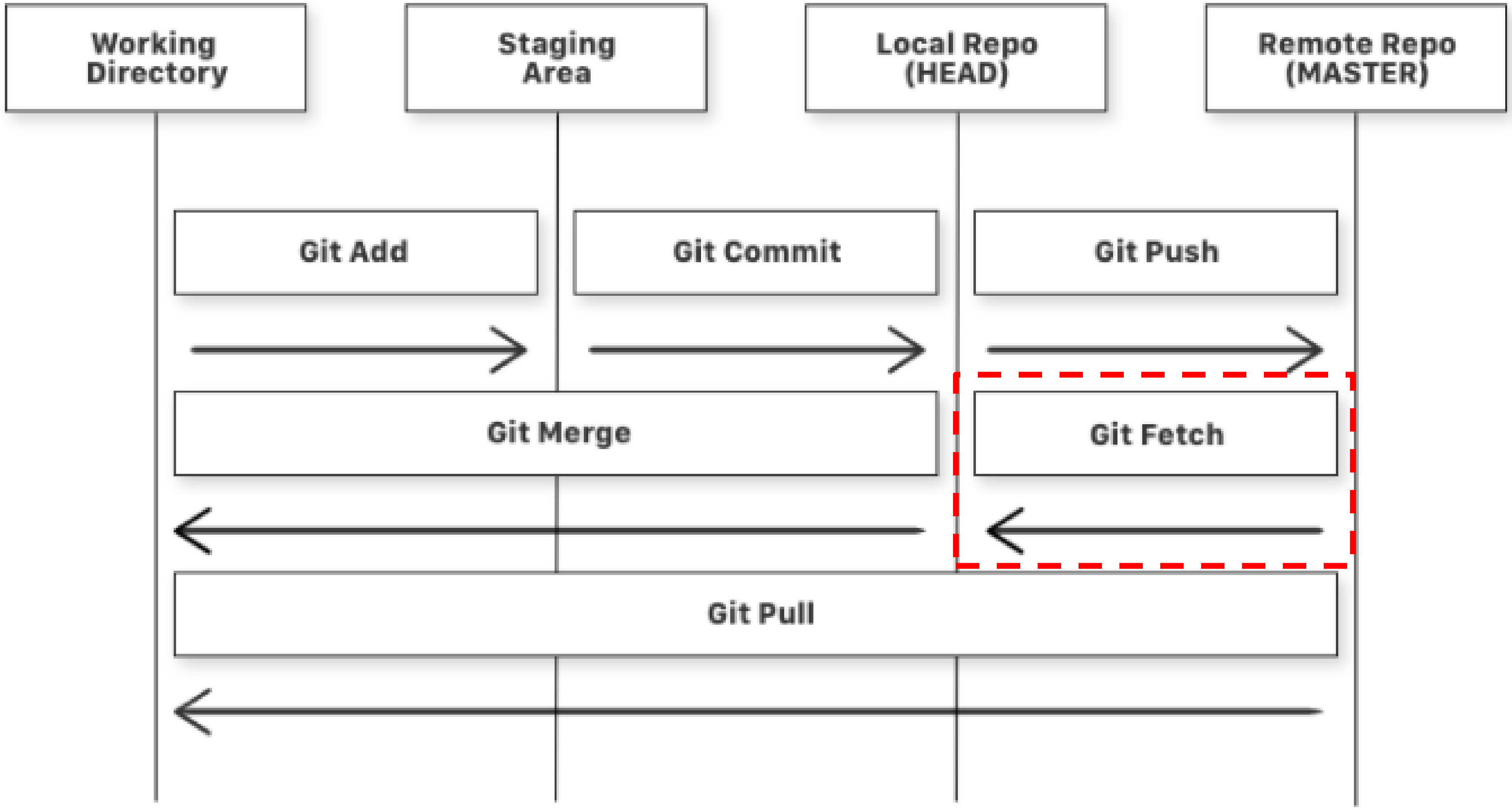
Git Workflow: Push

- **git push** - add all committed files in the local repository to the remote repository



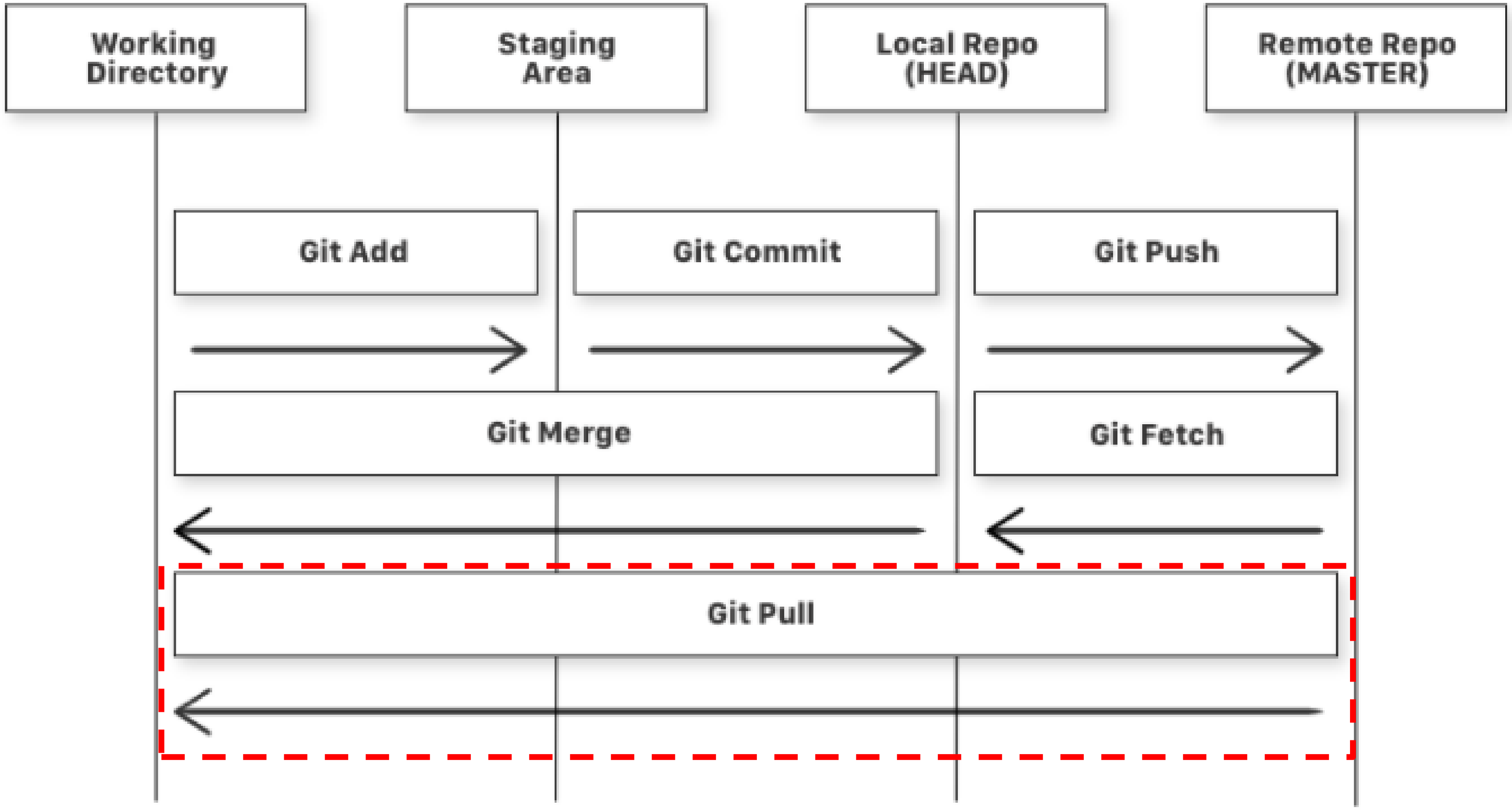
Git Workflow: Fetch

- **git fetch** - get files from the remote repository to the local repository (but not into the working directory)



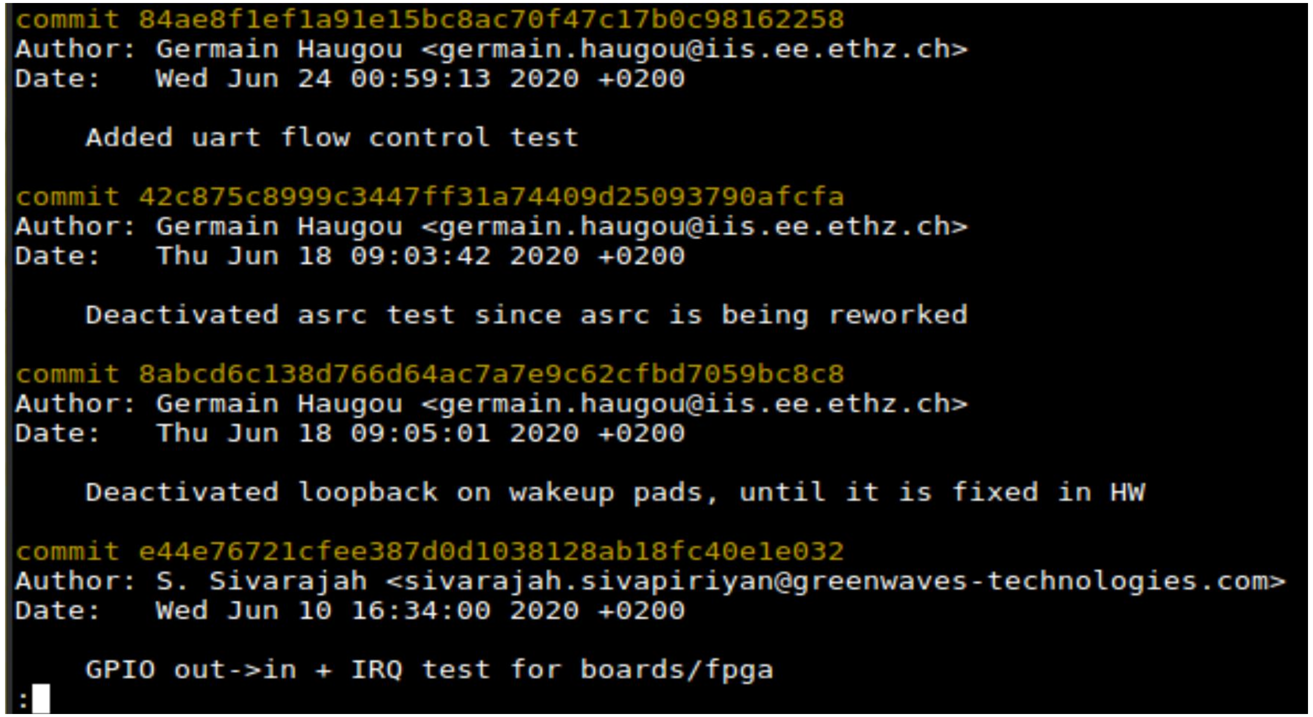
Git Workflow: Pull

- **git pull** - get files from the remote repository directly into the working directory



Repository commit history

- You can inspect the commit history using **git log**
- For example, have a look at the pmsis_tests folder
  - Open a Terminal
  - Move to the pmsis_tests folder with cd
  - Show the commit history
  - Use arrows to scroll up and down
  - Press q to exit the log



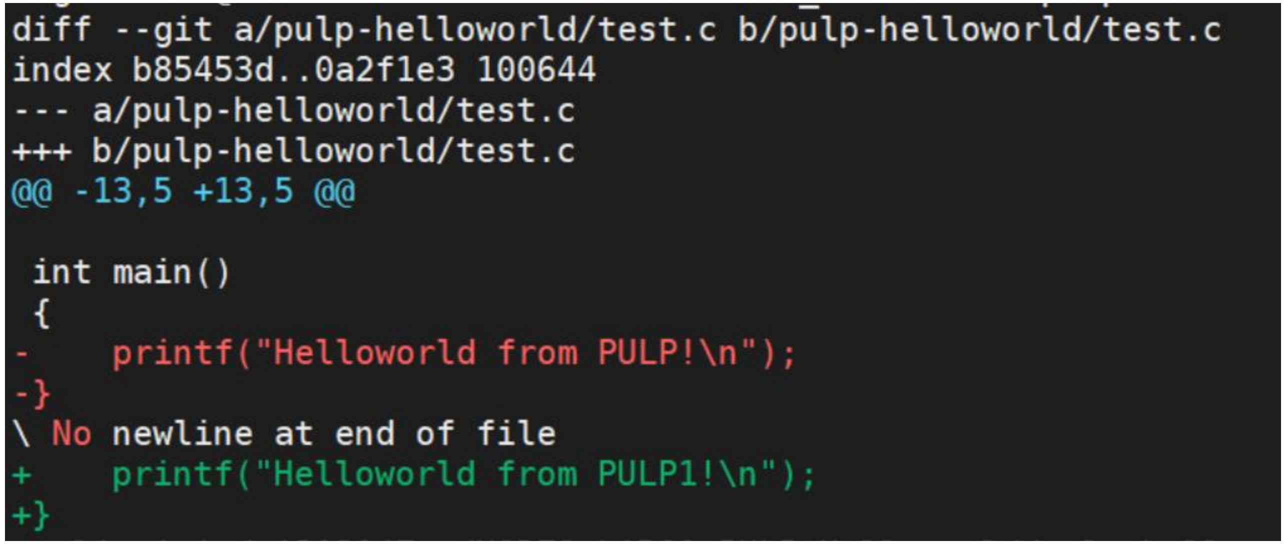
```
commit 84ae8f1ef1a91e15bc8ac70f47c17b0c98162258
Author: Germain Haugou <germain.haugou@iis.ee.ethz.ch>
Date:   Wed Jun 24 00:59:13 2020 +0200

    Added uart flow control test

commit 42c875c8999c3447ff31a74409d25093790afcfa
Author: Germain Haugou <germain.haugou@iis.ee.ethz.ch>
Date:   Thu Jun 18 09:03:42 2020 +0200

    Deactivated asrc test since asrc is being reworked

commit 8abcd6c138d766d64ac7a7e9c62cfbd7059bc8c8
Author: Germain Haugou <germain.haugou@iis.ee.ethz.ch>
Date:   Thu Jun 18 09:05:01 2020 +0200

    Deactivated loopback on wakeup pads, until it is fixed in HW

commit e44e76721cfee387d0d1038128ab18fc40e1e032
Author: S. Sivarajah <sivarajah.sivapiriyan@greenwaves-technologies.com>
Date:   Wed Jun 10 16:34:00 2020 +0200

    GPIO out->in + IRQ test for boards/fpga
:
```

Check changes to a file

To show changes that are not yet staged: **git diff**

Example:

1. Enter a test folder: **cd HSDES-LAB01-PULP_Helloworld/pulp-helloworld/**
2. Modify **test.c** using a text editor
3. Execute **git diff**



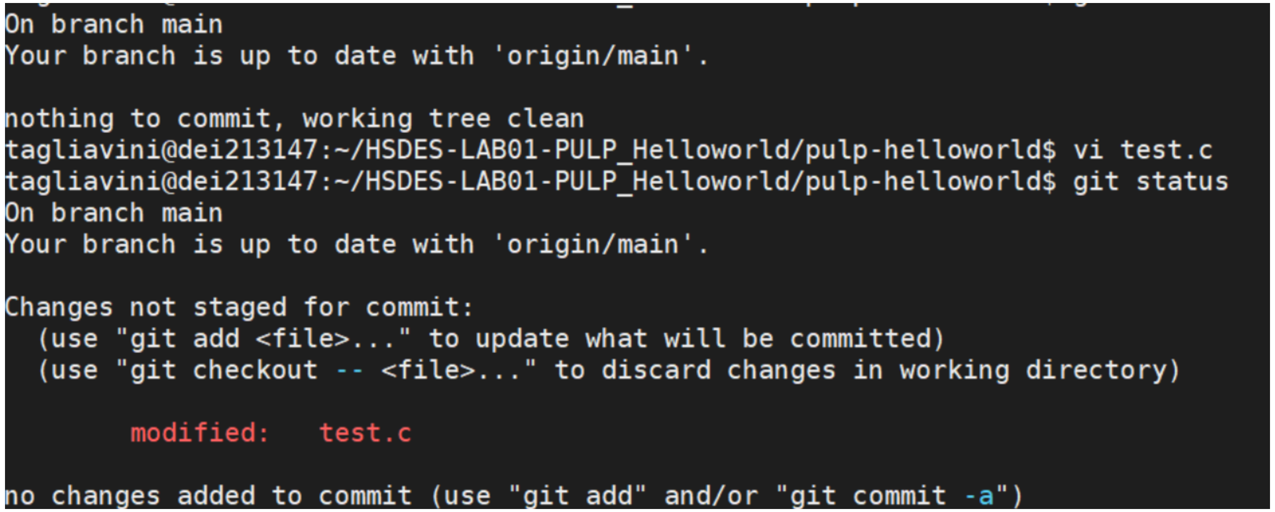
```
diff --git a/pulp-helloworld/test.c b/pulp-helloworld/test.c
index b85453d..0a2f1e3 100644
--- a/pulp-helloworld/test.c
+++ b/pulp-helloworld/test.c
@@ -13,5 +13,5 @@

 int main()
 {
-    printf("Helloworld from PULP!\n");
-}
\ No newline at end of file
+    printf("Helloworld from PULP1!\n");
+}
```

Checking the current status of a repository

- You can inspect the current of the local repository using **git status**

Example:

1. Enter a test folder: **cd HSDES-LAB01-PULP_Helloworld/pulp-helloworld/**
2. Modify **test.c** using a text editor (if not done in the previous example)
3. Execute **git status**



```
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
tagliavini@dei213147:~/HSDES-LAB01-PULP_Helloworld/pulp-helloworld$ vi test.c
tagliavini@dei213147:~/HSDES-LAB01-PULP_Helloworld/pulp-helloworld$ git status
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 checkout -- <file>..." to discard changes in working directory)

        modified:   test.c

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

Revert to the last commit

- We can revert back to the last committed version using **git checkout [options]**
- To revert a specific file: git checkout -- filename

Example:

1. Enter a test folder: **cd HSDES-LAB01-PULP_Helloworld/pulp-helloworld/**
2. Modify **test.c** using a text editor
3. Execute **git checkout test.c**

Git reference: https://git-scm.com/book/en/v2

We recommend having a look at branches and tags, which are very useful in programming practice